

PYTHON

programming language



tutorialspoint

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

Today, Python is one of the most popular programming languages. Although it is a general-purpose language, it is used in various areas of applications such as Machine Learning, Artificial Intelligence, web development, IoT, and more.

This Python tutorial has been written for the beginners to help them understand the basic to advanced concepts of Python Programming Language. After completing this tutorial, you will find yourself at a great level of expertise in Python, from where you can take yourself to the next levels to become a world class Software Engineer.

This Python tutorial is based on the Latest Python 3.11.2 version.

What is Python?

Python is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language. Python is dynamically-typed and garbage-collected programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

Python supports multiple programming paradigms, including Procedural, Object Oriented and Functional programming Language. Python design philosophy emphasizes code readability with the use of significant indentation.

This tutorial gives a complete understanding of Python programming language, starting from basic concepts to advanced concepts. This tutorial will take you through simple and practical approaches while learning Python Programming language.

Python Jobs

Today, Python is very high in demand, and all the major companies are looking for great Python programmers to develop websites, software components, and applications or to work with data science, AI, and ML technologies. When we were developing this tutorial in 2022, there was a high shortage of Python programmers, where the market demanded a greater number of Python programmers due to its applications in machine learning, artificial intelligence, etc.

Today, a Python programmer with 3-5 years of experience is asking for around \$150,000 in an annual package, and this is the most demanding programming language in America. Though it can vary depending on the location of the job. It's impossible to list all of the companies using Python, to name a few big companies are:

- Google
- Intel
- NASA
- PayPal
- Facebook
- IBM
- Amazon
- Netflix
- Pinterest
- Uber
- Many more...

So, you could be the next potential employee for any of these major companies. We have developed great learning material for you to learn Python programming, which will help you prepare for the technical interviews and certification exams based on Python. So, start learning Python using this simple and effective tutorial from anywhere and anytime, absolutely at your pace.

Why to Learn Python?

Python is consistently rated as one of the world's most popular programming languages. Python is fairly easy to learn, so if you are starting to learn any programming language, then Python could be your great choice. Today, various schools, colleges, and universities are teaching Python as their primary programming language. There are many other good reasons that make Python the top choice of any programmer:

- Python is open source, which means it's available free of cost.
- Python is simple and easy to learn.
- Python is versatile and can be used to create different kinds of applications.
- Python has powerful development libraries, including AI, ML, etc.
- Python is much in demand and ensures a high salary.

Python is a MUST for students and working professionals to become great software engineers, especially when they are working in the web development domain. I will list down some of the key advantages of learning Python:

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python "Hello, World!"

To start with Python programming, the very basic program is to print "Hello, World!" You can use the `print()` function. Below is an example of Python code to print "Hello, World!"

```
# Python code to print "Hello, World!"
print ("Hello, World!")
```

Python Online Compiler

Our Python programming tutorial provides various examples to explain different concepts. We have provided Online Python Compiler/Interpreter. You can Edit and Execute almost all the examples directly from your browser without the need to set up your development environment.

Try to click the icon  to run the following Python code to print conventional "Hello, World!".

Below code box allows you to change the value of the code. Try to change the value inside print() and run it again to verify the result.

```
# This is my first Python program.

# This will print 'Hello, World!' as the output

print ("Hello, World!");
```

Careers with Python

If you know Python nicely, then you have a great career ahead. Here are just a few of the career options where Python is a key skill:

- Game developer
- Web designer
- Python developer
- Full-stack developer
- Machine learning engineer
- Data scientist
- Data analyst
- Data engineer
- DevOps engineer
- Software engineer
- Many more other roles

Characteristics of Python

Following are important characteristics of **Python Programming** –

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Applications of Python

Python is a general purpose programming language known for its readability. It is widely applied in various fields.

- In Data Science, Python libraries like Numpy, Pandas, and Matplotlib are used for data analysis and visualization.
- Python frameworks like Django, and Pyramid, make the development and deployment of Web Applications easy.
- This programming language also extends its applications to **computer vision** and image processing.
- It is also favored in many tasks like Automation, Job Scheduling, GUI development, etc.

Features of Python

The latest release of Python is 3.x. As mentioned before, Python is one of the most widely used languages on the web. I'm going to list a few of them here:

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- **A broad standard library** – Python has a bulk of portable and cross-platform libraries and they are compatible with UNIX, Windows, and Macintosh.
- **Interactive Mode** – Python has support for an interactive mode that allows interactive testing and debugging of snippets of code.
- **Portable** – Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** – Python provides interfaces to all major commercial databases.
- **GUI Programming** – Python supports GUI applications that can be created and ported to many system calls, libraries, and operating systems, such as Windows, MFC, Macintosh, and the X Window system of Unix.
- **Scalable** – Python provides a better structure and support for large programs than shell scripting.

Python Reference

The complete function and method references –

- [Python Complete Reference](#)
- [Python Built-in Functions Reference](#)
- [Python Modules Reference](#)
- [Python Keywords Reference](#)
- [Python Cheatsheet](#)

Python Practice

Practice Python from the below-given links:

- [Python Quick Guide](#)
- [Python Online Quiz](#)
- [Python Interview Questions & Answers](#)

Download Python

You can download Python from its official website: <https://www.python.org/downloads/>

Target Audience

This tutorial has been prepared for the beginners to help them understand the basics to advanced concepts of Python programming language. After completing this tutorial, you will find yourself at a great level of expertise in Python programming, from where you can take yourself to the next levels.

Prerequisites

Although it is a beginner's tutorial, we assume that the readers have a reasonable exposure to any programming environment and knowledge of basic concepts such as variables, commands, syntax, etc.

Python Questions & Answers

You can explore a set of Python Questions and Answers at [Python Questions & Answers](#)

Copyright & Disclaimer

© Copyright 2025 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial	i
What is Python?.....	i
Python Reference	iv
Python Practice.....	iv
Download Python	iv
Target Audience.....	iv
Prerequisites.....	v
Python Questions & Answers	v
Table of Contents.....	vi
 PYTHON BASICS	1
1. Python – Overview.....	2
2. Python - History and Versions	4
3. Python - Features	7
4. Python vs C++.....	11
5. Python - Hello World Program	17
6. Python - Application Areas.....	20
7. Python Interpreter and Its Modes	23
8. Python - Environment Setup	28
9. Python - Virtual Environment.....	37
10. Python - Syntax	40
11. Python - Variables	47
12. Python - Data Types	57
13. Python - Type Casting.....	75
14. Python - Unicode System	85
15. Python - Literals	88
16. Python - Operators.....	94

17. Python - Arithmetic Operators	104
18. Python - Comparison Operators	115
19. Python - Assignment Operators	121
20. Python - Logical Operators	128
21. Python - Bitwise Operators	132
22. Python - Membership Operators.....	137
23. Python - Identity Operators	141
24. Python Operator Precedence	144
25. Python - Comments.....	147
26. Python - User Input	151
27. Python - Numbers	157
28. Python - Booleans	174
 PYTHON CONTROL STATEMENTS	176
29. Python - Control Flow.....	177
30. Python - Decision Making.....	183
31. Python - if Statement	186
32. Python if-else Statement.....	189
33. Python - Nested if Statement	195
34. Python - Match-Case Statement.....	198
35. Python - Loops	201
36. Python - For Loops	203
37. Python for-else Loops.....	209
38. Python - While Loops	213
39. Python - break Statement	218
40. Python - Continue Statement.....	221
41. Python - pass Statement	224
42. Python - Nested Loops	226

PYTHON FUNCTIONS & MODULES	229
43. Python - Functions	230
44. Python - Default Arguments.....	243
45. Python - Keyword Arguments	246
46. Python - Keyword-Only Arguments.....	249
47. Python - Positional Arguments.....	251
48. Python - Positional-Only Arguments	254
49. Python - Arbitrary or, Variable-length Arguments.....	256
50. Python Variable Scope	259
51. Python - Function Annotations.....	266
52. Python - Modules.....	270
53. Python - Built-in Functions	281
 PYTHON STRINGS	287
54. Python - Strings.....	288
55. Python Slicing Strings	298
56. Python - Modify Strings.....	304
57. Python - String Concatenation.....	307
58. Python - String Formatting	310
59. Python - Escape Characters	312
60. Python - String Methods	316
61. Python - String Exercises	321
 PYTHON LISTS	323
62. Python - Lists.....	324
63. Python - Access List Items	328
64. Python - Change List Items	331
65. Python - Add List Items	333
66. Python - Remove List Items	335

67. Python - Loop Lists	338
68. Python - List Comprehension	342
69. Python - Sort Lists	346
70. Python - Copy Lists	349
71. Python - Join Lists.....	353
72. Python - List Methods	356
73. Python - List Exercises	359
 PYTHON TUPLES.....	361
74. Python - Tuples	362
75. Python - Access Tuple Items.....	366
76. Python - Update Tuples.....	370
77. Python - Unpack Tuple Items	374
78. Python - Loop Tuples.....	376
79. Python - Join Tuples	379
80. Python - Tuple Methods.....	383
81. Python Tuple Exercises.....	386
 PYTHON SETS	388
82. Python - Sets	389
83. Python - Access Set Items.....	394
84. Python - Add Set Items.....	397
85. Python - Remove Set Items	401
86. Python - Loop Sets.....	408
87. Python - Join Sets	412
88. Python - Copy Sets	416
89. Python - Set Operators.....	419
90. Python - Set Methods.....	424
91. Python - Set Exercises	426

PYTHON DICTIONARIES	427
92. Python - Dictionaries	428
93. Python - Access Dictionary Items	436
94. Python - Change Dictionary Items	442
95. Python - Add Dictionary Items	446
96. Python - Remove Dictionary Items	451
97. Python - Dictionary View Objects	455
98. Python - Loop Dictionaries	458
99. Python - Copy Dictionaries	461
100. Python - Nested Dictionaries	465
101. Python - Dictionary Methods	470
102. Python - Dictionary Exercises	471
PYTHON ARRAYS	473
103. Python - Arrays	474
104. Python - Access Array Items	479
105. Python - Add Array Items	482
106. Python - Remove Array Items	484
107. Python - Loop Arrays	486
108. Python - Copy Arrays	488
109. Python - Reverse Arrays	490
110. Python - Sort Arrays	493
111. Python - Join Arrays	496
112. Python - Array Methods	498
113. Python - Array Exercises	501
PYTHON FILE HANDLING	503
114. Python - File Handling	504
115. Python - Write to File	510

116. Python - Read Files.....	515
117. Python - Renaming and Deleting Files	523
118. Python - Directories	525
119. Python - File Methods	529
120. Python OS File/Directory Methods.....	531
121. Python OS.Path Methods	535
OBJECT ORIENTED PROGRAMMING.....	538
122. Python - OOP Concepts	539
123. Python - Classes and Objects.....	547
124. Python - Class Attributes	555
125. Python - Class Methods.....	559
126. Python - Static Methods.....	563
127. Python - Constructors.....	566
128. Python - Access Modifiers	571
129. Python - Inheritance.....	576
130. Python - Polymorphism.....	584
131. Python - Method Overriding	588
132. Python - Method Overloading	591
133. Python - Dynamic Binding	594
134. Python - Dynamic Typing.....	597
135. Python - Abstraction	599
136. Python - Encapsulation.....	601
137. Python - Interfaces	604
138. Python - Packages	607
139. Python - Inner Classes	611
140. Python - Anonymous Class and Objects.....	615
141. Python - Singleton Class	617
142. Python - Wrapper Classes.....	619

143. Python - Enums	620
144. Python - Reflection.....	624
 PYTHON ERRORS & EXCEPTIONS.....	631
145. Python - Syntax Errors.....	632
146. Python - Exceptions Handling.....	635
147. Python - The try-except Block	644
148. Python - The try-finally Block	648
149. Python - Raising Exceptions.....	651
150. Python - Exception Chaining.....	655
151. Python - Nested try Block.....	658
152. Python - User-Defined Exceptions	661
153. Python - Logging.....	665
154. Python - Assertions	669
155. Python - Built-in Exceptions	671
 PYTHON MULTITHREADING	679
156. Python - Multithreading.....	680
157. Python - Thread Lifecycle	688
158. Python - Creating a Thread.....	692
159. Python - Starting a Thread.....	697
160. Python - Joining the Threads	700
161. Python - Naming the Threads.....	703
162. Python - Thread Scheduling.....	707
163. Python - Thread Pools	711
164. Python - Main Thread.....	715
165. Python - Thread Priority	719
166. Python - Daemon Threads	725
167. Python - Synchronizing Threads	729

PYTHON SYNCHRONIZATION.....	734
168. Python - Inter-Thread Communication.....	735
169. Python - Thread Deadlock	741
170. Python - Interrupting a Thread	746
PYTHON NETWORKING	749
171. Python - Network Programming.....	750
172. Python - Socket Programming	751
173. Python - URL Processing.....	760
174. Python - Generics	766
PYTHON LIBRARIES.....	768
175. NumPy Tutorial	769
176. Python Pandas Tutorial	772
177. SciPy Tutorial	777
178. Matplotlib Tutorial.....	778
179. Django Tutorial	784
180. OpenCV Python Tutorial.....	787
PYTHON MISCELLANEOUS.....	788
181. Python - Date and Time.....	789
182. Python - math Module	807
183. Python - Iterators.....	813
184. Python - Generators.....	819
185. Python - Closures	825
186. Python - Decorators	829
187. Python - Recursion	835
188. Python - Regular Expressions	839
189. Python - PIP.....	854
190. Python - Database Access.....	859

191. Python - Weak References	869
192. Python - Serialization	875
193. Python - Templating.....	882
194. Python - Output Formatting.....	888
195. Python - Performance Measurement	897
196. Python - Data Compression	900
197. Python - CGI Programming	901
198. Python - XML Processing	916
199. Python - GUI Programming.....	927
200. Python - Command-Line Arguments.....	945
201. Python - Docstrings	952
202. Python - JSON.....	960
203. Python - Sending Email.....	965
204. Python - Further Extensions	974
205. Python - Tools/Utilities	983
206. Python - GUIs	987
 PYTHON ADVANCED CONCEPTS.....	994
207. Python - Abstract Base Classes	995
208. Python - Custom Exceptions.....	1000
209. Python - Higher Order Functions	1003
210. Python - Object Internals	1007
211. Python - Memory Management	1010
212. Python - Metaclasses	1014
213. Python - Metaprogramming with Metaclasses.....	1018
214. Python - Mocking and Stubbing	1021
215. Python - Monkey Patching	1024
216. Python - Signal Handling	1026
217. Python - Type Hints	1030

218. Python - Automation Tutorial	1040
219. Python - Humanize Package	1046
220. Python - Context Managers.....	1052
221. Python - Coroutines	1057
222. Python - Descriptors.....	1063
223. Python - Diagnosing and Fixing Memory Leaks.....	1071
224. Python - Immutable Data Structures	1076

Python Basics

1. Python – Overview

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation. It has fewer syntactical constructions than other languages.

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python is an open-source and cross-platform programming language. It is available for use under **Python Software Foundation License** (compatible to GNU General Public License) on all the major operating system platforms such as Linux, Windows and Mac OS.

To facilitate new features and to maintain that readability, the Python Enhancement Proposal (PEP) process was developed. This process allows anyone to submit a PEP for a new feature, library, or other addition.

The design philosophy of Python emphasizes on simplicity, readability and unambiguity. Python is known for its batteries included approach as Python software is distributed with a comprehensive standard library of functions and modules.

Python's design philosophy is documented in the Zen of Python. It consists of nineteen aphorisms such as –

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated

To obtain the complete Zen of Python document, type import this in the Python Shell –

```
>>>import this
```

This will produce following 19 aphorisms -

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.
```

Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Python supports imperative, structured as well as object-oriented programming methodology. It provides features of functional programming as well.

Pythontic Code Style

Python leaves you free to choose to program in an object-oriented, procedural, functional, aspect-oriented, or even logic-oriented way. These freedoms make Python a great language to write clean and beautiful code.

Pythontic Code Style is actually more of a design philosophy and suggests to write a code which is:

- Clean
- Simple
- Beautiful
- Explicit
- Readable

The Zen of Python

The Zen of Python is about code that not only works, but is Pythontic. Pythontic code is readable, concise, and maintainable.

2. Python - History and Versions

History of Python

Python was developed by Guido van Rossum (a Dutch programmer) in the late 1980s and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages. Guido van Rossum wanted Python to be a high-level language that was powerful yet readable and easy to use.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Being the principal architect of Python, the developer community conferred upon him the title of **Benevolent Dictator for Life** (BDFL). However, in 2018, Rossum relinquished the title. Thereafter, the development and distribution of the reference implementation of Python is handled by a nonprofit organization **Python Software Foundation**.

Who Invented Python?

Python was invented by a Dutch Programmer Guido Van Rossum in the late 1980s. He began working on Python in December 1989 as a hobby project while working at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. Python's first version (0.9.0) was released in 1991.

Evolution of Python – The Major Python Versions

Following are the important stages in the history of Python –

Python 0.9.0

Python's first published version is 0.9. It was released in February 1991. It consisted of features such as classes with inheritance, exception handling, and core data types like lists and dictionaries.

Python 1.0

In January 1994, version 1.0 was released, armed with functional programming tools, features like support for complex numbers etc. and module system which allows a better code organization and reuse.

Python 2.0

Next major version – Python 2.0 was launched in October 2000. Many new features such as list comprehension, garbage collection and Unicode support were included with it. Throughout the 2000s, Python 2.x became the dominant version, gaining traction in industries ranging from web development to scientific research. Various useful libraries like NumPy, SciPy, and Django were also developed.

Python 3.0

Python 3.0, a completely revamped version of Python was released in December 2008. The primary objective of this revamp was to remove a lot of discrepancies that had crept in Python 2.x versions. Python 3 was backported to Python 2.6. It also included a utility named as `python2to3` to facilitate automatic translation of Python 2 code to Python 3. Python 3 provided new syntax, Unicode support and improved integer division.

EOL for Python 2.x

Even after the release of Python 3, Python Software Foundation continued to support the Python 2 branch with incremental micro versions till 2019. However, it decided to discontinue the support by the end of year 2020, at which time Python 2.7.17 was the last version in the branch.

Current Version of Python

Meanwhile, more and more features have been incorporated into Python's 3.x branch. As of date, Python **3.11.2** is the current stable version, released in February 2023.

What's New in Python 3.11?

One of the most important features of Python's version 3.11 is the significant improvement in speed. According to Python's official documentation, this version is faster than the previous version (3.10) by up to 60%. It also states that the standard benchmark suite shows a 25% faster execution rate.

- Python 3.11 has a better exception messaging. Instead of generating a long traceback on the occurrence of an exception, we now get the exact expression causing the error.
- As per the recommendations of PEP 678, the `add_note()` method is added to the `BaseException` class. You can call this method inside the `except` clause and pass a custom error message.
- It also adds the `cbrt()` function in the `maths` module. It returns the cube root of a given number.
- A new module `tomllib` is added in the standard library. TOML (Tom's Obvious Minimal Language) can be parsed with `tomllib` module function.

Python in the Future

Python is evolving every day and Python 3.x is receiving regular updates. Python's developer community is focusing on performance improvements making it more efficient while retaining its ease of use.

Python is being heavily used for machine learning, AI, and data science, so for sure its future remains bright. Its role in these rapidly growing fields ensures that Python will stay relevant for years.

Python is also increasingly becoming the first programming language taught in schools and universities worldwide, solidifying its place in the tech landscape.

Frequently Asked Questions About Python History

1. Who created Python?

Python created by Guido Van Rossum, a Dutch Programmer.

2. Why Python is called Python?

Python does not have any relation to Snake. The name of the Python programming language was inspired by a British Comedy Group Monty Python.

3. When was Python's first version released?

Python's first version was released in February 1991.

4. What was the first version of Python?

Python's first version was Python 0.9.0

5. When was Python 3.0 version released?

Python 3.0 version was released in December 2008.

3. Python - Features

Python is a feature-rich, high-level, interpreted, interactive, and object-oriented scripting language. Python is a versatile and very popular programming language due to its features such as readability, simplicity, extensive libraries, and many more. In this tutorial, we will learn about the various features of Python that make it a powerful and versatile programming language.

1. Easy To Learn

2. Interpreter Based

3. Interactive

4. Multi-Paradigm

5. Large Standard Library

6. Open source & Cross-Platform

7. GUI Development

8. Database Connectivity

9. Extensible

10. Developer Community



Python – Important Features

Features of Python

Python's most important features are as follows:

- Easy to Learn
- Dynamically Typed
- Interpreter Based
- Interactive
- Multi-paradigm
- Standard Library
- Open Source and Cross Platform
- GUI Applications
- Database Connectivity
- Extensible
- Active Developer Community

Easy to Learn

This is one of the most important reasons for the popularity of Python. Python has a limited set of keywords. Its features such as simple syntax, usage of indentation to avoid clutter of curly brackets, and dynamic typing that doesn't necessitate prior declaration of variable help a beginner to learn Python quickly and easily.

Dynamically Typed

Python is a dynamically typed programming language. In Python, you don't need to specify the variable at the time of the declaration. The types are specified at the runtime based on the assigned value due to its dynamically typed feature.

Interpreter Based

Instructions in any programming languages must be translated into machine code for the processor to execute them. Programming languages are either compiler based or interpreter based.

In case of a compiler, a machine language version of the entire source program is generated. The conversion fails even if there is a single erroneous statement. Hence, the development process is tedious for the beginners. The C family languages (including C, C++, Java, C# etc.) are compiler based.

Python is an interpreter based language. The interpreter takes one instruction from the source code at a time, translates it into machine code and executes it. Instructions before the first occurrence of error are executed. With this feature, it is easier to debug the program and thus proves useful for the beginner level programmer to gain confidence gradually. Python therefore is a beginner-friendly language.

Interactive

Standard Python distribution comes with an interactive shell that works on the principle of REPL (Read – Evaluate – Print – Loop). The shell presents a Python prompt `>>>`. You can type any valid Python expression and press Enter. Python interpreter immediately returns the response and the prompt comes back to read the next expression.

```
>>> 2*3+1
7
>>> print ("Hello World")
Hello World
```

The interactive mode is especially useful to get familiar with a library and test out its functionality. You can try out small code snippets in interactive mode before writing a program.

Multi-paradigm

Python is a completely object-oriented language. Everything in a Python program is an object. However, Python conveniently encapsulates its object orientation to be used as an imperative or procedural language – such as C. Python also provides certain functionalities that resemble functional programming. Moreover, certain third-party tools have been developed to support other programming paradigms such as aspect-oriented and logic programming.

Standard Library

Even though it has a very few keywords (only Thirty-Five), Python software is distributed with a standard library made of large number of modules and packages. Thus Python has out of box support for programming needs such as serialization, data compression,

internet data handling, and many more. Python is known for its batteries included approach.

Some of the Python's popular modules are:

- [NumPy](#)
- [Pandas](#)
- [Matplotlib](#)
- Tkinter
- [Math](#)

Open Source and Cross Platform

Python's standard distribution can be downloaded from

<https://www.python.org/downloads/> without any restrictions. You can download pre-compiled binaries for various operating systems. In addition, the source code is also freely available, which is why it comes under open source category.

Python software (along with the documentation) is distributed under Python Software Foundation License. It is a BSD style permissive software license and compatible to GNU GPL (General Public License).

Python is a cross-platform language. Pre-compiled binaries are available for use on various operating systems such as Windows, Linux, Mac OS, Android OS. The reference implementation of Python is called CPython and is written in C. You can download the source code and compile it for your OS platform.

A Python program is first compiled to an intermediate platform independent byte code. The virtual machine inside the interpreter then executes the byte code. This behavior makes Python a cross-platform language, and thus a Python program can be easily ported from one OS platform to other.

GUI Applications

Python's standard distribution has an excellent graphics library called Tkinter. It is a Python port for the vastly popular GUI toolkit called TCL/Tk. You can build attractive user-friendly GUI applications in Python. GUI toolkits are generally written in C/C++. Many of them have been ported to Python. Examples are PyQt, WxWidgets, PySimpleGUI etc.

Database Connectivity

Almost any type of database can be used as a backend with the Python application. DB-API is a set of specifications for database driver software to let Python communicate with a relational database. With many third party libraries, Python can also work with NoSQL databases such as MongoDB.

Extensible

The term extensibility implies the ability to add new features or modify existing features. As stated earlier, CPython (which is Python's reference implementation) is written in C. Hence one can easily write modules/libraries in C and incorporate them in the standard library. There are other implementations of Python such as Jython (written in Java) and IPython (written in C#). Hence, it is possible to write and merge new functionality in these implementations with Java and C# respectively.

Active Developer Community

As a result of Python's popularity and open-source nature, a large number of Python developers often interact with online forums and conferences. Python Software Foundation also has a significant member base, involved in the organization's mission to "**Promote, Protect, and Advance the Python Programming Language**"

Python also enjoys a significant institutional support. Major IT companies Google, Microsoft, and Meta contribute immensely by preparing documentation and other resources.

Apart from the above-mentioned features, Python has another big list of good features, few are listed below –

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

4. Python vs C++

Python is a general-purpose, high-level programming language. Python is used for web development, Machine Learning, and other cutting-edge software development technologies. Python is suitable for both new and seasoned C++ and Java programmers. **Guido Van Rossum** has created Python in 1989 at Netherlands' National Research Institute. Python was released in 1991.

C++ is a middle-level, case-sensitive, object-oriented programming language. Bjarne Stroustrup created C++ at Bell Labs. C++ is a platform-independent programming language that works on Windows, Mac OS, and Linux. C++ is near to hardware, allowing low-level programming. This provides a developer control over memory, improved performance, and dependable software.

Read through this article to get an overview of C++ and Python and how these two programming languages are different from each other.

What is Python?

Python is currently one of the most widely used programming languages. It is an interpreted programming language that operates at a high level. When compared to other languages, the learning curve for Python is much lower, and it is also quite straightforward to use.

Python is the programming language of choice for professionals working in fields such as Artificial Intelligence, Machine Learning (ML), Data Science, the Internet of Things (IoT), etc., because it excels at both scripting applications and as standalone programmers.

In addition to this, Python is the language of choice because it is easy to learn. Because of its excellent syntax and readability, the amount of money spent on maintenance is decreased. The modularity of the program and the reusability of the code both contribute to its support for a variety of packages and modules.

Using Python, we can perform –

- Web development
- Data analysis and machine learning
- Automation and scripting
- Software testing and many more

Features

Here is a list of some of the important features of Python –

- **Easy to learn** – Python has a simple structure, few keywords, and a clear syntax. This makes it easy for the student to learn quickly. Code written in Python is easier to read and understand.
- **Easy to maintain** – The source code for Python is pretty easy to keep up with.
- **A large standard library** – Most of Python's libraries are easy to move around and work on UNIX, Windows, Mac.
- **Portable** – Python can run on a wide range of hardware platforms, and all of them have the same interface.

Python Example

Take a look at the following simple Python program –

```
a = int(input("Enter value for a"))
b = int(input("Enter value for b"))

print("The number you have entered for a is ", a)
print("The number you have entered for b is ", b)
```

In our example, we have taken two variables "a" and "b" and assigning some value to those variables. Note that in Python, we don't need to declare datatype for variables explicitly, as the PVM will assign datatype as per the user's input.

- The **input()** function is used to [take input from the user](#) through keyboard.
- In Python, the return type of **input()** is string only, so we have to convert it explicitly to the type of data which we require. In our example, we have converted to int type explicitly through **int()** function.
- **print()** is used to display the output.

Output

On execution, this Python code will produce the following output –

```
Enter value for a 10
Enter value for b 20

The number you have entered for a is 10
The number you have entered for b is 20
```

What is C++?

C++ is a statically typed, compiled, multi-paradigm, general-purpose programming language with a steep learning curve. Video games, desktop apps, and embedded systems use it extensively. C++ is so compatible with C that it can build practically all C source code without any changes. Object-oriented programming makes C++ a better-structured and safer language than C.

Features

Let's see some features of C++ and the reason of its popularity.

- **Middle-level language** – It's a middle-level language since it can be used for both system development and large-scale consumer applications like Media Players, Photoshop, Game Engines, etc.
- **Execution Speed** – C++ code runs quickly because it's compiled and uses procedures extensively. Garbage collection, dynamic typing, and other modern features impede program execution.
- **Object-oriented language** – Object-oriented programming is flexible and manageable. Large apps can also be developed through this language.
- **Extensive Library Support** – C++ has a vast library. Third-party libraries are supported for fast development.

C++ Example

Let's understand the syntax of C++ through an example written below.

```
#include
using namespace std;

int main() {
    int a, b;
    cout << "Enter The value for variable a \n";
    cin >> a;
    cout << "Enter The value for variable b";
    cin >> b;
    cout << "The value of a is " << a << "and" << b;
    return 0;
}
```

In our example, we are taking input for two variables "a" and "b" from the user through the keyboard and displaying the data on the console.

Output

On execution, it will produce the following output –

```
Enter The value for variable a
10
Enter The value for variable b
20
The value of a is 10 and 20
```

Comparison Between Python and C++ across Various Aspects

Both Python and C++ are among the most popular programming languages. Both of them have their advantages and disadvantages. In this tutorial, we shall take a look at their features which differentiate one from another.

Compiled vs Interpreted

Like C, C++ is also a compiler-based language. A compiler translates the entire code in a machine language code specific to the operating system in use and processor architecture.

Python is interpreter-based language. The interpreter executes the source code line by line.

Cross platform

When a C++ source code such as hello.cpp is compiled on Linux, it can be only run on any other computer with Linux operating system. If required to run on other OS, it needs to be recompiled.

Python interpreter doesn't produce compiled code. Source code is converted to byte code every time it is run on any operating system without any changes or additional steps.

Portability

Python code is easily portable from one OS to other. C++ code is not portable as it must be recompiled if the OS changes.

Speed of Development

C++ program is compiled to the machine code. Hence, its execution is faster than interpreter based language.

Python interpreter doesn't generate the machine code. Conversion of intermediate byte code to machine language is done on each execution of program.

If a program is to be used frequently, C++ is more efficient than Python.

Easy to Learn

Compared to C++, Python has a simpler syntax. Its code is more readable. Writing C++ code seems daunting in the beginning because of complicated syntax rules such as use of curly braces and semicolon for sentence termination.

Python doesn't use curly brackets for marking a block of statements. Instead, it uses indents. Statements of similar indent level mark a block. This makes a Python program more readable.

Static vs Dynamic Typing

C++ is a statically typed language. The type of variables for storing data need to be declared in the beginning. Undeclared variables can't be used. Once a variable is declared to be of a certain type, value of only that type can be stored in it.

Python is a dynamically typed language. It doesn't require a variable to be declared before assigning it a value. Since, a variable may store any type of data, it is called dynamically typed.

OOP Concepts

Both C++ and Python implement object oriented programming concepts. C++ is closer to the theory of OOP than Python. C++ supports the concept of data encapsulation as the visibility of the variables can be defined as public, private and protected.

Python doesn't have the provision of defining the visibility. Unlike C++, Python doesn't support method overloading. Because it is dynamically typed, all the methods are polymorphic in nature by default.

C++ is in fact an extension of C. One can say that additional keywords are added in C so that it supports OOP. Hence, we can write a C type procedure oriented program in C++.

Python is completely object oriented language. Python's data model is such that, even if you can adapt a procedure oriented approach, Python internally uses object-oriented methodology.

Garbage Collection

C++ uses the concept of pointers. Unused memory in a C++ program is not cleared automatically. In C++, the process of garbage collection is manual. Hence, a C++ program is likely to face memory related exceptional behavior.

Python has a mechanism of automatic garbage collection. Hence, Python program is more robust and less prone to memory related issues.

Application Areas

Because C++ program compiles directly to machine code, it is more suitable for system programming, writing device drivers, embedded systems and operating system utilities.

Python program is suitable for application programming. Its main area of application today is data science, machine learning, API development etc.

Difference Between Python and C++

The following table summarizes the differences between Python and C++ –

Criteria	Python	C++
Execution	Python is an interpreted-based programming language. Python programs are interpreted by an interpreter.	C++ is a compiler-based programming language. C++ programs are compiled by a compiler.
Typing	Python is a dynamic-typed language.	C++ is a static-typed language.
Portability	Python is a highly portable language, code written and executed on a system can be easily run on another system.	C++ is not a portable language, code written and executed on a system cannot be run on another system without making changes.
Garbage collection	Python provides a garbage collection feature. You do not need to worry about the memory management. It is automatic in Python.	C++ does not provide garbage collection. You have to take care of freeing memories. It is manual in C++.
Syntax	Python's syntaxes are very easy to read, write, and understand.	C++'s syntaxes are tedious.
Performance	Python's execution performance is slower than C++'s.	The speed of the execution of C++ codes is faster than Python codes.

Application areas	Python's application areas are machine learning, web applications, and more.	C++'s application areas are embedded systems, device drivers, and more.
-------------------	--	---

5. Python - Hello World Program

This tutorial will teach you how to write a simple Hello World program using Python Programming language. This program will make use of Python built-in print() function to print the output.

Hello World Program in Python

Printing "**Hello World**" is the first program in Python. This program will not take any user input. It will just print text on the output screen. It is used to test if the software needed to compile and run the program has been installed correctly.

Steps

The following are the steps to write a Python program to print Hello World –

- **Step 1:** Install Python. Make sure that Python is installed on your system. If Python is not installed, then install it from here:
<https://www.python.org/downloads/>
- **Step 2:** Choose Text Editor or IDE to write the code.
- **Step 3:** Open Text Editor or IDE, create a new file, and write the code to print Hello World.
- **Step 4:** Save the file with a file name and extension ".py".
- **Step 5:** Compile/Run the program.

Python Program to Print Hello World

```
# Python code to print "Hello World"  
print ("Hello World")
```

In the above code, we wrote two lines. The first line is the Python comment that will be ignored by the [Python interpreter](#), and the second line is the print() statement that will print the given message ("Hello World") on the output screen.

Output

```
Hello World
```

Different Ways to Write and Execute Hello World Program

Using Python Interpreter Command Prompt Mode

It is very easy to display the Hello World message using the Python interpreter. Launch the Python interpreter from a command terminal of your Windows Operating System and issue the print statement from the Python prompt as follows –

Example

```
PS C:\> python
```

```
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> print ("Hello World")
Hello World
```

Similarly, Hello World message is printed on Linux System.

Example

```
$ python3
Python 3.10.6 (main, Mar 10 2023, 10:55:28) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> print ("Hello World")
Hello World
```

Using Python Interpreter Script Mode

Python interpreter also works in scripted mode. Open any text editor, enter the following text and save as Hello.py

```
print ("Hello World")
```

For Windows OS, open the command prompt terminal (CMD) and run the program as shown below –

```
C:\>python hello.py
```

This will display the following output

```
Hello World
```

To run the program from Linux terminal

```
$ python3 hello.py
```

This will display the following output:

```
Hello World
```

Using Shebang #! in Linux Scripts

In Linux, you can convert a Python program into a self-executable script. The first statement in the code should be a shebang `#!`. It must contain the path to Python executable. In Linux, Python is installed in `/usr/bin` directory, and the name of the executable is `python3`. Hence, we add this statement to `hello.py` file

```
#!/usr/bin/python3
```

```
print ("Hello World")
```

You also need to give the file executable permission by using the chmod +x command

```
$ chmod +x hello.py
```

Then, you can run the program with following command line –

```
$ ./hello.py
```

This will display the following output

```
Hello World
```

FAQs

1. Why the first program is called Hello World?

It is just a simple program to test the basic syntax and compiler/interpreter configuration of [Python programming language](#).

2. Is installation of Python required to run Hello World program?

Yes. Python installation is required to run Hello World program.

3. How do I run a Python program without installing it?

TutorialsPoint developed an online environment where you can run your codes. You can use the [Python online compiler](#) to run your Python programs.

4. Which method is used to print Hello World or any message?

You can use the following methods –

- print() method
- sys.stdout.write() method by importing the sys module
- Using python-f string

6. Python - Application Areas

Python is a general-purpose programming language. It is suitable for the development of a wide range of software applications. Over the last few years Python has been the preferred language of choice for developers in the following application areas –

- Data Science
- Machine Learning
- Web Development
- Computer Vision and Image processing
- Embedded Systems and IoT
- Job Scheduling and Automation
- Desktop GUI Applications
- Console-based Applications
- CAD Applications
- Game Development

Let's look into these application areas in more detail:

Data Science

Python's recent meteoric rise in the popularity charts is largely due to its Data science libraries. Python has become an essential skill for data scientists. Today, real time web applications, mobile applications and other devices generate huge amount of data. Python's data science libraries help companies generate business insights from this data.

Libraries like [NumPy](#), [Pandas](#), and [Matplotlib](#) are extensively used to apply mathematical algorithms to the data and generate [visualizations](#). Commercial and community Python distributions like Anaconda and ActiveState bundle all the essential libraries required for data science.

Machine Learning

Python libraries such as Scikit-learn and TensorFlow help in building models for prediction of trends like customer satisfaction, projected values of stocks etc. based upon the past data. Machine learning applications include (but not restricted to) medical diagnosis, statistical arbitrage, basket analysis, sales prediction etc.

Web Development

Python's web frameworks facilitate rapid web application development. Django, Pyramid, Flask are very popular among the web developer community. etc. make it very easy to develop and deploy simple as well as complex web applications.

Latest versions of Python provide asynchronous programming support. Modern web frameworks leverage this feature to develop fast and high performance web apps and APIs.

Computer Vision and Image processing

OpenCV is a widely popular library for capturing and processing images. Image processing algorithms extract information from images, reconstruct image and video data. Computer Vision uses image processing for face detection and pattern recognition. OpenCV is a C++ library. Its Python port is extensively used because of its rapid development feature.

Some of the application areas of computer vision are robotics, industrial surveillance, automation, and biometrics etc.

Embedded Systems and IoT

Micropython (<https://micropython.org/>) is a lightweight version especially developed for microcontrollers like Arduino. Many automation products, robotics, IoT, and kiosk applications are built around Arduino and programmed with Micropython. Raspberry Pi is also very popular and a low cost single board computer can be used for these types of applications.

Job Scheduling and Automation

Python found one of its first applications in automating CRON (Command Run ON) jobs. Certain tasks, like periodic data backups, can be written in Python scripts and can be scheduled to be invoked automatically by operating system scheduler.

Many software products like Maya embed Python API for writing automation scripts (something similar to Excel macros).

Desktop GUI Applications

Python is a great option for building ergonomic, attractive, and user-friendly desktop GUI applications. Several graphics libraries, though built in C/C++, have been ported to Python. The popular Qt graphics toolkit is available as a PyQt package in Python. Similarly, wxWidgets has been ported to Python as WxPython. Python's built-in GUI package, Tkinter is a Python interface to the Tk Graphics toolkit.

Here is a select list of Python GUI libraries:

- **Tkinter** – Tkinter is the Python interface to the Tk GUI toolkit shipped with Python's standard library.
- **wxPython** – This is the Python interface for the wxWidgets GUI toolkit. BitTorrent Client application has been built with wxPython functionality.
- **PyQt** – Qt is one of the most popular GUI toolkits. It has been ported to Python as a PyQt5 package. Notable desktop GUI apps that use PyQt include QGIS, Spyder IDE, Calibre Ebook Manager, etc.
- **PyGTK** – PyGTK is a set of wrappers written in Python and C for GTK + GUI library. The complete PyGTK tutorial is available [here](#).
- **PySimpleGUI** – PySimpleGui is an open-source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide, and WxPython toolkits.
- **Jython** – Jython is a Python port for Java, which gives Python scripts seamless access to the Java GUI libraries on the local machine.

Console-based Applications

Python is often employed to build CLI (command-line interface) applications. Such scripts can be used to run scheduled CRON jobs such as taking database backups etc. There are many Python libraries that parse the command line arguments. The argparse library comes bundled with Python's standard library. You can use Click (part of Flask framework) and Typer (included in FastAPI framework) to build console interfaces to the web-based applications built by the respective frameworks. Textual is a rapid development framework to build apps that run inside a terminal as well as browsers.

CAD Applications

CAD engineers can take advantage of Python's versatility to automate repetitive tasks such as drawing shapes and generating reports.

Autodesk Fusion 360 is a popular CAD software, which has a Python API that allows users to automate tasks and create custom tools. Similarly, SolidWorks has a built-in Python shell that allows users to run Python scripts inside the software.

CATIA is another very popular CAD software. Along with a VBScript, certain third-party Python libraries can be used to control CATIA.

Game Development

Some popular gaming apps have been built with Python. Examples include BattleField2, The Sims 4, World of Tanks, Pirates of the Caribbean, and more. These apps are built with one of the following Python libraries.

Pygame is one of the most popular Python libraries used to build engaging computer games. Pygame is an open-source Python library for making multimedia applications like games built on top of the excellent SDL library. It is a cross-platform library, which means you can build a game that can run on any operating system.

Another library **Kivy** is also widely used to build desktop as well as mobile-based games. Kivy has a multi-touch interface. It is an open-source and cross-platform Python library for rapid development of game applications. Kivy runs on Linux, Windows, OS X, Android, iOS, and Raspberry Pi.

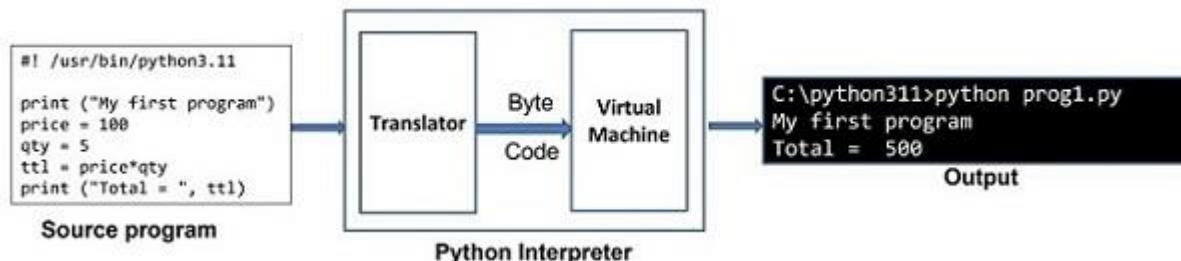
PyKyra library is based on both SDL (Software and Documentation Localisation) and the Kyra engine. It is one of the fastest game development frameworks. PyKyra supports MPEG , MP3, Ogg Vorbis, Wav, etc., multimedia formats.

7. Python Interpreter and Its Modes

Python Interpreter

Python is an interpreter-based language. In a Linux system, Python's executable is installed in /usr/bin/ directory. For Windows, the executable (python.exe) is found in the installation folder (for example C:\python311).

This tutorial will teach you **How Python Interpreter Works** in interactive and scripted mode. Python code is executed by one statement at a time method. Python interpreter has two components. The translator checks the statement for syntax. If found correct, it generates an intermediate byte code. There is a Python virtual machine which then converts the byte code in native binary and executes it. The following diagram illustrates the mechanism:



Python interpreter has an interactive mode and a scripted mode.

Python Interpreter - Interactive Mode

When launched from a command line terminal without any additional options, a Python prompt >>> appears and the Python interpreter works on the principle of **REPL (Read, Evaluate, Print, Loop)**. Each command entered in front of the Python prompt is read, translated and executed. A typical interactive session is as follows.

```
>>> price = 100
>>> qty = 5
>>> total = price*qty
>>> total
500
>>> print ("Total = ", total)
Total = 500
```

To close the interactive session, enter the end-of-line character (ctrl+D for Linux and ctrl+Z for Windows). You may also type **quit()** in front of the Python prompt and press Enter to return to the OS prompt.

```
>>> quit()
$
```

The interactive shell available with standard Python distribution is not equipped with features like line editing, history search, auto-completion etc. You can use other advanced interactive interpreter software such as **IPython** and **bpython** to have additional functionalities.

Python Interpreter - Scripting Mode

Instead of entering and obtaining the result of one instruction at a time as in the interactive environment, it is possible to save a set of instructions in a text file, make sure that it has .py extension, and use the name as the command line parameter for Python command.

Save the following lines as prog.py, with the use of any text editor such as vim on Linux or Notepad on Windows.

```
print ("My first program")
price = 100
qty = 5
total = price*qty
print ("Total = ", total)
```

When we execute above program on a Windows machine, it will produce following result:

```
C:\Users\Acer>python prog.py
My first program
Total = 500
```

Note that even though Python executes the entire script in one go, but internally it is still executed in line by line fashion.

In case of any compiler-based language such as Java, the source code is not converted in byte code unless the entire code is error-free. In Python, on the other hand, statements are executed until first occurrence of error is encountered.

Let us introduce an error purposefully in the above code.

```
print ("My first program")
price = 100
qty = 5
total = prive*qty #Error in this statement
print ("Total = ", total)
```

Note the misspelt variable **prive** instead of **price**. Try to execute the script again as before –

```
C:\Users\Acer>python prog.py
My first program
Traceback (most recent call last):
  File "C:\Python311\prog.py", line 4, in <module>
    total = prive*qty
```

```
^^^^^
```

```
NameError: name 'prive' is not defined. Did you mean: 'price'?
```

Note that the statements before the erroneous statement are executed and then the error message appears. Thus it is now clear that Python script is executed in interpreted manner.

Python Interpreter - Using Shebang #!

In addition to executing the Python script as above, the script itself can be a self-executable in Linux, like a shell script. You have to add a shebang line on top of the script. The shebang indicates which executable is used to interpret Python statements in the script. Very first line of the script starts with `#!` And followed by the path to Python executable.

Modify the `prog.py` script as follows –

```
#!/usr/bin/python3.11

print ("My first program")
price = 100
qty = 5
total = price*qty
print ("Total = ", total)
```

To mark the script as self-executable, use the `chmod` command

```
$ chmod +x prog.py
```

You can now execute the script directly, without using it as a command-line argument.

```
$ ./hello.py
```

Interactive Python - IPython

IPython (stands for Interactive Python) is an enhanced and powerful interactive environment for Python with many functionalities compared to the standard Python shell. IPython was originally developed by Fernando Perez in 2001.

IPython has the following important features –

- [IPython](#)'s object introspection ability to check properties of an object during runtime.
- Its syntax highlighting proves to be useful in identifying the language elements such as keywords, [variables](#) etc.
- The history of interactions is internally stored and can be reproduced.
- Tab completion of keywords, variables and function names is one of the most important features.
- IPython's Magic command system is useful for controlling [Python environment](#) and performing OS tasks.
- It is the main kernel for [Jupyter notebook](#) and other front-end tools of Project Jupyter.

Install IPython with PIP installer utility.

```
pip3 install ipython
```

Launch IPython from command-line

```
C:\Users\Acer>ipython
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type 'copyright', 'credits' or 'license' for more information
IPython 8.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Instead of the regular >>> prompt as in standard interpreter, you will notice two major IPython prompts as explained below –

- In[1] appears before any input expression.
- Out[1] appears before the Output appears.

```
In [1]: price = 100
In [2]: quantity = 5
In [3]: total = price*quantity
In [4]: total
Out[4]: 500
In [5]:
```

Tab completion is one of the most useful enhancements provided by IPython. IPython pops up appropriate list of methods as you press tab key after dot in front of object.

IPython provides information (introspection) of any object by putting ? in front of it. It includes **docstring**, function definitions and constructor details of class. For example, to explore the string object var defined above, in the input prompt enter var?.

```
In [5]: var = "Hello World"
In [6]: var?
Type: str
String form: Hello World
Length: 11
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str
Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
```

```
or repr(object).  
encoding defaults to sys.getdefaultencoding().  
errors defaults to 'strict'.
```

IPython's magic functions are extremely powerful. Line magics let you run DOS commands inside IPython. Let us run the dir command from within IPython console

```
In [8]: !dir *.exe  
Volume in drive F has no label.  
Volume Serial Number is E20D-C4B9  
  
Directory of F:\Python311  
  
07-02-2023 16:55      103,192 python.exe  
07-02-2023 16:55      101,656 pythonw.exe  
          2 File(s)   204,848 bytes  
          0 Dir(s)  105,260,306,432 bytes free
```

Jupyter notebook is a web-based interface to programming environments of Python, Julia, R and many others. For Python, it uses IPython as its main kernel.

8. Python - Environment Setup

First step in the journey of learning Python is to install it on your machine. Today most computer machines, especially having Linux OS, have Python pre-installed. However, it may not be the latest version.

Python is available on a wide variety of platforms including Linux and Mac OS X. The operating systems on which Python can be installed are listed below:

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion

Python has also been ported to the Java and .NET virtual machines. Let us know how to set up the Python environment.

Local Environment Setup

Open a terminal window and type "python" to find out if it is already installed along with its version. If Python is already installed, you will get a message given below:

```
$ python
Python 3.11.2 (main, Feb 8 2023, 14:49:24) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Downloading Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python

<https://www.python.org/>

You can download Python documentation from <https://www.python.org/doc/>. The documentation is available in HTML, PDF, and PostScript formats.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms –

Install Python on Ubuntu Linux

To check whether Python is already installed, open the Linux terminal and enter the following command –

```
$ python3.11 --version
```

In Ubuntu Linux, the easiest way to install Python is to use APT – Advanced Packaging Tool. It is always recommended to update the list of packages in all the configured repositories.

```
$ sudo apt update
```

Even after the update, the latest version of Python may not be available for install, depending upon the version of Ubuntu you are using. To overcome this, add the deadsnakes repository.

```
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

Update the package list again.

```
$ sudo apt update
```

To install the latest Python 3.11 version, enter the following command in the terminal –

```
$ sudo apt-get install python3.11
```

Check whether it has been properly installed.

```
$ python3
Python 3.11.2 (main, Feb 8 2023, 14:49:24) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> print ("Hello World")
Hello World

>>>
```

Install Python on other Linux

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to <https://www.python.org/downloads/>.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the Modules/Setup file if you want to customize some options.

Now issue the following commands:

```
$ run ./configure script
$ make
$ make install
```

This installs Python at standard location /usr/local/bin and its libraries at /usr/local/lib/pythonXX where XX is the version of Python.

Using Yum Command

Red Hat Enterprise Linux (RHEL 8) does not install Python 3 by default. We usually use yum command on CentOS and other related variants. The procedure for installing Python-3 on RHEL 8 is as follows:

```
$ sudo yum install python3
```

Install Python on Windows

It should be noted that Python's version 3.10 onwards cannot be installed on Windows 7 or earlier operating systems.

The recommended way to install Python is to use the official installer. A link to the latest stable version is given on the home page itself. It is also found at

<https://www.python.org/downloads/windows/>

You can find embeddable packages and installers for 32 as well as 64-bit architecture.

- Python 3.11.2 - Feb. 8, 2023

Note that Python 3.11.2 cannot be used on Windows 7 or earlier.

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (64-bit)
- Download Windows embeddable package (ARM64)
- Download Windows installer (32-bit)
- Download Windows installer (64-bit)
- Download Windows installer (ARM64)

Let us download 64-bit Windows installer –

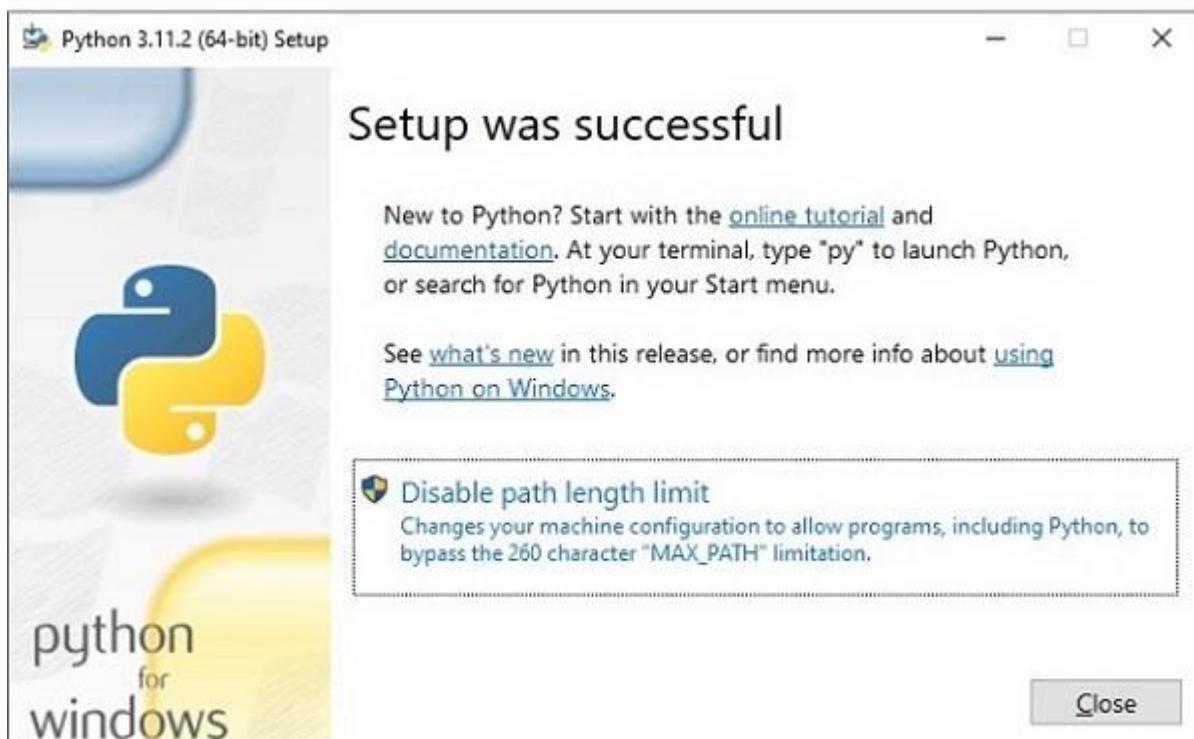
(<https://www.python.org/ftp/python/3.11.2/python-3.11.2-amd64.exe>)

Double click the file where it has been downloaded to start the installation.



Although you can straight away proceed by clicking the Install Now button, it is advised to choose the installation folder with a relatively shorter path, and tick the second check box to update the PATH variable.

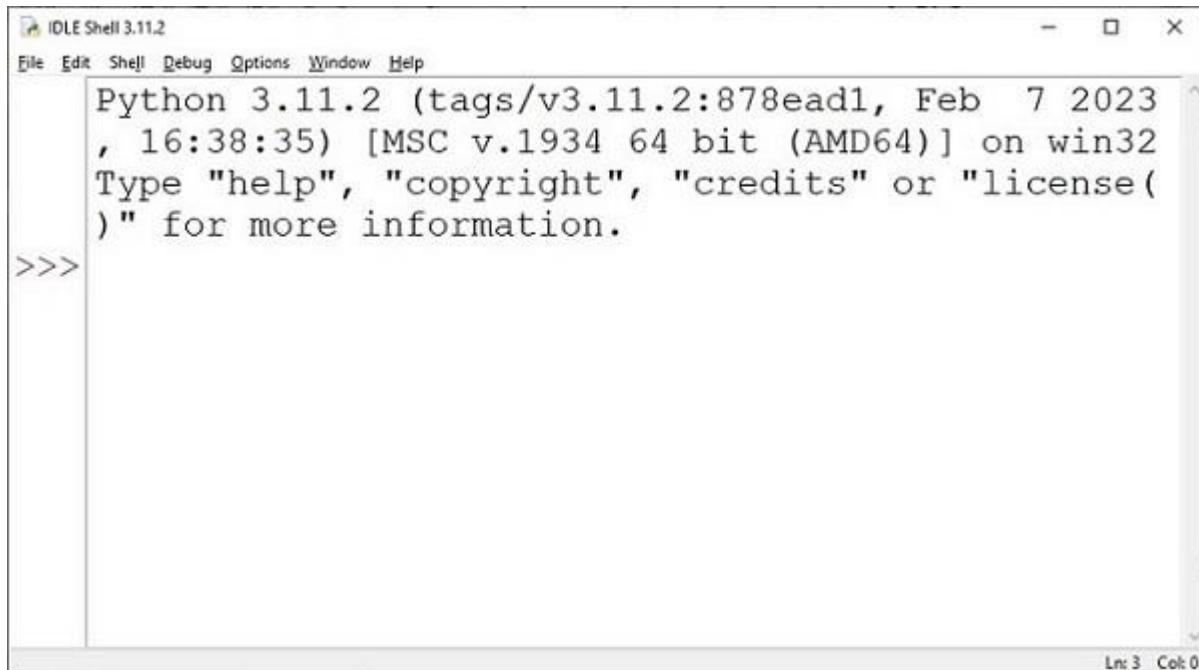
Accept defaults for rest of the steps in this installation wizard to complete the installation.



Open the Windows Command Prompt terminal and run Python to check the success of installation.

```
C:\Users\Acer>python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python's standard library has an executable module called IDLE – short for Integrated Development and Learning Environment. Find it from Window start menu and launch.



IDLE contains Python shell (interactive interpreter) and a customizable multi-window text editor with features such as syntax highlighting, smart indent, auto completion etc. It is cross-platform so works the same on Windows, MacOS and Linux. It also has a debugger with provision to set breakpoints, stepping, and viewing of global and local namespaces.

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to <https://www.python.org/downloads/>.
- Follow the link for the Windows installer python-XYZ.msi file where XYZ is the version you need to install.
- To use this installer python-XYZ.msi, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Macintosh Installation

Recent Macs come with Python installed, but it may be the old version. See <http://www.python.org/download/mac/> for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available.

Jack Jansen maintains it and you can have full access to the entire documentation at his website – <http://www.cwi.nl/~jack/macpython.html>. You can find complete installation details for Mac OS installation.

Setting up PATH

Programs and other executable files are available in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The path variable is named as PATH in Unix or Path in Windows (Unix is case sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Setting path at Unix/Linux

To add the Python directory to the path for a particular session in Unix –

- **In the csh shell** – type `setenv PATH "$PATH:/usr/local/bin/python"` and press Enter.
- **In the bash shell (Linux)** – type `export PATH="$PATH:/usr/local/bin/python"` and press Enter.
- **In the sh or ksh shell** – type `PATH="$PATH:/usr/local/bin/python"` and press Enter.

Note – `/usr/local/bin/python` is the path of the Python directory

Setting path at Windows

To add the Python directory to the path for a particular session in Windows –

At the command prompt – type `path %path%;C:\Python` and press Enter.

Note – `C:\Python` is the path of the Python directory

Python Environment Variables

Here are important environment variables, which can be recognized by Python

Sr.No.	Variable & Description
1	PYTHONPATH It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.
2	PYTHONSTARTUP

	It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.
3	PYTHONCASEOK It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
4	PYTHONHOME It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Running Python

There are three different ways to start Python –

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** in the command line.

Start coding right away in the interactive interpreter.

```
$python # Unix/Linux
or
python% # Unix/Linux
or
C:> python # Windows/DOS
```

Here is the list of all the available command line options –

Sr.No.	Option & Description
1	-d It provides debug output.
2	-O It generates optimized bytecode (resulting in .pyo files).
3	-S Do not run import site to look for Python paths on startup.

		-v
4		verbose output (detailed trace on import statements).
		-X
5		disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
		-c cmd
6		run Python script sent in as cmd string
		File
7		run Python script from given file

Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following –

```
$python script.py # Unix/Linux
or
python% script.py # Unix/Linux
or
C: >python script.py # Windows/DOS
```

Note – Be sure the file permission mode allows execution.

Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix** – IDLE is the very first Unix IDE for Python.
- **Windows** – PythonWin is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh** – The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

If you are not able to set up the environment properly, then you can take help from your system admin. Make sure the Python environment is properly set up and working perfectly.

We have provided [Python Online Compiler/Interpreter](#) which helps you to **Edit** and **Execute** the code directly from your browser. Try to click the icon  to run the following Python code to print conventional "Hello, World!".

Below code box allows you to change the value of the code. Try to change the value inside print() and run it again to verify the result.

```
# This is my first Python program.  
# This will print 'Hello, World!' as the output  
  
print ("Hello, World!");
```

9. Python - Virtual Environment

Python Virtual Environment

Python virtual environments create a virtual installation of Python inside a project directory. Users can then install and manage Python packages for each project. This allows users to be able to install packages and modify their Python environment without fear of breaking packages installed in other environments.

What is Virtual Environment in Python?

A Python virtual environment is:

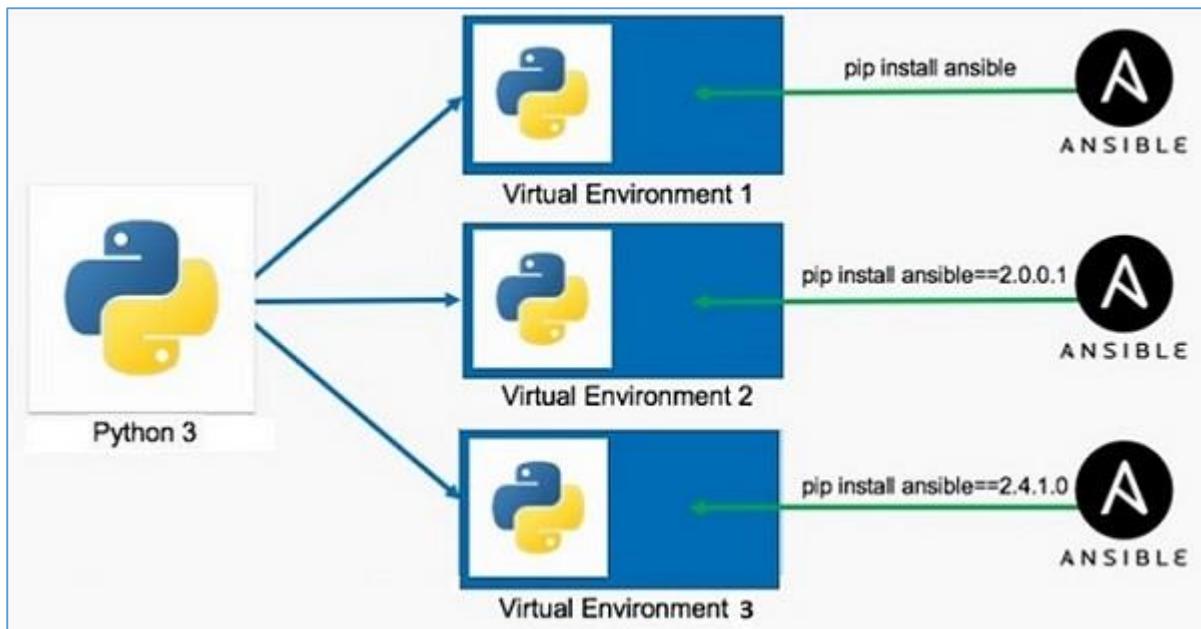
- Considered as disposable.
- Used to contain a specific Python interpreter and software libraries and binaries which are needed to support a project.
- Contained in a directory, conventionally either named venv or .venv in the project directory.
- Not considered as movable or copyable.

When you install Python software on your computer, it is available for use from anywhere in the file system. This is a system-wide installation.

While developing an application in Python, one or more libraries may be required to be installed using the pip utility (e.g., **pip3 install somelib**). Moreover, an application (let us say App1) may require a particular version of the library – say **somelib 1.0**. At the same time another Python application (for example App2) may require newer version of same library say **somelib 2.0**. Hence by installing a new version, the functionality of App1 may be compromised because of conflict between two different versions of same library.

This conflict can be avoided by providing two isolated environments of Python in the same machine. These are called virtual environment. A virtual environment is a separate directory structure containing isolated installation having a local copy of Python interpreter, standard library and other modules.

The following figure shows the purpose of advantage of using virtual environment. Using the global Python installation, more than one virtual environments are created, each having different version of the same library, so that conflict is avoided.



Creation of Virtual Environments in Python using venv

This functionality is supported by **venv** module in standard Python distribution. Use following commands to create a new virtual environment.

```
C:\Users\Acer>md\pythonapp
C:\Users\Acer>cd\pythonapp
C:\pythonapp>python -m venv myvenv
```

Here, `myvenv` is the folder in which a new Python virtual environment will be created showing following directory structure –

```
Directory of C:\pythonapp\myvenv
22-02-2023 09:53 <DIR> .
22-02-2023 09:53 <DIR> ..
22-02-2023 09:53 <DIR> Include
22-02-2023 09:53 <DIR> Lib
22-02-2023 09:53 77 pyvenv.cfg
22-02-2023 09:53 <DIR> Scripts
```

The utilities for activating and deactivating the virtual environment as well as the local copy of Python interpreter will be placed in the `Scripts` folder.

```
Directory of C:\pythonapp\myvenv\scripts
22-02-2023 09:53 <DIR> .
22-02-2023 09:53 <DIR> ..
22-02-2023 09:53 2,063 activate
22-02-2023 09:53 992 activate.bat
22-02-2023 09:53 19,611 Activate.ps1
```

```
22-02-2023 09:53 393 deactivate.bat
22-02-2023 09:53 106,349 pip.exe
22-02-2023 09:53 106,349 pip3.10.exe
22-02-2023 09:53 106,349 pip3.exe
22-02-2023 09:53 242,408 python.exe
22-02-2023 09:53 232,688 pythonw.exe
```

Activating Virtual Environment

To enable this new virtual environment, execute **activate.bat** in Scripts folder.

```
C:\pythonapp>myvenv\scripts\activate
(myvenv) C:\pythonapp>
```

Note the name of the virtual environment in the parentheses. The Scripts folder contains a local copy of Python interpreter. You can start a Python session in this virtual environment.

Checking If Python is Running Inside a Virtual Environment

To confirm whether this Python session is in virtual environment check the sys.path.

```
(myvenv) C:\pythonapp>python
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec 6 2021, 19:10:37) [MSC v.1929
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', 'C:\\Python310\\python310.zip', 'C:\\Python310\\DLLs',
'C:\\Python310\\lib', 'C:\\Python310', 'C:\\pythonapp\\myvenv',
'C:\\pythonapp\\myvenv\\lib\\site-packages']
>>>
```

The scripts folder of this virtual environment also contains pip utilities. If you install a package from PyPI, that package will be active only in current virtual environment.

Deactivating Virtual Environment

To deactivate this environment, run deactivate.bat.

10. Python - Syntax

Python - Syntax

The Python syntax defines a set of rules that are used to create a Python Program. The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages. However, there are some definite differences between the languages.

First Python Program

Let us execute a Python program to print "Hello, World!" in two different modes of Python Programming. (a) Interactive Mode Programming (b) Script Mode Programming.

Python - Interactive Mode Programming

We can invoke a Python interpreter from command line by typing python at the command prompt as following –

```
$ python3
Python 3.10.6 (main, Mar 10 2023, 10:55:28) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Here >>> denotes a Python Command Prompt where you can type your commands. Let's type the following text at the Python prompt and press the Enter –

```
>>> print ("Hello, World!")
```

If you are running older version of Python, like Python 2.4.x, then you would need to use print statement without parenthesis as in print "Hello, World!". However, in Python version 3.x, this produces the following result –

```
Hello, World!
```

Python - Script Mode Programming

We can invoke the Python interpreter with a script parameter which begins the execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script which is simple text file. Python files have extension **.py**. Type the following source code in a **test.py** file –

```
print ("Hello, World!")
```

We assume that you have Python interpreter path set in the PATH variable. Now, let's try to run this program as follows –

```
$ python3 test.py
```

This produces the following result –

```
Hello, World!
```

Let us try another way to execute a Python script. Here is the modified test.py file –

```
#!/usr/bin/python3
print ("Hello, World!")
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows –

```
$ chmod +x test.py      # This is to make file executable
$ ./test.py
```

This produces the following result –

```
Hello, World!
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, \$, and % within identifiers.

Python is a case sensitive programming Language. Thus, Manpower and manpower are two different identifiers in Python.

Here are naming conventions for Python identifiers –

- Python Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is **private** identifier.
- Starting an identifier with two leading underscores indicates a strongly **private** identifier.
- If the identifier also ends with two trailing underscores, the identifier is a **language-defined** special name.

Python Reserved Words

The following list shows the Python keywords or reserved words. These are reserved words which cannot be used as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	as	assert
break	class	continue
def	del	elif
else	except	FALSE
finally	for	from
global	if	import

in	is	lambda
None	nonlocal	not
or	pass	raise
return	TRUE	try
while	with	yield

Python Lines and Indentation

Python programming provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if True:
    print ("True")
else:
    print ("False")
```

However, the following block generates an error –

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
    print ("False")
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks –

Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

```
import sys
try:
    # open file stream
    file = open(file_name, "w")
except IOError:
    print "There was an error writing to", file_name
    sys.exit()
print "Enter '', file_finish,
print '' When finished"
while file_text != file_finish:
```

```

file_text = raw_input("Enter text: ")
if file_text == file_finish:
    # close the file
    file.close()
    break
file.write(file_text)
file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
    print "Next time please enter something"
    sys.exit()
try:
    file = open(file_name, "r")
except IOError:
    print "There was an error reading file"
    sys.exit()
file_text = file.read()
file.close()
print file_text

```

Python Multi-Line Statements

Statements in Python typically end with a new line. Python however, allows the use of the line continuation character (\) to denote that the line should continue. For example –

```

total = item_one + \
        item_two + \
        item_three

```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example, following statement works well in Python –

```

days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']

```

Quotations in Python

Python accepts single ('), double ("") and triple ("'" or "'''") quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal –

```
word = 'word'
print (word)

sentence = "This is a sentence."
print (sentence)

paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
print (paragraph)
```

Comments in Python

A comment is a programmer-readable explanation or annotation in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter

Just like most modern languages, Python supports single-line (or end-of-line) and multi-line (block) comments. Python comments are very much similar to the comments available in PHP, BASH and Perl Programming languages.

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
# First comment
print ("Hello, World!") # Second comment
```

This produces the following result –

```
Hello, World!
```

You can type a comment on the same line after a statement or expression –

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows –

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comment:

```
...
This is a multiline
```

```
comment.  
...  
...
```

Using Blank Lines in Python Programs

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt, the statement saying "Press the enter key to exit", and waits for the user to take action –

```
#!/usr/bin/python  
  
raw_input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon –

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example –

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

Command Line Arguments in Python

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h –

```
$ python3 -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d      : debug output from parser (also PYTHONDEBUG=x)
-E      : ignore environment variables (such as PYTHONPATH)
-h      : print this help message and exit

[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

11. Python - Variables

Python Variables

Python variables are the reserved memory locations used to store values within a Python Program. This means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.

Memory Addresses

Data items belonging to different data types are stored in computer's memory. Computer's memory locations are having a number or address, internally represented in binary form. Data is also stored in binary form as the computer works on the principle of binary representation. In the following diagram, a string May and a number 18 is shown as stored in memory locations.

Memory

100	101	102	103	104
			May	
200	201	202	203	204
300	301	302	303	304
		18		
400	401	402	403	404
500	501	502	503	504

If you know the assembly language, you can convert these data items and the memory address, and give a machine language instruction. However, it is not easy for everybody. Language translator such as Python interpreter performs this type of conversion. It stores the object in a randomly chosen memory location. Python's built-in `id()` function returns the address where the object is stored.

```
>>> "May"  
May  
>>> id("May")  
2167264641264
```

```
>>> 18
18
>>> id(18)
140714055169352
```

Once the data is stored in the memory, it can be accessed repeatedly for performing a certain process. Obviously, fetching the data from its ID is cumbersome. High level languages like Python make it possible to give a suitable alias or a label to refer to the memory location.

In the above example, let us label the location of May as month, and location in which 18 is stored as age. Python uses the assignment operator (=) to bind an object with the label.

```
>>> month="May"
>>> age=18
```

The data object (May) and its name (month) have the same id(). The id() of 18 and age are also same.

```
>>> id(month)
2167264641264
>>> id(age)
140714055169352
```

The label is an identifier. It is usually called as a variable. A Python variable is a symbolic name that is a reference or pointer to an object.

Creating Python Variables

Python variables do not need explicit declaration to reserve memory space or you can say to create a variable. A Python variable is created automatically when you assign a value to it. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example –

Example to Create Python Variables

This example creates different types (an integer, a float, and a string) of variables.

```
counter = 100          # Creates an integer variable
miles    = 1000.0       # Creates a floating point variable
name     = "Zara Ali"   # Creates a string variable
```

Printing Python Variables

Once we create a Python variable and assign a value to it, we can print it using print() function. Following is the extension of previous example and shows how to print different variables in Python:

Example to Print Python Variables

This example prints variables.

```
counter = 100          # Creates an integer variable
miles    = 1000.0       # Creates a floating point variable
name     = "Zara Ali"   # Creates a string variable

print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "Zara Ali" are the values assigned to counter, miles, and name variables, respectively. When running the above Python program, this produces the following result –

```
100
1000.0
Zara Ali
```

Deleting Python Variables

You can delete the reference to a number object by using the del statement. The syntax of the del statement is –

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the del statement. For example –

```
del var
del var_a, var_b
```

Example

Following example shows how we can delete a variable and if we try to use a deleted variable then Python interpreter will throw an error:

```
counter = 100
print (counter)

del counter
print (counter)
```

This will produce the following result:

```
100
Traceback (most recent call last):
```

```
File "main.py", line 7, in <module>
    print (counter)
NameError: name 'counter' is not defined
```

Getting Type of a Variable

You can get the data type of a Python variable using the python built-in function type() as follows.

Example: Printing Variables Type

```
x = "Zara"
y = 10
z = 10.10
print(type(x))
print(type(y))
print(type(z))
```

This will produce the following result:

```
<class 'str'>
<class 'int'>
<class 'float'>
```

Casting Python Variables

You can specify the data type of a variable with the help of casting as follows:

Example

This example demonstrates case sensitivity of variables.

```
x = str(10)      # x will be '10'
y = int(10)      # y will be 10
z = float(10)    # z will be 10.0
print( "x =", x )
print( "y =", y )
print( "z =", z )
```

This will produce the following result:

```
x = 10
y = 10
z = 10.0
```

Case-Sensitivity of Python Variables

Python variables are case sensitive which means Age and age are two different variables:

```
age = 20
Age = 30

print( "age =", age )
print( "Age =", Age )
```

This will produce the following result:

```
age = 20
Age = 30
```

Python Variables - Multiple Assignment

Python allows to initialize more than one variables in a single statement. In the following case, three variables have same value.

```
>>> a=10
>>> b=10
>>> c=10
```

Instead of separate assignments, you can do it in a single assignment statement as follows –

```
>>> a=b=c=10
>>> print (a,b,c)
10 10 10
```

In the following case, we have three variables with different values.

```
>>> a=10
>>> b=20
>>> c=30
```

These separate assignment statements can be combined in one. You need to give comma separated variable names on left, and comma separated values on the right of = operator.

```
>>> a,b,c = 10,20,30
>>> print (a,b,c)
10 20 30
```

Let's try few examples in script mode: –

```
a = b = c = 100
```

```
print (a)
print (b)
print (c)
```

This produces the following result:

```
100
100
100
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –

```
a,b,c = 1,2,"Zara Ali"

print (a)
print (b)
print (c)
```

This produces the following result:

```
1
2
Zara Ali
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Zara Ali" is assigned to the variable c.

Python Variables - Naming Convention

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number or any special character like \$, (, * % etc.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
- Python reserved keywords cannot be used naming the variable.

If the name of variable contains multiple words, we should use these naming patterns –

- **Camel case** – First letter is a lowercase, but first letter of each subsequent word is in uppercase. For example: kmPerHour, pricePerLitre
- **Pascal case** – First letter of each word is in uppercase. For example: KmPerHour, PricePerLitre

- **Snake case** – Use single underscore (_) character to separate words. For example: km_per_hour, price_per_litre

Example

Following are valid Python variable names:

```
counter = 100
_count = 100
name1 = "Zara"
name2 = "Nuha"
Age = 20
zara_salary = 100000

print(counter)
print(_count)
print(name1)
print(name2)
print(Age)
print(zara_salary)
```

This will produce the following result:

```
100
100
Zara
Nuha
20
100000
```

Example

Following are invalid Python variable names:

```
1counter = 100
$_count = 100
zara-salary = 100000
print(1counter)
print($count)
print(zara-salary)
```

This will produce the following result:

```
File "main.py", line 3
```

```
1counter = 100
^
SyntaxError: invalid syntax
```

Example

Once you use a variable to identify a data object, it can be used repeatedly without its id() value. Here, we have a variables height and width of a rectangle. We can compute the area and perimeter with these variables.

```
>>> width=10
>>> height=20
>>> area=width*height
>>> area
200
>>> perimeter=2*(width+height)
>>> perimeter
60
```

Use of variables is especially advantageous when writing scripts or programs. Following script also uses the above variables.

```
#!/usr/bin/python3
width = 10
height = 20
area = width*height
perimeter = 2*(width+height)
print ("Area = ", area)
print ("Perimeter = ", perimeter)
```

Save the above script with .py extension and execute from command-line. The result would be –

```
Area = 200
Perimeter = 60
```

Python Local Variables

Python Local Variables are defined inside a function. We cannot access variable outside the function.

A Python function is a piece of reusable code and you will learn more about function in Python - Functions tutorial.

Example

Following is an example to show the usage of local variables:

```
def sum(x,y):
    sum = x + y
    return sum
print(sum(5, 10))
```

This will produce the following result –

15

Python Global Variables

Any variable created outside a function can be accessed within any function and so they have global scope.

Example

Following is an example of global variables –

```
x = 5
y = 10
def sum():
    sum = x + y
    return sum
print(sum())
```

This will produce the following result –

15

Constants in Python

Python doesn't have any formally defined constants, However, you can indicate a variable to be treated as a constant by using all-caps names with underscores. For example, the name PI_VALUE indicates that you don't want the variable redefined or changed in any way.

The naming convention using all-caps is sometimes referred to as screaming snake case - where the all-caps refer to screaming and the underscores refer to snakes.

Python vs C/C++ Variables

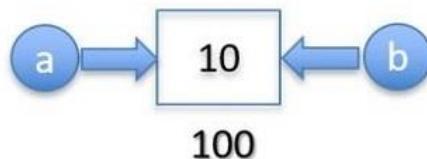
The concept of variable works differently in Python than in C/C++. In C/C++, a variable is a named memory location. If `a=10` and also `b=10`, both are two different memory locations. Let us assume their memory address is 100 and 200 respectively.



If a different value is assigned to "a" - say 50, 10 in the address 100 is overwritten.



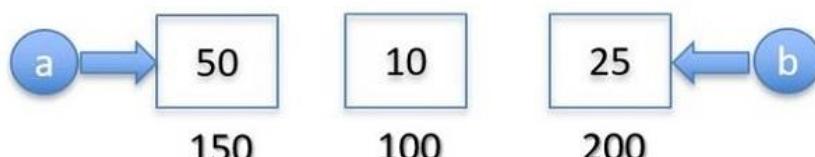
A Python variable refers to the object and not the memory location. An object is stored in memory only once. Multiple variables are really the multiple labels to the same object.



The statement `a=50` creates a new int object 50 in the memory at some other location, leaving the object 10 referred by "b".



Further, if you assign some other value to b, the object 10 is not referred.



Python's garbage collector mechanism releases the memory occupied by any object that is not referred.

Python's identity operator returns True if both the operands have same `id()` value.

```
>>> a=b=10
>>> a is b
True
>>> id(a), id(b)
(140731955278920, 140731955278920)
```

12. Python - Data Types

Python Data Types

Python data types are actually classes, and the defined variables are their instances or objects. Since Python is dynamically typed, the data type of a variable is determined at runtime based on the assigned value.

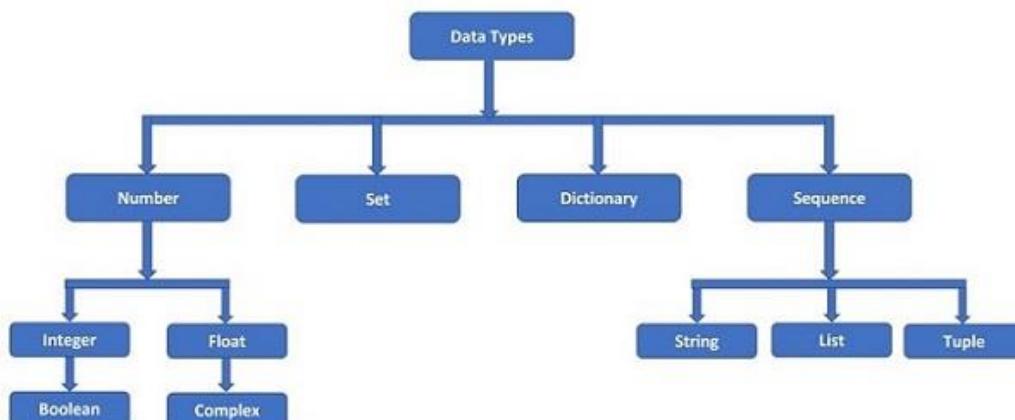
In general, the data types are used to define the type of a variable. It represents the type of data we are going to store in a variable and determines what operations can be done on it.

Each programming language has its own classification of data items. With these datatypes, we can store different types of data values.

Types of Data Types in Python

Python supports the following built-in data types –

- [Numeric Data Types](#)
 - int
 - float
 - complex
- [String Data Types](#)
- [Sequence Data Types](#)
 - list
 - tuple
 - range
- [Binary Data Types](#)
 - bytes
 - bytearray
 - memoryview
- [Dictionary Data Type](#)
- [Set Data Type](#)
 - set
 - frozenset
- [Boolean Data Type](#)
- [None Type](#)



1. Python Numeric Data Types

Python numeric data types store numeric values. Number objects are created when you assign a value to them. For example –

```
var1 = 1      # int data type
var2 = True    # bool data type
var3 = 10.023  # float data type
var4 = 10+3j   # complex data type
```

Python supports four different numerical types and each of them have built-in classes in Python library, called **int**, **bool**, **float** and **complex** respectively –

- **int** (signed integers)
- **float** (floating point real values)
- **complex** (complex numbers)

A complex number is made up of two parts - real and imaginary. They are separated by '+' or '-' signs. The imaginary part is suffixed by 'j' which is the imaginary number. The square root of -1 ($\sqrt{-1}$), is defined as imaginary number. Complex number in Python is represented as $x+yj$, where x is the real part, and y is the imaginary part. So, $5+6j$ is a complex number.

```
>>> type(5+6j)
<class 'complex'>
```

Here are some examples of numbers –

int	float	complex
10	0	3.14j
00777	15.2	45.j
-786	-21.9	9.322e-36j
80	32.3+e18	.876j
0x17	-90	-.6545+0J
-0x260	-3.25E+101	3e+26J
0x69	70.2-E12	4.53e-7j

Example of Numeric Data Types

Following is an example to show the usage of Integer, Float and Complex numbers:

```
# integer variable.
a=100
print("The type of variable having value", a, " is ", type(a))

# float variable.
c=20.345
print("The type of variable having value", c, " is ", type(c))
```

```
# complex variable.
d=10+3j
print("The type of variable having value", d, " is ", type(d))
```

2. Python String Data Type

Python string is a sequence of one or more Unicode characters, enclosed in single, double or triple quotation marks (also called inverted commas). Python strings are immutable which means when you perform an operation on strings, you always produce a new string object of the same type, rather than mutating an existing string.

As long as the same sequence of characters is enclosed, single or double or triple quotes don't matter. Hence, following string representations are equivalent.

```
>>> 'TutorialsPoint'
'TutorialsPoint'
>>> "TutorialsPoint"
'TutorialsPoint'
>>> '''TutorialsPoint'''
'TutorialsPoint'
```

A string in Python is an object of str class. It can be verified with type() function.

```
>>> type("Welcome To TutorialsPoint")
<class 'str'>
```

A string is a non-numeric data type. Obviously, we cannot perform arithmetic operations on it. However, operations such as slicing and concatenation can be done. Python's str class defines a number of useful methods for string processing. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator in Python.

Example of String Data Type

```
str = 'Hello World!'
print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])       # Prints string starting from 3rd character
print (str * 2)       # Prints string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result –

```
Hello World!
```

```
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

3. Python Sequence Data Types

Sequence is a collection data type. It is an ordered collection of items. Items in the sequence have a positional index starting with 0. It is conceptually similar to an array in C or C++. There are following three sequence data types defined in Python.

- List Data Type
- Tuple Data Type
- Range Data Type

Python sequences are bounded and iterable - Whenever we say an iterable in Python, it means a sequence data type (for example, a list).

(a) Python List Data Type

Python Lists are the most versatile compound data types. A Python list contains items separated by commas and enclosed within square brackets ([]). To some extent, Python lists are similar to arrays in C. One difference between them is that all the items belonging to a Python list can be of different data type whereas C array can store elements related to a particular data type.

```
>>> [2023, "Python", 3.11, 5+6j, 1.23E-4]
```

A list in Python is an object of list class. We can check it with type() function.

```
>>> type([2023, "Python", 3.11, 5+6j, 1.23E-4])
<class 'list'>
```

As mentioned, an item in the list may be of any data type. It means that a list object can also be an item in another list. In that case, it becomes a nested list.

```
>>> [[ 'One', 'Two', 'Three'], [1,2,3], [1.0, 2.0, 3.0]]
```

A list can have items which are simple numbers, strings, tuple, dictionary, set or object of user defined class.

The values stored in a Python list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Example of List Data Type

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print (list)          # Prints complete list
```

```

print (list[0])          # Prints first element of the list
print (list[1:3])        # Prints elements starting from 2nd till 3rd
print (list[2:])         # Prints elements starting from 3rd element
print (tinylist * 2)     # Prints list two times
print (list + tinylist)  # Prints concatenated lists

```

This produce the following result –

```

['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

```

(b) Python Tuple Data Type

Python tuple is another sequence data type that is similar to a list. A Python tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses (...).

A tuple is also a sequence, hence each item in the tuple has an index referring to its position in the collection. The index starts from 0.

```
>>> (2023, "Python", 3.11, 5+6j, 1.23E-4)
```

In Python, a tuple is an object of tuple class. We can check it with the type() function.

```
>>> type((2023, "Python", 3.11, 5+6j, 1.23E-4))
<class 'tuple'>
```

As in case of a list, an item in the tuple may also be a list, a tuple itself or an object of any other Python class.

```
>>> ([ 'One', 'Two', 'Three'], 1,2.0,3, (1.0, 2.0, 3.0))
```

To form a tuple, use of parentheses is optional. Data items separated by comma without any enclosing symbols are treated as a tuple by default.

```
>>> 2023, "Python", 3.11, 5+6j, 1.23E-4
(2023, 'Python', 3.11, (5+6j), 0.000123)
```

Example of Tuple Data Type

```

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print (tuple)          # Prints the complete tuple
print (tuple[0])       # Prints first element of the tuple

```

```

print (tuple[1:3])           # Prints elements of the tuple starting from 2nd
till 3rd

print (tuple[2:])            # Prints elements of the tuple starting from 3rd
element

print (tinytuple * 2)         # Prints the contents of the tuple twice

print (tuple + tinytuple)     # Prints concatenated tuples

```

The code produces the following result –

```

('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')

```

The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed i.e. lists are mutable, while tuples are enclosed in parentheses (()) and cannot be updated (immutable). Tuples can be thought of as read-only lists.

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000    # Valid syntax with list

```

(c) Python Range Data Type

A Python range is an immutable sequence of numbers which is typically used to iterate through a specific number of items.

It is represented by the Range class. The constructor of this class accepts a sequence of numbers starting from 0 and increments to 1 until it reaches a specified number. Following is the syntax of the function –

```
range(start, stop, step)
```

Here is the description of the parameters used –

- **start:** Integer number to specify starting position, (Its optional, Default: 0)
- **stop:** Integer number to specify ending position (It's mandatory)
- **step:** Integer number to specify increment, (Its optional, Default: 1)

Example of Range Data Type

Following is a program which uses for loop to print numbers from 0 to 4 –

```

for i in range(5):
    print(i)

```

The code produces the following result –

```
0
1
2
3
4
```

Now let's modify above program to print the number starting from 2 instead of 0 –

```
for i in range(2, 5):
    print(i)
```

This code produces the following result –

```
2
3
4
```

Again, let's modify the program to print the number starting from 1 but with an increment of 2 instead of 1:

```
for i in range(1, 5, 2):
    print(i)
```

This code produces the following result –

```
1
3
```

4. Python Binary Data Types

A binary data type in Python is a way to represent data as a series of binary digits, which are 0's and 1's. It is like a special language computers understand to store and process information efficiently.

This type of data is commonly used when dealing with things like files, images, or anything that can be represented using just two possible values. So, instead of using regular numbers or letters, binary sequence data types use a combination of 0s and 1s to represent information.

Python provides three different ways to represent binary data. They are as follows –

- bytes
- bytearray
- memoryview

Let us discuss each of these data types individually –

(a) Python Bytes Data Type

The byte data type in Python represents a sequence of bytes. Each byte is an integer value between 0 and 255. It is commonly used to store binary data, such as images, files, or network packets.

We can create bytes in Python using the built-in bytes() function or by prefixing a sequence of numbers with b.

Example of Bytes Data Type

In the following example, we are using the built-in bytes() function to explicitly specify a sequence of numbers representing ASCII values –

- # Using bytes() function to create bytes
- b1 = bytes([65, 66, 67, 68, 69])
- print(b1)

The result obtained is as follows –

```
b'ABCDE'
```

Here we are using the "b" prefix before a string to automatically create a bytes object –

```
# Using prefix 'b' to create bytes
b2 = b'Hello'
print(b2)
```

Following is the output of the above code –

```
b'Hello'
```

(b) Python Bytearray Data Type

The bytearray data type in Python is quite similar to the bytes data type, but with one key difference: it is mutable, meaning you can modify the values stored in it after it is created.

You can create a bytearray using various methods by passing an iterable of integers representing byte values, by encoding a string, or by converting an existing bytes or bytearray object. For this, we use bytearray() function.

Example of Bytearray Data Type

In the example below, we are creating a bytearray by passing an iterable of integers representing byte values –

```
# Creating a bytearray from an iterable of integers
value = bytearray([72, 101, 108, 108, 111])
print(value)
```

The output obtained is as shown below –

```
bytearray(b'Hello')
```

Now, we are creating a bytearray by encoding a string using a "UTF-8" encoding –

```
# Creating a bytearray by encoding a string
val = bytearray("Hello", 'utf-8')
```

```
print(val)
```

The result produced is as follows –

```
bytearray(b'Hello')
```

(c) Python Memoryview Data Type

In Python, a memoryview is a built-in object that provides a view into the memory of the original object, generally objects that support the buffer protocol, such as byte arrays (bytearray) and bytes (bytes). It allows you to access the underlying data of the original object without copying it, providing efficient memory access for large datasets.

You can create a memoryview using various methods. These methods include using the memoryview() constructor, slicing bytes or bytearray objects, extracting from array objects, or using built-in functions like open() when reading from files.

Example of Memoryview Data Type

In the given example, we are creating a memoryview object directly by passing a supported object to the memoryview() constructor. The supported objects generally include byte arrays (bytearray), bytes (bytes), and other objects that support the buffer protocol –

```
data = bytearray(b'Hello, world!')
view = memoryview(data)
print(view)
```

Following is the output of the above code –

```
<memory at 0x00000186FFAA3580>
```

If you have an array object, you can create a memoryview using the buffer interface as shown below –

```
import array
arr = array.array('i', [1, 2, 3, 4, 5])
view = memoryview(arr)
print(view)
```

The output obtained is as shown below –

```
<memory at 0x0000017963CD3580>
```

You can also create a memoryview by slicing a bytes or bytearray object –

```
data = b'Hello, world!'
# Creating a view of the last part of the data
view = memoryview(data[7:])
print(view)
```

The result obtained is as follows –

```
<memory at 0x00000200D9AA3580>
```

5. Python Dictionary Data Type

Python dictionaries are kind of hash table type. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Python dictionary is like associative arrays or hashes found in Perl and consist of key:value pairs. The pairs are separated by comma and put inside curly brackets {}. To establish mapping between key and value, the colon':' symbol is put between the two.

```
>>> {1:'one', 2:'two', 3:'three'}
```

In Python, dictionary is an object of the built-in dict class. We can check it with the type() function.

```
>>> type({1:'one', 2:'two', 3:'three'})  
<class 'dict'>
```

Dictionaries are enclosed by curly braces ({}) and values can be assigned and accessed using square braces ([]).

Example of Dictionary Data Type

```
dict = {}  
dict['one'] = "This is one"  
dict[2] = "This is two"  
tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}  
print (dict['one'])      # Prints value for 'one' key  
print (dict[2])        # Prints value for 2 key  
print (tinydict)       # Prints complete dictionary  
print (tinydict.keys()) # Prints all the keys  
print (tinydict.values()) # Prints all the values
```

This code produces the following result –

```
This is one  
This is two  
{'dept': 'sales', 'code': 6734, 'name': 'john'}  
['dept', 'code', 'name']  
['sales', 6734, 'john']
```

Python's dictionary is not a sequence. It is a collection of items but each item (key:value pair) is not identified by positional index as in string, list or tuple. Hence, slicing operation cannot be done on a dictionary. Dictionary is a mutable object, so it is possible to perform add, modify or delete actions with corresponding functionality defined in dict class. These operations will be explained in a subsequent chapter.

6. Python Set Data Type

Set is a Python implementation of set as defined in Mathematics. A set in Python is a collection, but is not an indexed or ordered collection as string, list or tuple. An object cannot appear more than once in a set, whereas in List and Tuple, same object can appear more than once.

Comma separated items in a set are put inside curly brackets or braces {}. Items in the set collection can be of different data types.

```
>>> {2023, "Python", 3.11, 5+6j, 1.23E-4}
{(5+6j), 3.11, 0.000123, 'Python', 2023}
```

Note that items in the set collection may not follow the same order in which they are entered. The position of items is optimized by Python to perform operations over set as defined in mathematics.

Python's Set is an object of built-in **set** class, as can be checked with the type() function.

```
>>> type({2023, "Python", 3.11, 5+6j, 1.23E-4})
<class 'set'>
```

A set can store only **immutable objects** such as number (int, float, complex or bool), string or tuple. If you try to put a list or a dictionary in the set collection, Python raises a **TypeError**.

```
>>> {[ 'One', 'Two', 'Three'], 1,2,3, (1.0, 2.0, 3.0)}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Hashing is a mechanism in computer science which enables quicker searching of objects in computer's memory. **Only immutable objects are hashable**.

Even if a set doesn't allow mutable items, the set itself is mutable. Hence, add/delete/update operations are permitted on a set object, using the methods in built-in set class. Python also has a set of operators to perform set manipulation. The methods and operators are explained in latter chapters

Example of Set

```
set1 = {123, 452, 5, 6}
set2 = {'Java', 'Python', 'JavaScript'}
print(set1)
print(set2)
```

This will generate the following output –

```
{123, 452, 5, 6}
{'Python', 'JavaScript', 'Java'}
```

7. Python Boolean Data Type

Python Boolean type is one of built-in data types which represents one of the two values either True or False. Python `bool()` function allows you to evaluate the value of any expression and returns either True or False based on the expression.

A Boolean number has only two possible values, as represented by the keywords, True and False. They correspond to integer 1 and 0 respectively.

```
>>> type (True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Example of Boolean Data Type

Following is a program which prints the value of boolean variables a and b –

```
a = True
# display the value of a
print(a)
# display the data type of a
print(type(a))
```

This code will produce the following result –

```
true
<class 'bool'>
```

Following is another program which evaluates the expressions and prints the return values –

```
# Returns false as a is not equal to b
a = 2
b = 4
print(bool(a==b))

# Following also prints the same
print(a==b)

# Returns False as a is None
a = None
print(bool(a))

# Returns false as a is an empty sequence
```

```
a = ()
print(bool(a))

# Returns false as a is 0
a = 0.0
print(bool(a))

# Returns false as a is 10
a = 10
print(bool(a))
```

This produces the following result –

```
False
False
False
False
False
True
```

8. Python None Type

Python's none type is represented by the "**nonetype**." It is an object of its own data type. The **nonetype** represents the null type of values or absence of a value.

Example of None Type

In the following example, we are assigning None to a variable x and printing its type, which will be nonetype –

```
# Declaring a variable
# And, assigning a Null value (None)

x = None

# Printing its value and type
print("x = ", x)
print("type of x = ", type(x))
```

This produce the following result –

```
x = None
type of x = <class 'NoneType'>
```

Getting Data Type

To get the data types in Python, you can use the `type()` function. The `type()` is a built-in function that returns the class of the given object.

Example

In the following example, we are getting the type of the values and variables –

```
# Getting type of values
print(type(123))
print(type(9.99))

# Getting type of variables
a = 10
b = 2.12
c = "Hello"
d = (10, 20, 30)
e = [10, 20, 30]

print(type(a))
print(type(b))
print(type(c))
print(type(d))
print(type(e))
```

This produces the following result –

```
<class 'int'>
<class 'float'>
<class 'int'>
<class 'float'>
<class 'str'>
<class 'tuple'>
<class 'list'>
```

Setting Data Type

In Python, during declaring a variable or an object, you don't need to set the data types. Data type is set automatically based on the assigned value.

Example

The following example demonstrates how a variable's data type is set based on the given value –

```
# Declaring a variable
# And, assigning an integer value

x = 10

# Printing its value and type
print("x = ", x)
print("type of x = ", type(x))

# Now, assigning string value to
# the same variable
x = "Hello World!"

# Printing its value and type
print("x = ", x)
print("type of x = ", type(x))
```

This produces the following result –

```
x = 10
type of x = <class 'int'>
x = Hello World!
type of x = <class 'str'>
```

Primitive and Non-Primitive Data Types

The above-explained data types can also be categorized as primitive and non-primitive.

1. Primitive Types

The primitive data types are the fundamental data types that are used to create complex data types (sometimes called complex data structures). There are mainly four primitive data types, which are –

- Integers
- Floats
- Booleans, and
- Strings

2. Non-primitive Types

The non-primitive data types store values or collections of values. There are mainly four types of non-primitive types, which are –

Lists
Tuples
Dictionaries, and
Sets

Python Data Type Conversion

Sometimes, you may need to perform conversions between the built-in data types. To convert data between different Python data types, you simply use the type name as a function.

Read: [Python Type Casting](#)

Example

Following is an example which converts different values to integer, floating point and string values respectively –

```
print("Conversion to integer data type")

a = int(1)      # a will be 1
b = int(2.2)    # b will be 2
c = int("3.3")  # c will be 3

print (a)
print (b)
print (c)

print("Conversion to floating point number")

a = float(1)    # a will be 1.0
b = float(2.2)  # b will be 2.2
c = float("3.3") # c will be 3.3

print (a)
print (b)
print (c)

print("Conversion to string")

a = str(1)      # a will be "1"
b = str(2.2)    # b will be "2.2"
c = str("3.3")  # c will be "3.3"
```

```
print (a)
print (b)
print (c)
```

This produce the following result –

```
Conversion to integer data type
1
2
3
Conversion to floating point number
1.0
2.2
3.3
Conversion to string
1
2.2
3.3
```

Data Type Conversion Functions

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	Python int() function Converts x to an integer. base specifies the base if x is a string.
2	Python long() function Converts x to a long integer. base specifies the base if x is a string. <i>This function has been deprecated.</i>
3	Python float() function Converts x to a floating-point number.
4	Python complex() function Creates a complex number.
5	Python str() function Converts object x to a string representation.
6	Python repr() function Converts object x to an expression string.
7	Python eval() function

	Evaluates a string and returns an object.
8	Python tuple() function Converts s to a tuple.
9	Python list() function Converts s to a list.
10	Python set() function Converts s to a set.
11	Python dict() function Creates a dictionary. d must be a sequence of (key,value) tuples.
12	Python frozenset() function Converts s to a frozen set.
13	Python chr() function Converts an integer to a character.
14	Python unichr() function Converts an integer to a Unicode character.
15	Python ord() function Converts a single character to its integer value.
16	Python hex() function Converts an integer to a hexadecimal string.
17	Python oct() function Converts an integer to an octal string.

13. Python - Type Casting

From a programming point of view, a type casting refers to converting an object of one type into another. Here, we shall learn about type casting in Python Programming.

*Python Type Casting is a process in which we convert a Literal of one data type to another data type. Python supports two types of casting – **implicit** and **explicit**.*

In Python there are different data types, such as numbers, sequences, mappings etc. There may be a situation where, you have the available data of one type but you want to use it in another form. For example, the user has input a string but you want to use it as a number. Python's type casting mechanism let you do that.

Python Implicit Casting

When a compiler/interpreter of any language automatically converts object of one type into other, it is called automatic or **implicit casting**. Python is a strongly typed language. It doesn't allow automatic type conversion between unrelated data types. For example, a string cannot be converted to any number type. However, an integer can be cast into a float. Other languages such as JavaScript is a weakly typed language, where an integer is coerced into a string for concatenation.

Note that memory requirement of each data type is different. For example, an integer object in Python occupies 4 bytes of memory, while a **float** object needs 8 bytes because of its fractional part. Hence, Python interpreter doesn't automatically convert a **float** to **int**, because it will result in loss of data. On the other hand, int can be easily converted into float by setting its fractional part to 0.

Implicit **int** to **float** casting takes place when any arithmetic operation on int and float operands is done.

Consider we have one int and one float variable

```
<<< a=10    # int object  
<<< b=10.5 # float object
```

To perform their addition, 10 – the integer object is upgraded to 10.0. It is a float, but equivalent to its earlier numeric value. Now we can perform addition of two floats.

```
<<< c=a+b  
<<< print (c)  
20.5
```

In implicit type casting, a Python object with lesser byte size is upgraded to match the bigger byte size of other object in the operation. For example, a Boolean object is first upgraded to int and then to float, before the addition with a floating point object. In the following example, we try to add a Boolean object in a float, please note that True is equal to 1, and False is equal to 0.

```
a=True;
```

```
b=10.5;
c=a+b;
print (c);
```

This will produce the following result:

```
11.5
```

Python Explicit Casting

Although automatic or implicit casting is limited to int to float conversion, you can use Python's built-in functions `int()`, `float()` and `str()` to perform the explicit conversions such as string to integer.

Python `int()` Function

Python's built-in `int()` function converts an integer literal to an integer object, a float to integer, and a string to integer if the string itself has a valid integer literal representation.

Using `int()` with an `int` object as argument is equivalent to declaring an `int` object directly.

```
<<< a = int(10)
<<< a
10
```

is same as –

```
<<< a = 10
<<< a
10
<<< type(a)
<class 'int'>
```

If the argument to `int()` function is a float object or floating point expression, it returns an `int` object. For example –

```
<<< a = int(10.5) #converts a float object to int
<<< a
10
<<< a = int(2*3.14) #expression results float, is converted to int
<<< a
6
<<< type(a)
<class 'int'>
```

The `int()` function also returns integer 1 if a Boolean object is given as argument.

```
<<< a=int(True)
```

```
<<< a
1
<<< type(a)
<class 'int'>
```

String to Integer

The `int()` function returns an integer from a string object, only if it contains a valid integer representation.

```
<<< a = int("100")
<<< a
100
<<< type(a)
<class 'int'>
<<< a = ("10"+"01")
<<< a = int("10"+"01")
<<< a
1001
<<< type(a)
<class 'int'>
```

However, if the string contains a non-integer representation, Python raises `ValueError`.

```
<<< a = int("10.5")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '10.5'
<<< a = int("Hello World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello World'
```

The `int()` function also returns integer from binary, octal and hexa-decimal string. For this, the function needs a `base` parameter which must be 2, 8 or 16 respectively. The string should have a valid binary/octal/Hexa-decimal representation.

Binary String to Integer

The string should be made up of 1 and 0 only, and the base should be 2.

```
<<< a = int("110011", 2)
<<< a
```

51

The Decimal equivalent of binary number 110011 is 51.

Octal String to Integer

The string should only contain 0 to 7 digits, and the base should be 8.

```
<<< a = int("20", 8)
<<< a
16
```

The Decimal equivalent of octal 20 is 16.

Hexa-Decimal String to Integer

The string should contain only the Hexadecimal symbols i.e., 0-9 and A, B, C, D, E or F. Base should be 16.

```
<<< a = int("2A9", 16)
<<< a
681
```

Decimal equivalent of Hexadecimal 2A9 is 681. You can easily verify these conversions with calculator app in Windows, Ubuntu or Smartphones.

Following is an example to convert number, float and string into integer data type:

```
a = int(1)      # a will be 1
b = int(2.2)    # b will be 2
c = int("3")    # c will be 3

print (a)
print (b)
print (c)
```

This will produce the following result –

1
2
3

Python float() Function

The float() is a built-in function in Python. It returns a float object if the argument is a float literal, integer or a string with valid floating point representation.

Using float() with an float object as argument is equivalent to declaring a float object directly

```
<<< a = float(9.99)
```

```
<<< a
9.99
<<< type(a)
<class 'float'>
```

is same as –

```
<<< a = 9.99
<<< a
9.99
<<< type(a)
<class 'float'>
```

If the argument to float() function is an integer, the returned value is a floating point with fractional part set to 0.

```
<<< a = float(100)
<<< a
100.0
<<< type(a)
<class 'float'>
```

The float() function returns float object from a string, if the string contains a valid floating point number, otherwise ValueError is raised.

```
<<< a = float("9.99")
<<< a
9.99
<<< type(a)
<class 'float'>
<<< a = float("1,234.50")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '1,234.50'
```

The reason of ValueError here is the presence of comma in the string.

For the purpose of string to float conversion, the scientific notation of floating point is also considered valid.

```
<<< a = float("1.00E4")
<<< a
10000.0
<<< type(a)
```

```
<class 'float'>
<<< a = float("1.00E-4")
<<< a
0.0001
<<< type(a)
<class 'float'>
```

Following is an example to convert number, float and string into float data type:

```
a = float(1)      # a will be 1.0
b = float(2.2)    # b will be 2.2
c = float("3.3") # c will be 3.3

print (a)
print (b)
print (c)
```

This will produce the following result –

```
1.0
2.2
3.3
```

Python str() Function

We saw how a Python obtains integer or float number from corresponding string representation. The str() function works the opposite. It surrounds an integer or a float object with quotes ('') to return a str object. The str() function returns the string representation of any Python object. In this section, we shall see different examples of str() function in Python.

The str() function has three parameters. First required parameter (or argument) is the object whose string representation we want. Other two operators, encoding and errors, are optional.

We shall execute str() function in Python console to easily verify that the returned object is a string, with the enclosing quotation marks ('').

Integer to string

You can convert any integer number into a string as follows:

```
<<< a = str(10)
<<< a
'10'
<<< type(a)
<class 'str'>
```

Float to String

The str() function converts floating point objects with both the notations of floating point, standard notation with a decimal point separating integer and fractional part, and the scientific notation to string object.

```
<<< a=str(11.10)
<<< a
'11.1'
<<< type(a)
<class 'str'>
<<< a = str(2/5)
<<< a
'0.4'
<<< type(a)
<class 'str'>
```

In the second case, a division expression is given as argument to str() function. Note that the expression is evaluated first and then result is converted to string.

Floating points in scientific notations using E or e and with positive or negative power are converted to string with str() function.

```
<<< a=str(10E4)
<<< a
'100000.0'
<<< type(a)
<class 'str'>
<<< a=str(1.23e-4)
<<< a
'0.000123'
<<< type(a)
<class 'str'>
```

When Boolean constant is entered as argument, it is surrounded by (') so that True becomes 'True'. List and Tuple objects can also be given argument to str() function. The resultant string is the list/tuple surrounded by (').

```
<<< a=str('True')
<<< a
'True'
<<< a=str([1,2,3])
<<< a
'[1, 2, 3]' 
```

```
<<< a=str((1,2,3))
<<< a
'(1, 2, 3)'
<<< a=str({1:100, 2:200, 3:300})
<<< a
'{1: 100, 2: 200, 3: 300}'
```

Following is an example to convert number, float and string into string data type:

```
a = str(1)      # a will be "1"
b = str(2.2)    # b will be "2.2"
c = str("3.3") # c will be "3.3"

print (a)
print (b)
print (c)
```

This will produce the following result –

```
1
2.2
3.3
```

Conversion of Sequence Types

List, Tuple and String are Python's sequence types. They are ordered or indexed collection of items.

A string and tuple can be converted into a list object by using the `list()` function. Similarly, the `tuple()` function converts a string or list to a tuple.

We shall take an object of each of these three sequence types and study their inter-conversion.

```
<<< a=[1,2,3,4,5]  # List Object
<<< b=(1,2,3,4,5)  # Tuple Object
<<< c="Hello"       # String Object

### list() separates each character in the string and builds the list
<<< obj=list(c)
<<< obj
['H', 'e', 'l', 'l', 'o']

### The parentheses of tuple are replaced by square brackets
```

```

<<< obj=list(b)
<<< obj
[1, 2, 3, 4, 5]

### tuple() separates each character from string and builds a tuple of
characters
<<< obj=tuple(c)
<<< obj
('H', 'e', 'l', 'l', 'o')

### square brackets of list are replaced by parentheses.
<<< obj=tuple(a)
<<< obj
(1, 2, 3, 4, 5)

### str() function puts the list and tuple inside the quote symbols.
<<< obj=str(a)
<<< obj
'[1, 2, 3, 4, 5]'

<<< obj=str(b)
<<< obj
'(1, 2, 3, 4, 5)'

```

Thus Python's explicit type casting feature allows conversion of one data type to other with the help of its built-in functions.

Data Type Conversion Functions

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	Python int() function Converts x to an integer. base specifies the base if x is a string.
2	Python long() function Converts x to a long integer. base specifies the base if x is a string. <i>This function has been deprecated.</i>
3	Python float() function

	Converts x to a floating-point number.
4	Python complex() function Creates a complex number.
5	Python str() function Converts object x to a string representation.
6	Python repr() function Converts object x to an expression string.
7	Python eval() function Evaluates a string and returns an object.
8	Python tuple() function Converts s to a tuple.
9	Python list() function Converts s to a list.
10	Python set() function Converts s to a set.
11	Python dict() function Creates a dictionary. d must be a sequence of (key,value) tuples.
12	Python frozenset() function Converts s to a frozen set.
13	Python chr() function Converts an integer to a character.
14	Python unichr() function Converts an integer to a Unicode character.
15	Python ord() function Converts a single character to its integer value.
16	Python hex() function Converts an integer to a hexadecimal string.
17	Python oct() function Converts an integer to an octal string.

14. Python - Unicode System

What is Unicode System?

Software applications often require to display messages output in a variety in different languages such as in English, French, Japanese, Hebrew, or Hindi. Python's string type uses the Unicode Standard for representing characters. It makes the program possible to work with all these different possible characters.

A character is the smallest possible component of a text. 'A', 'B', 'C', etc., are all different characters. So are 'È' and 'Í'. A unicode string is a sequence of code points, which are numbers from 0 through 0x10FFFF (1,114,111 decimal). This sequence of code points needs to be represented in memory as a set of code units, and code units are then mapped to 8-bit bytes.

Character Encoding

A sequence of code points is represented in memory as a set of code units, mapped to 8-bit bytes. The rules for translating a Unicode string into a sequence of bytes are called a character encoding.

Three types of encodings are present, UTF-8, UTF-16 and UTF-32. UTF stands for Unicode Transformation Format.

Python's Unicode Support

Python 3.0 onwards has built-in support for Unicode. The str type contains Unicode characters, hence any string created using single, double or the triple-quoted string syntax is stored as Unicode. The default encoding for Python source code is UTF-8.

Hence, string may contain literal representation of a Unicode character (3/4) or its Unicode value (\u00BE).

Example

```
var = "3/4"  
print (var)  
  
var = "\u00BE"  
print (var)
```

This above code will produce the following output –

```
3/4  
¾
```

Example

In the following example, a string '10' is stored using the Unicode values of 1 and 0 which are \u0031 and u0030 respectively.

```
var = "\u0031\u0030"
print (var)
```

It will produce the following output –

```
10
```

Strings display the text in a human-readable format, and bytes store the characters as binary data. Encoding converts data from a character string to a series of bytes. Decoding translates the bytes back to human-readable characters and symbols. It is important not to confuse these two methods. Encode is a string method, while decode is a method of the Python byte object.

Example

In the following example, we have a string variable that consists of ASCII characters. ASCII is a subset of Unicode character set. The encode() method is used to convert it into a bytes object.

```
string = "Hello"
tobytes = string.encode('utf-8')
print (tobytes)
string = tobytes.decode('utf-8')
print (string)
```

The decode() method converts byte object back to the str object. The encoding method used is utf-8.

```
b'Hello'
Hello
```

Example

In the following example, the Rupee symbol (₹) is stored in the variable using its Unicode value. We convert the string to bytes and back to str.

```
string = "\u20B9"
print (string)
tobytes = string.encode('utf-8')
print (tobytes)
string = tobytes.decode('utf-8')
print (string)
```

When you execute the above code, it will produce the following output –

```
₹
b'\xe2\x82\xb9'
₹
```


15. Python - Literals

What are Python Literals?

Python literals or constants are the notation for representing a fixed value in source code. In contrast to variables, literals (123, 4.3, "Hello") are static values or you can say constants which do not change throughout the operation of the program or application. For example, in the following assignment statement:

```
x = 10
```

Here 10 is a literal as numeric value representing 10, which is directly stored in memory. However,

```
y = x*2
```

Here, even if the expression evaluates to 20, it is not literally included in source code. You can also declare an int object with built-in int() function. However, this is also an indirect way of instantiation and not with literal.

```
x = int(10)
```

Different Types of Python Literals

Python provides following literals which will be explained in this tutorial:

- Integer Literal
- Float Literal
- Complex Literal
- String Literal
- List Literal
- Tuple Literal
- Dictionary Literal

Python Integer Literal

Any representation involving only the digit symbols (0 to 9) creates an object of int type. The object so declared may be referred by a variable using an assignment operator.

Integer literals consist three different types of different literal values decimal, octal, and hexadecimal literals.

1. Decimal Literal

Decimal literals represent the signed or unsigned numbers. Digits from 0 to 9 are used to create a decimal literal value.

Look at the below statement assigning decimal literal to the variable –

```
x = 10
```

```
y = -25
```

```
z = 0
```

2. Octal Literal

Python allows an integer to be represented as an octal number or a hexadecimal number. A numeric representation with only eight digit symbols (0 to 7) but prefixed by 0o or 0O is an octal number in Python.

Look at the below statement assigning octal literal to the variable –

```
x = 0034
```

3. Hexadecimal Literal

Similarly, a series of hexadecimal symbols (0 to 9 and a to f), prefixed by 0x or 0X represents an integer in Hexadecimal form in Python.

Look at the below statement assigning hexadecimal literal to the variable –

```
x = 0X1C
```

However, it may be noted that, even if you use octal or hexadecimal literal notation, Python internally treats them as of int type.

Example

```
# Using Octal notation
x = 0034
print ("0034 in octal is", x, type(x))

# Using Hexadecimal notation
x = 0X1C
print ("0X1c in Hexadecimal is", x, type(x))
```

When you run this code, it will produce the following output –

```
0034 in octal is 28 <class 'int'>
0X1c in Hexadecimal is 28 <class 'int'>
```

Python Float Literal

A floating point number consists of an integral part and a fractional part. Conventionally, a decimal point symbol (.) separates these two parts in a literal representation of a float. For example,

Example of Float Literal

```
x = 25.55
y = 0.05
z = -12.2345
```

For a floating point number which is too large or too small, where number of digits before or after decimal point is more, a scientific notation is used for a compact literal

representation. The symbol E or e followed by positive or negative integer, follows after the integer part.

Example of Float Scientific Notation Literal

For example, a number 1.23E05 is equivalent to 123000.00. Similarly, 1.23e-2 is equivalent to 0.0123

```
# Using normal floating point notation
x = 1.23
print ("1.23 in normal float literal is", x, type(x))

# Using Scientific notation
x = 1.23E5
print ("1.23E5 in scientific notation is", x, type(x))
x = 1.23E-2
print ("1.23E-2 in scientific notation is", x, type(x))
```

Here, you will get the following output –

```
1.23 in normal float literal is 1.23 <class 'float'>
1.23E5 in scientific notation is 123000.0 <class 'float'>
1.23E-2 in scientific notation is 0.0123 <class 'float'>
```

Python Complex Literal

A complex number comprises of a real and imaginary component. The imaginary component is any number (integer or floating point) multiplied by square root of "-1"

($\sqrt{-1}$). In literal representation ($\sqrt{-1}$) is representation by "j" or "J". Hence, a literal representation of a complex number takes a form $x+yj$.

Example of Complex Type Literal

```
#Using literal notation of complex number
x = 2+3j
print ("2+3j complex literal is", x, type(x))
y = 2.5+4.6j
print ("2.5+4.6j complex literal is", x, type(x))
```

This code will produce the following output –

```
2+3j complex literal is (2+3j) <class 'complex'>
2.5+4.6j complex literal is (2+3j) <class 'complex'>
```

Python String Literal

A string object is one of the sequence data types in Python. It is an immutable sequence of Unicode code points. Code point is a number corresponding to a character according to Unicode standard. Strings are objects of Python's built-in class 'str'.

String literals are written by enclosing a sequence of characters in single quotes ('hello'), double quotes ("hello") or triple quotes (''''hello''' or """hello""").

Example of String Literal

```
var1='hello'

print ("'hello' in single quotes is:", var1, type(var1))

var2="hello"

print ('"hello" in double quotes is:', var1, type(var1))

var3=''''hello'''

print (''''''hello''' in triple quotes is:', var1, type(var1))

var4="""hello"""

print ('''''''hello''' in triple quotes is:', var1, type(var1))
```

Here, you will get the following output –

```
'hello' in single quotes is: hello <class 'str'>
"hello" in double quotes is: hello <class 'str'>
'''hello''' in triple quotes is: hello <class 'str'>
"""hello""" in triple quotes is: hello <class 'str'>
```

Example of String Literal with Double Quotes Inside String

If it is required to embed double quotes as a part of string, the string itself should be put in single quotes. On the other hand, if single quoted text is to be embedded, string should be written in double quotes.

```
var1='Welcome to "Python Tutorial" from TutorialsPoint'

print (var1)

var2="Welcome to 'Python Tutorial' from TutorialsPoint"

print (var2)
```

It will produce the following output –

```
Welcome to "Python Tutorial" from TutorialsPoint
Welcome to 'Python Tutorial' from TutorialsPoint
```

Python List Literal

List object in Python is a collection of objects of other data type. List is an ordered collection of items not necessarily of same type. Individual object in the collection is accessed by index starting with zero.

Literal representation of a list object is done with one or more items which are separated by comma and enclosed in square brackets [].

Example of List Type Literal

```
L1=[1, "Ravi", 75.50, True]
```

```
print (L1, type(L1))
```

It will produce the following output –

```
[1, 'Ravi', 75.5, True] <class 'list'>
```

Python Tuple Literal

Tuple object in Python is a collection of objects of other data type. Tuple is an ordered collection of items not necessarily of same type. Individual object in the collection is accessed by index starting with zero.

Literal representation of a tuple object is done with one or more items which are separated by comma and enclosed in parentheses () .

Example of Tuple Type Literal

```
T1=(1, "Ravi", 75.50, True)
print (T1, type(T1))
```

It will produce the following output –

```
[1, 'Ravi', 75.5, True] <class tuple>
```

Example of Tuple Type Literal Without Parenthesis

Default delimiter for Python sequence is parentheses, which means a comma separated sequence without parentheses also amounts to declaration of a tuple.

```
T1=1,"Ravi",75.50, True
print (T1, type(T1))
```

Here too, you will get the same output –

```
[1, 'Ravi', 75.5, True] <class tuple>
```

Python Dictionary Literal

Like list or tuple, dictionary is also a collection data type. However, it is not a sequence. It is an unordered collection of items, each of which is a key-value pair. Value is bound to key by the ":" symbol. One or more key:value pairs separated by comma are put inside curly brackets to form a dictionary object.

Example of Dictionary Type Literal

```
capitals={"USA":"New York", "France":"Paris", "Japan":"Tokyo",
"India":"New Delhi"}
numbers={1:"one", 2:"Two", 3:"three",4:"four"}
points={"p1":(10,10), "p2":(20,20)}

print (capitals, type(capitals))
print (numbers, type(numbers))
```

```
print (points, type(points))
```

Key should be an immutable object. Number, string or tuple can be used as key. Key cannot appear more than once in one collection. If a key appears more than once, only the last one will be retained. Values can be of any data type. One value can be assigned to more than one keys. For example,

```
staff={"Krishna":"Officer", "Rajesh":"Manager", "Ragini":"officer",
"Anil":"Clerk", "Kavita":"Manager"}
```

16. Python - Operators

Python Operators

Python operators are special symbols used to perform specific operations on one or more operands. The variables, values, or expressions can be used as operands. For example, Python's addition operator (+) is used to perform addition operations on two variables, values, or expressions.

The following are some of the terms related to Python operators:

- **Unary operators:** Python operators that require one operand to perform a specific operation are known as unary operators.
- **Binary operators:** Python operators that require two operands to perform a specific operation are known as binary operators.
- **Operands:** Variables, values, or expressions that are used with the operator to perform a specific operation are known as operands.

Types of Python Operators

Python operators are divided in the following categories –

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look at all the operators one by one.

Python Arithmetic Operators

Python Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, etc.

The following table contains all arithmetic operators with their symbols, names, and examples (assume that the values of a and b are 10 and 20, respectively) –

Operator	Name	Example
+	Addition	$a + b = 30$
-	Subtraction	$a - b = -10$
*	Multiplication	$a * b = 200$
/	Division	$b / a = 2$
%	Modulus	$b \% a = 0$
**	Exponent	$a^{**}b = 10^{**}20$
//	Floor Division	$9//2 = 4$

Example of Python Arithmetic Operators

```

a = 21
b = 10
c = 0

c = a + b
print ("a: {} b: {} a+b: {}".format(a,b,c))

c = a - b
print ("a: {} b: {} a-b: {}".format(a,b,c) )

c = a * b
print ("a: {} b: {} a*b: {}".format(a,b,c))

c = a / b
print ("a: {} b: {} a/b: {}".format(a,b,c))

c = a % b
print ("a: {} b: {} a%b: {}".format(a,b,c))

a = 2
b = 3
c = a**b
print ("a: {} b: {} a**b: {}".format(a,b,c))

a = 10
b = 5
c = a//b
print ("a: {} b: {} a//b: {}".format(a,b,c))

```

Output

```

a: 21 b: 10 a+b: 31
a: 21 b: 10 a-b: 11
a: 21 b: 10 a*b: 210
a: 21 b: 10 a/b: 2.1

```

```
a: 21 b: 10 a%b: 1
a: 2 b: 3 a**b: 8
a: 10 b: 5 a//b: 2
```

Python Comparison Operators

Python comparison operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

The following table contains all comparison operators with their symbols, names, and examples (assume that the values of a and b are 10 and 20, respectively) –

Operator	Name	Example
==	Equal	(a == b) is not true.
!=	Not equal	(a != b) is true.
>	Greater than	(a > b) is not true.
<	Less than	(a < b) is true.
>=	Greater than or equal to	(a >= b) is not true.
<=	Less than or equal to	(a <= b) is true.

Example of Python Comparison Operators

```
a = 21
b = 10
if ( a == b ):
    print ("Line 1 - a is equal to b")
else:
    print ("Line 1 - a is not equal to b")

if ( a != b ):
    print ("Line 2 - a is not equal to b")
else:
    print ("Line 2 - a is equal to b")

if ( a < b ):
    print ("Line 3 - a is less than b" )
else:
    print ("Line 3 - a is not less than b")

if ( a > b ):
```

```

print ("Line 4 - a is greater than b")
else:
    print ("Line 4 - a is not greater than b")

a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

if ( a <= b ):
    print ("Line 5 - a is either less than or equal to b")
else:
    print ("Line 5 - a is neither less than nor equal to b")

if ( b >= a ):
    print ("Line 6 - b is either greater than or equal to b")
else:
    print ("Line 6 - b is neither greater than nor equal to b")

```

Output

```

Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to b
Line 6 - b is either greater than or equal to b

```

Python Assignment Operators

Python Assignment operators are used to assign values to variables. Following is a table which shows all Python assignment operators.

The following table contains all assignment operators with their symbols, names, and examples –

Operator	Example	Same As
=	a = 10	a = 10
+=	a += 30	a = a + 30
-=	a -= 15	a = a - 15
*=	a *= 10	a = a * 10
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5
**=	a **= 4	a = a ** 4
//=	a // = 5	a = a // 5
&=	a &= 5	a = a & 5

<code> =</code>	<code>a = 5</code>	<code>a = a 5</code>
<code>^=</code>	<code>a ^= 5</code>	<code>a = a ^ 5</code>
<code>>>=</code>	<code>a >>= 5</code>	<code>a = a >> 5</code>
<code><<=</code>	<code>a <<= 5</code>	<code>a = a << 5</code>

Example of Python Assignment Operators

```

a = 21
b = 10
c = 0

print ("a: {} b: {} c : {}".format(a,b,c))

c = a + b
print ("a: {} c = a + b: {}".format(a,c))

c += a
print ("a: {} c += a: {}".format(a,c))

c *= a
print ("a: {} c *= a: {}".format(a,c))

c /= a
print ("a: {} c /= a : {}".format(a,c))

c = 2
print ("a: {} b: {} c : {}".format(a,b,c))
c %= a
print ("a: {} c %= a: {}".format(a,c))

c **= a
print ("a: {} c **= a: {}".format(a,c))

c //= a
print ("a: {} c //= a: {}".format(a,c))

```

Output

```

a: 21 b: 10 c: 0
a: 21  c = a + b: 31
a: 21 c += a: 52
a: 21 c *= a: 1092

```

```
a: 21 c /= a : 52.0
a: 21 b: 10 c : 2
a: 21 c %= a: 2
a: 21 c **= a: 2097152
a: 21 c //= a: 99864
```

Python Bitwise Operators

Python bitwise operator works on bits and performs bit by bit operation. These operators are used to compare binary numbers.

The following table contains all bitwise operators with their symbols, names, and examples

Operator	Name	Example
&	AND	a & b
	OR	a b
^	XOR	a ^ b
~	NOT	~a
<<	Zero fill left shift	a << 3
>>	Signed right shift	a >> 3

Example of Python Bitwise Operators

```
a = 20
b = 10

print ('a=',a,':',bin(a),'b=',b,':',bin(b))
c = 0

c = a & b;
print ("result of AND is ", c,':',bin(c))

c = a | b;
print ("result of OR is ", c,':',bin(c))

c = a ^ b;
print ("result of EXOR is ", c,':',bin(c))

c = ~a;
print ("result of COMPLEMENT is ", c,':',bin(c))
```

```
c = a << 2;
print ("result of LEFT SHIFT is ", c,':',bin(c))

c = a >> 2;
print ("result of RIGHT SHIFT is ", c,':',bin(c))
```

Output

```
a= 20 : 0b10100 b= 10 : 0b1010
result of AND is  0 : 0b0
result of OR is  30 : 0b11110
result of EXOR is  30 : 0b11110
result of COMPLEMENT is -21 : -0b10101
result of LEFT SHIFT is  80 : 0b1010000
result of RIGHT SHIFT is  5 : 0b101
```

Python Logical Operators

Python logical operators are used to combine two or more conditions and check the final result.

The following table contains all logical operators with their symbols, names, and examples

Operator	Name	Example
and	AND	a and b
or	OR	a or b
not	NOT	not(a)

Example of Python Logical Operators

```
var = 5

print(var > 3 and var < 10)
print(var > 3 or var < 4)
print(not (var > 3 and var < 10))
```

Output

```
True
True
False
```

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

There are two membership operators as explained below –

Operator	Description	Example
in	Returns True if it finds a variable in the specified sequence, false otherwise.	a in b
not in	returns True if it does not find a variable in the specified sequence and false otherwise.	a not in b

Example of Python Membership Operators

```

a = 10
b = 20
list = [1, 2, 3, 4, 5]

print ("a:", a, "b:", b, "list:", list)

if ( a in list ):
    print ("a is present in the given list")
else:
    print ("a is not present in the given list")

if ( b not in list ):
    print ("b is not present in the given list")
else:
    print ("b is present in the given list")

c=b/a
print ("c:", c, "list:", list)
if ( c in list ):
    print ("c is available in the given list")
else:
    print ("c is not available in the given list")

```

Output

```
a: 10 b: 20 list: [1, 2, 3, 4, 5]
a is not present in the given list
b is not present in the given list
c: 2.0 list: [1, 2, 3, 4, 5]
c is available in the given list
```

Python Identity Operators

Python identity operators compare the memory locations of two objects.

There are two Identity operators explained below –

Operator	Description	Example
is	Returns True if both variables are the same object and false otherwise.	a is b
is not	Returns True if both variables are not the same object and false otherwise.	a is not b

Example of Python Identity Operators

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
c = a

print(a is c)
print(a is b)

print(a is not c)
print(a is not b)
```

Output

```
True
False
False
True
```

Python Operators Precedence

Operators precedence decides the order of the evaluation in which an operator is evaluated. Python operators have different levels of precedence. The following table contains the list of operators having highest to lowest precedence –

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	**
	Exponentiation (raise to the power)
2	~ + -
	Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % //
	Multiply, divide, modulo and floor division
4	+ -
	Addition and subtraction
5	>><<
	Right and left bitwise shift
6	&
	Bitwise 'AND'
7	^
	Bitwise exclusive 'OR' and regular 'OR'
8	<= < > >=
	Comparison operators
9	<> == !=
	Equality operators
10	= %= /= //= -= += *= **=
	Assignment operators
11	is is not
	Identity operators
12	in not in
	Membership operators
13	not or and
	Logical operators

Read more about the Python operators precedence here: [Python operators precedence](#)

17. Python - Arithmetic Operators

Python Arithmetic Operators

Python arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, and more on numbers. Arithmetic operators are binary operators in the sense they operate on two operands. Python fully supports mixed arithmetic. That is, the two operands can be of two different number types. In such a situation.

Types of Arithmetic Operators

Following is the table which lists down all the arithmetic operators available in Python:

Operator	Name	Example
+	Addition	$a + b = 30$
-	Subtraction	$a - b = -10$
*	Multiplication	$a * b = 200$
/	Division	$b / a = 2$
%	Modulus	$b \% a = 0$
**	Exponent	$a^{**}b = 10^{**}20$
//	Floor Division	$9//2 = 4$

Let us study these operators with examples.

Addition Operator

The addition operator is represented by the + symbol. It is a basic arithmetic operator. It adds the two numeric operands on the either side and returns the addition result.

Example to add two integer numbers

In the following example, the two integer variables are the operands for the "+" operator.

```
a=10  
b=20  
  
print ("Addition of two integers")  
print ("a =",a,"b =",b,"addition =",a+b)
```

It will produce the following output –

```
Addition of two integers  
a = 10 b = 20 addition = 30
```

Example to add integer and float numbers

Addition of integer and float results in a float.

```
a=10
b=20.5
print ("Addition of integer and float")
print ("a =",a,"b =",b,"addition =",a+b)
```

It will produce the following output –

```
Addition of integer and float
a = 10 b = 20.5 addition = 30.5
```

Example to add two complex numbers

The result of adding float to complex is a complex number.

```
a=10+5j
b=20.5
print ("Addition of complex and float")
print ("a=",a,"b=",b,"addition=",a+b)
```

It will produce the following output –

```
Addition of complex and float
a= (10+5j) b= 20.5 addition= (30.5+5j)
```

Subtraction Operator

The subtraction operator is represented by the - symbol. It subtracts the second operand from the first. The resultant number is negative if the second operand is larger.

Example to subtract two integer numbers

First example shows subtraction of two integers.

```
a=10
b=20
print ("Subtraction of two integers:")
print ("a =",a,"b =",b,"a-b =",a-b)
print ("a =",a,"b =",b,"b-a =",b-a)
```

Result –

```
Subtraction of two integers
a = 10 b = 20 a-b = -10
a = 10 b = 20 b-a = 10
```

Example to subtract integer and float numbers

Subtraction of an integer and a float follows the same principle.

```
a=10
```

```
b=20.5
print ("subtraction of integer and float")
print ("a=",a,"b=",b,"a-b=",a-b)
print ("a=",a,"b=",b,"b-a=",b-a)
```

It will produce the following output –

```
subtraction of integer and float
a= 10 b= 20.5 a-b= -10.5
a= 10 b= 20.5 b-a= 10.5
```

Example to subtract complex numbers

In the subtraction involving a complex and a float, real component is involved in the operation.

```
a=10+5j
b=20.5
print ("subtraction of complex and float")
print ("a=",a,"b=",b,"a-b=",a-b)
print ("a=",a,"b=",b,"b-a=",b-a)
```

It will produce the following output –

```
subtraction of complex and float
a= (10+5j) b= 20.5 a-b= (-10.5+5j)
a= (10+5j) b= 20.5 b-a= (10.5-5j)
```

Multiplication Operator

The * (asterisk) symbol is defined as multiplication operator in Python (as in many languages). It returns the product of the two operands on its either side. If any of the operands negative, the result is also negative. If both are negative, the result is positive. Changing the order of operands doesn't change the result

Example to multiply two integers

```
a=10
b=20
print ("Multiplication of two integers")
print ("a =",a,"b =",b,"a*b =",a*b)
```

It will produce the following output –

```
Multiplication of two integers
a = 10 b = 20 a*b = 200
```

Example to multiply integer and float numbers

In multiplication, a float operand may have a standard decimal point notation, or a scientific notation.

```
a=10
b=20.5

print ("Multiplication of integer and float")
print ("a=",a,"b=",b,"a*b=",a*b)

a=-5.55
b=6.75E-3

print ("Multiplication of float and float")
print ("a =",a,"b =",b,"a*b =",a*b)
```

It will produce the following output –

```
Multiplication of integer and float
a = 10 b = 20.5 a*b = -10.5
Multiplication of float and float
a = -5.55 b = 0.00675 a*b = -0.03746249999999996
```

Example to multiply complex numbers

For the multiplication operation involving one complex operand, the other operand multiplies both the real part and imaginary part.

```
a=10+5j
b=20.5

print ("Multiplication of complex and float")
print ("a =",a,"b =",b,"a*b =",a*b)
```

It will produce the following output –

```
Multiplication of complex and float
a = (10+5j) b = 20.5 a*b = (205+102.5j)
```

Division Operator

The "/" symbol is usually called as forward slash. The result of division operator is numerator (left operand) divided by denominator (right operand). The resultant number is negative if any of the operands is negative. Since infinity cannot be stored in the memory, Python raises `ZeroDivisionError` if the denominator is 0.

The result of division operator in Python is always a float, even if both operands are integers.

Example to divide two numbers

```
a=10
b=20
```

```
print ("Division of two integers")
print ("a=",a,"b=",b,"a/b=",a/b)
print ("a=",a,"b=",b,"b/a=",b/a)
```

It will produce the following output –

```
Division of two integers
a= 10 b= 20 a/b= 0.5
a= 10 b= 20 b/a= 2.0
```

Example to divide two float numbers

In Division, a float operand may have a standard decimal point notation, or a scientific notation.

```
a=10
b=-20.5

print ("Division of integer and float")
print ("a=",a,"b=",b,"a/b=",a/b)

a=-2.50
b=1.25E2

print ("Division of float and float")
print ("a=",a,"b=",b,"a/b=",a/b)
```

It will produce the following output –

```
Division of integer and float
a= 10 b= -20.5 a/b= -0.4878048780487805

Division of float and float
a= -2.5 b= 125.0 a/b= -0.02
```

Example to divide complex numbers

When one of the operands is a complex number, division between the other operand and both parts of complex number (real and imaginary) object takes place.

```
a=7.5+7.5j
b=2.5

print ("Division of complex and float")
print ("a =",a,"b =",b,"a/b =",a/b)
print ("a =",a,"b =",b,"b/a =",b/a)
```

It will produce the following output –

```
Division of complex and float
a = (7.5+7.5j) b = 2.5 a/b = (3+3j)
```

```
a = (7.5+7.5j) b = 2.5 b/a = (0.1666666666666666-0.1666666666666666j)
```

If the numerator is 0, the result of division is always 0 except when denominator is 0, in which case, Python raises `ZeroDivisionError` with Division by Zero error message.

```
a=0
b=2.5
print ("a=",a,"b=",b,"a/b=",a/b)
print ("a=",a,"b=",b,"b/a=",b/a)
```

It will produce the following output –

```
a= 0 b= 2.5 a/b= 0.0
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 20, in <module>
    print ("a=",a,"b=",b,"b/a=",b/a)
                  ~^~
ZeroDivisionError: float division by zero
```

Modulus Operator

Python defines the "%" symbol, which is known as Percent symbol, as Modulus (or modulo) operator. It returns the remainder after the denominator divides the numerator. It can also be called Remainder operator. The result of the modulus operator is the number that remains after the integer quotient. To give an example, when 10 is divided by 3, the quotient is 3 and remainder is 1. Hence, `10%3` (normally pronounced as 10 mod 3) results in 1.

Example for modulus operation on integers

If both the operands are integers, the modulus value is an integer. If numerator is completely divisible, remainder is 0. If numerator is smaller than denominator, modulus is equal to the numerator. If denominator is 0, Python raises `ZeroDivisionError`.

```
a=10
b=2
print ("a=",a, "b=",b, "a%b=", a%b)
a=10
b=4
print ("a=",a, "b=",b, "a%b=", a%b)
print ("a=",a, "b=",b, "b%a=", b%a)
a=0
b=10
print ("a=",a, "b=",b, "a%b=", a%b)
print ("a=", a, "b=", b, "b%a=",b%a)
```

It will produce the following output –

```
a= 10 b= 2 a%b= 0
a= 10 b= 4 a%b= 2
a= 10 b= 4 b%a= 4
a= 0 b= 10 a%b= 0
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 13, in <module>
    print ("a=", a, "b=", b, "b%a=",b%a)
                  ~^~
ZeroDivisionError: integer modulo by zero
```

Example for modulus operation on floats

If any of the operands is a float, the mod value is always float.

```
a=10
b=2.5
print ("a=",a, "b=",b, "a%b=", a%b)
a=10
b=1.5
print ("a=",a, "b=",b, "a%b=", a%b)
a=7.7
b=2.5
print ("a=",a, "b=",b, "a%b=", a%b)
a=12.4
b=3
print ("a=",a, "b=",b, "a%b=", a%b)
```

It will produce the following output –

```
a= 10 b= 2.5 a%b= 0.0
a= 10 b= 1.5 a%b= 1.0
a= 7.7 b= 2.5 a%b= 0.20000000000000018
a= 12.4 b= 3 a%b= 0.4000000000000036
```

Python doesn't accept complex numbers to be used as operand in modulus operation. It throws `TypeError: unsupported operand type(s) for %.`

Exponent Operator

Python uses `**` (double asterisk) as the exponent operator (sometimes called raised to operator). So, for `a**b`, you say a raised to b, or even bth power of a.

If in the exponentiation expression, both operands are integer, result is also an integer. In case either one is a float, the result is float. Similarly, if either one operand is complex number, exponent operator returns a complex number.

If the base is 0, the result is 0, and if the index is 0 then the result is always 1.

Example of exponent operator

```
a=10
b=2
print ("a=",a, "b=",b, "a**b=", a**b)

a=10
b=1.5
print ("a=",a, "b=",b, "a**b=", a**b)

a=7.7
b=2
print ("a=",a, "b=",b, "a**b=", a**b)

a=1+2j
b=4
print ("a=",a, "b=",b, "a**b=", a**b)

a=12.4
b=0
print ("a=",a, "b=",b, "a**b=", a**b)
print ("a=",a, "b=",b, "b**a=", b**a)
```

It will produce the following output –

```
a= 10 b= 2 a**b= 100
a= 10 b= 1.5 a**b= 31.622776601683793
a= 7.7 b= 2 a**b= 59.290000000000006
a= (1+2j) b= 4 a**b= (-7-24j)
a= 12.4 b= 0 a**b= 1.0
a= 12.4 b= 0 b**a= 0.0
```

Floor Division Operator

Floor division is also called as integer division. Python uses // (double forward slash) symbol for the purpose. Unlike the modulus or modulo which returns the remainder, the floor division gives the quotient of the division of operands involved.

If both operands are positive, floor operator returns a number with fractional part removed from it. For example, the floor division of 9.8 by 2 returns 4 (pure division is 4.9, strip the fractional part, result is 4).

But if one of the operands is negative, the result is rounded away from zero (towards negative infinity). Floor division of -9.8 by 2 returns 5 (pure division is -4.9, rounded away from 0).

Example of floor division operator

```
a=9
b=2
print ("a=",a, "b=",b, "a//b=", a//b)
a=9
b=-2
print ("a=",a, "b=",b, "a//b=", a//b)
a=10
b=1.5
print ("a=",a, "b=",b, "a//b=", a//b)
a=-10
b=1.5
print ("a=",a, "b=",b, "a//b=", a//b)
```

It will produce the following output –

```
a= 9 b= 2 a//b= 4
a= 9 b= -2 a//b= -5
a= 10 b= 1.5 a//b= 6.0
a= -10 b= 1.5 a//b= -7.0
```

Precedence and Associativity of Arithmetic Operators

Operator(s)	Description	Associativity
**	Exponent Operator	Associativity of Exponent operator is from Right to Left .
%, *, /, //	Modulus, Multiplication, Division, and Floor Division	Associativity of Modulus, Multiplication, Division, and Floor Division operators are from Left to Right .
+, -	Addition and Subtraction Operators	Associativity of Addition and Subtraction operators are from Left to Right .

Arithmetic Operators with Complex Numbers

Arithmetic operators behave slightly differently when the both operands are complex number objects.

Addition and subtraction of complex numbers

Addition and subtraction of complex numbers is a simple addition/subtraction of respective real and imaginary components.

```
a=2.5+3.4j
b=-3+1.0j
print ("Addition of complex numbers - a=",a, "b=",b, "a+b=", a+b)
print ("Subtraction of complex numbers - a=",a, "b=",b, "a-b=", a-b)
```

It will produce the following output –

```
Addition of complex numbers - a= (2.5+3.4j) b= (-3+1j) a+b= (-0.5+4.4j)
Subtraction of complex numbers - a= (2.5+3.4j) b= (-3+1j) a-b= (5.5+2.4j)
```

Multiplication of complex numbers

Multiplication of complex numbers is similar to multiplication of two binomials in algebra. If " $a+bj$ " and " $x+yj$ " are two complex numbers, then their multiplication is given by this formula –

$$(a+bj)*(x+yj) = ax+ayj+xbj+byj^2 = (ax-by)+(ay+xb)j$$

For example,

```
a=6+4j
b=3+2j
c=a*b
c=(18-8)+(12+12)j
c=10+24j
```

The following program confirms the result –

```
a=6+4j
b=3+2j
print ("Multiplication of complex numbers - a=",a, "b=",b, "a*b=", a*b)
```

To understand the how the division of two complex numbers takes place, we should use the conjugate of a complex number. Python's complex object has a `conjugate()` method that returns a complex number with the sign of imaginary part reversed.

```
>>> a=5+6j
>>> a.conjugate()
(5-6j)
```

Division of complex numbers

To divide two complex numbers, divide and multiply the numerator as well as the denominator with the conjugate of denominator.

```
a=6+4j  
b=3+2j  
c=a/b  
c=(6+4j)/(3+2j)  
c=(6+4j)*(3-2j)/3+2j)*(3-2j)  
c=(18-12j+12j+8)/(9-6j+6j+4)  
c=26/13  
c=2+0j
```

To verify, run the following code –

```
a=6+4j  
b=3+2j  
print ("Division of complex numbers - a=",a, "b=",b, "a/b=", a/b)
```

Complex class in Python doesn't support the modulus operator (%) and floor division operator (//).

18. Python - Comparison Operators

Python Comparison Operators

Comparison operators in Python are very important in Python's conditional statements (if, else and elif) and looping statements (while and for loops). The comparison operators also called relational operators. Some of the well known operators are "<" stands for less than, and ">" stands for greater than operator.

Python uses two more operators, combining "=" symbol with these two. The "<=" symbol is for less than or equal to operator and the ">=" symbol is for greater than or equal to operator.

Different Comparison Operators in Python

Python has two more comparison operators in the form of "==" and "!=". They are for is equal to and is not equal to operators. Hence, there are six comparison operators in Python and they are listed below in this table:

Operator	Name	Example
<	Less than	a < b
>	Greater than	a > b
<=	Less than or equal to	a <= b
>=	Greater than or equal to	a >= b
==	Is equal to	a == b
!=	Is not equal to	a != b

Comparison operators are binary in nature, requiring two operands. An expression involving a comparison operator is called a Boolean expression, and always returns either True or False.

Example

```
a=5  
b=7  
print (a>b)  
print (a<b)
```

It will produce the following output –

```
False  
True
```

Both the operands may be Python literals, variables or expressions. Since Python supports mixed arithmetic, you can have any number type operands.

Example

The following code demonstrates the use of Python's comparison operators with integer numbers –

```

print ("Both operands are integer")
a=5
b=7
print ("a=",a, "b=",b, "a>b is", a>b)
print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)

```

It will produce the following output –

```

Both operands are integer
a= 5 b= 7 a>b is False
a= 5 b= 7 a<b is True
a= 5 b= 7 a==b is False
a= 5 b= 7 a!=b is True

```

Comparison of Float Number

In the following example, an integer operand and a float operand are compared.

Example

```

print ("comparison of int and float")
a=10
b=10.0
print ("a=",a, "b=",b, "a>b is", a>b)
print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)

```

It will produce the following output –

```

comparison of int and float
a= 10 b= 10.0 a>b is False
a= 10 b= 10.0 a<b is False
a= 10 b= 10.0 a==b is True
a= 10 b= 10.0 a!=b is False

```

Comparison of Complex Numbers

Although complex object is a number data type in Python, its behavior is different from others. Python doesn't support < and > operators, however it does support equality (==) and inequality (!=) operators.

Example

```
print ("comparison of complex numbers")
a=10+1j
b=10.-1j
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)
```

It will produce the following output –

```
comparison of complex numbers
a= (10+1j) b= (10-1j) a==b is False
a= (10+1j) b= (10-1j) a!=b is True
You get a TypeError with less than or greater than operators.
```

Example

```
print ("comparison of complex numbers")
a=10+1j
b=10.-1j
print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a>b is",a>b)
```

It will produce the following output –

```
comparison of complex numbers
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 5, in <module>
    print ("a=",a, "b=",b,"a<b is",a<b)
                                ^
TypeError: '<' not supported between instances of 'complex' and
'complex'
```

Comparison of Booleans

Boolean objects in Python are really integers: True is 1 and False is 0. In fact, Python treats any non-zero number as True. In Python, comparison of Boolean objects is possible. "False < True" is True!

Example

```
print ("comparison of Booleans")
a=True
b=False
print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a>b is",a>b)
```

```
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)
```

It will produce the following output –

```
comparison of Booleans
a= True b= False a<b is False
a= True b= False a>b is True
a= True b= False a==b is False
a= True b= False a!=b is True
```

Comparison of Sequence Types

In Python, comparison of only similar sequence objects can be performed. A string object is comparable with another string only. A list cannot be compared with a tuple, even if both have same items.

Example

```
print ("comparison of different sequence types")
a=(1,2,3)
b=[1,2,3]
print ("a=",a, "b=",b,"a<b is",a<b)
```

It will produce the following output –

```
comparison of different sequence types
Traceback (most recent call last):
  File "C:\Users\mlath\examples\example.py", line 5, in <module>
    print ("a=",a, "b=",b,"a<b is",a<b)
                                         ^
TypeError: '<' not supported between instances of 'tuple' and 'list'
```

Sequence objects are compared by lexicographical ordering mechanism. The comparison starts from item at 0th index. If they are equal, comparison moves to next index till the items at certain index happen to be not equal, or one of the sequences is exhausted. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one.

Which of the operands is greater depends on the difference in values of items at the index where they are unequal. For example, 'BAT'>'BAR' is True, as T comes after R in Unicode order.

If all items of two sequences compare equal, the sequences are considered equal.

Example

```
print ("comparison of strings")
a='BAT'
```

```
b='BALL'

print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a>b is",a>b)
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)
```

It will produce the following output –

```
comparison of strings

a= BAT b= BALL a<b is False
a= BAT b= BALL a>b is True
a= BAT b= BALL a==b is False
a= BAT b= BALL a!=b is True
```

In the following example, two tuple objects are compared –

Example

```
print ("comparison of tuples")

a=(1,2,4)
b=(1,2,3)

print ("a=",a, "b=",b,"a<b is",a<b)
print ("a=",a, "b=",b,"a>b is",a>b)
print ("a=",a, "b=",b,"a==b is",a==b)
print ("a=",a, "b=",b,"a!=b is",a!=b)
```

It will produce the following output –

```
a= (1, 2, 4) b= (1, 2, 3) a<b is False
a= (1, 2, 4) b= (1, 2, 3) a>b is True
a= (1, 2, 4) b= (1, 2, 3) a==b is False
a= (1, 2, 4) b= (1, 2, 3) a!=b is True
```

Comparison of Dictionary Objects

The use of "<" and ">" operators for Python's dictionary is not defined. In case of these operands, `TypeError: '<' not supported between instances of 'dict' and 'dict'` is reported.

Equality comparison checks if the length of both the dict items is same. Length of dictionary is the number of key-value pairs in it.

Python dictionaries are simply compared by length. The dictionary with fewer elements is considered less than a dictionary with more elements.

Example

```
print ("comparison of dictionary objects")

a={1:1,2:2}
```

```
b={2:2, 1:1, 3:3}  
print ("a=",a, "b=",b,"a==b is",a==b)  
print ("a=",a, "b=",b,"a!=b is",a!=b)
```

It will produce the following output –

```
comparison of dictionary objects  
a= {1: 1, 2: 2} b= {2: 2, 1: 1, 3: 3} a==b is False  
a= {1: 1, 2: 2} b= {2: 2, 1: 1, 3: 3} a!=b is True
```

19. Python - Assignment Operators

Python Assignment Operator

The = (equal to) symbol is defined as assignment operator in Python. The value of Python expression on its right is assigned to a single variable on its left. The = symbol as in programming in general (and Python in particular) should not be confused with its usage in Mathematics, where it states that the expressions on the either side of the symbol are equal.

Example of Assignment Operator in Python

Consider following Python statements –

```
a = 10  
b = 5  
a = a + b  
print (a)
```

At the first instance, at least for somebody new to programming but who knows maths, the statement "a=a+b" looks strange. How could a be equal to "a+b"? However, it needs to be reemphasized that the = symbol is an assignment operator here and not used to show the equality of LHS and RHS.

Because it is an assignment, the expression on right evaluates to 15, the value is assigned to a.

In the statement "a+=b", the two operators "+" and "=" can be combined in a "+=" operator. It is called as add and assign operator. In a single statement, it performs addition of two operands "a" and "b", and result is assigned to operand on left, i.e. "a".

Augmented Assignment Operators in Python

In addition to the simple assignment operators, Python provides few more assignment operators for advanced use. They are called cumulative or augmented assignment operators. In this chapter, we shall learn to use augmented assignment operators defined in Python.

Python has the augmented assignment operators for all arithmetic and comparison operators.

Python augmented assignment operators combines addition and assignment in one statement. Since Python supports mixed arithmetic, the two operands may be of different types. However, the type of left operand changes to the operand of on right, if it is wider.

Example

The += operator is an augmented operator. It is also called cumulative addition operator, as it adds "b" in "a" and assigns the result back to a variable.

The following are the augmented assignment operators in Python:

- Augmented Addition Operator
- Augmented Subtraction Operator
- Augmented Multiplication Operator
- Augmented Division Operator
- Augmented Modulus Operator
- Augmented Exponent Operator
- Augmented Floor division Operator

Augmented Addition Operator (+=)

Following examples will help in understanding how the "!=" operator works –

```
a=10
b=5
print ("Augmented addition of int and int")
a+=b # equivalent to a=a+b
print ("a=",a, "type(a):", type(a))

a=10
b=5.5
print ("Augmented addition of int and float")
a+=b # equivalent to a=a+b
print ("a=",a, "type(a):", type(a))

a=10.50
b=5+6j
print ("Augmented addition of float and complex")
a+=b #equivalent to a=a+b
print ("a=",a, "type(a):", type(a))

It will produce the following output -
Augmented addition of int and int
a= 15 type(a): <class 'int'>
Augmented addition of int and float
a= 15.5 type(a): <class 'float'>
Augmented addition of float and complex
a= (15.5+6j) type(a): <class 'complex'>
```

Augmented Subtraction Operator (-=)

Use -= symbol to perform subtract and assign operations in a single statement. The "a-=b" statement performs "a=a-b" assignment. Operands may be of any number type. Python performs implicit type casting on the object which is narrower in size.

```

a=10
b=5
print ("Augmented subtraction of int and int")
a-=b #equivalent to a=a-b
print ("a=",a, "type(a):", type(a))

a=10
b=5.5
print ("Augmented subtraction of int and float")
a-=b #equivalent to a=a-b
print ("a=",a, "type(a):", type(a))

a=10.50
b=5+6j
print ("Augmented subtraction of float and complex")
a-=b #equivalent to a=a-b
print ("a=",a, "type(a):", type(a))

```

It will produce the following output –

```

Augmented subtraction of int and int
a= 5 type(a): <class 'int'>
Augmented subtraction of int and float
a= 4.5 type(a): <class 'float'>
Augmented subtraction of float and complex
a= (5.5-6j) type(a): <class 'complex'>

```

Augmented Multiplication Operator (*=)

The *= operator works on similar principle. "a*=b" performs multiply and assign operations, and is equivalent to "a=a*b". In case of augmented multiplication of two complex numbers, the rule of multiplication as discussed in the previous chapter is applicable.

```

a=10
b=5
print ("Augmented multiplication of int and int")
a*=b #equivalent to a=a*b
print ("a=",a, "type(a):", type(a))

```

```

a=10
b=5.5
print ("Augmented multiplication of int and float")
a*=b #equivalent to a=a*b
print ("a=",a, "type(a):", type(a))

a=6+4j
b=3+2j
print ("Augmented multiplication of complex and complex")
a*=b #equivalent to a=a*b
print ("a=",a, "type(a):", type(a))

```

It will produce the following output –

```

Augmented multiplication of int and int
a= 50 type(a): <class 'int'>
Augmented multiplication of int and float
a= 55.0 type(a): <class 'float'>
Augmented multiplication of complex and complex
a= (10+24j) type(a): <class 'complex'>

```

Augmented Division Operator (/=)

The combination symbol "/=" acts as divide and assignment operator, hence "a/=b" is equivalent to "a=a/b". The division operation of int or float operands is float. Division of two complex numbers returns a complex number. Given below are examples of augmented division operator.

```

a=10
b=5
print ("Augmented division of int and int")
a/=b #equivalent to a=a/b
print ("a=",a, "type(a):", type(a))

a=10
b=5.5
print ("Augmented division of int and float")
a/=b #equivalent to a=a/b
print ("a=",a, "type(a):", type(a))

```

```
a=6+4j
b=3+2j
print ("Augmented division of complex and complex")
a/=b #equivalent to a=a/b
print ("a=",a, "type(a):", type(a))
```

It will produce the following output –

```
Augmented division of int and int
a= 2.0 type(a): <class 'float'>
Augmented division of int and float
a= 1.8181818181818181 type(a): <class 'float'>
Augmented division of complex and complex
a= (2+0j) type(a): <class 'complex'>
```

Augmented Modulus Operator (%=)

To perform modulus and assignment operation in a single statement, use the `%=` operator. Like the mod operator, its augmented version also is not supported for complex number.

```
a=10
b=5
print ("Augmented modulus operator with int and int")
a%=b #equivalent to a=a%b
print ("a=",a, "type(a):", type(a))

a=10
b=5.5
print ("Augmented modulus operator with int and float")
a%=b #equivalent to a=a%b
print ("a=",a, "type(a):", type(a))
```

It will produce the following output –

```
Augmented modulus operator with int and int
a= 0 type(a): <class 'int'>
Augmented modulus operator with int and float
a= 4.5 type(a): <class 'float'>
```

Augmented Exponent Operator (**=)

The `**=` operator results in computation of "a" raised to "b", and assigning the value back to "a". Given below are some examples –

```

a=10
b=5
print ("Augmented exponent operator with int and int")
a**=b #equivalent to a=a**b
print ("a=",a, "type(a):", type(a))

a=10
b=5.5
print ("Augmented exponent operator with int and float")
a**=b #equivalent to a=a**b
print ("a=",a, "type(a):", type(a))

a=6+4j
b=3+2j
print ("Augmented exponent operator with complex and complex")
a**=b #equivalent to a=a**b
print ("a=",a, "type(a):", type(a))

```

It will produce the following output –

```

Augmented exponent operator with int and int
a= 100000 type(a): <class 'int'>
Augmented exponent operator with int and float
a= 316227.7660168379 type(a): <class 'float'>
Augmented exponent operator with complex and complex
a= (97.52306038414744-62.22529992036203j) type(a): <class 'complex'>

```

Augmented Floor division Operator (//=)

For performing floor division and assignment in a single statement, use the "://" operator. "a//=b" is equivalent to "a=a//b". This operator cannot be used with complex numbers.

```

a=10
b=5
print ("Augmented floor division operator with int and int")
a//=b #equivalent to a=a//b
print ("a=",a, "type(a):", type(a))

a=10
b=5.5

```

```
print ("Augmented floor division operator with int and float")
a//=b #equivalent to a=a//b
print ("a=",a, "type(a):", type(a))
```

It will produce the following output –

```
Augmented floor division operator with int and int
a= 2 type(a): <class 'int'>
Augmented floor division operator with int and float
a= 1.0 type(a): <class 'float'>
```

20. Python - Logical Operators

Python Logical Operators

Python logical operators are used to form compound Boolean expressions. Each operand for these logical operators is itself a Boolean expression. For example,

Example

```
age > 16 and marks > 80  
percentage < 50 or attendance < 75
```

Along with the keyword `False`, Python interprets `None`, numeric zero of all types, and empty sequences (strings, tuples, lists), empty dictionaries, and empty sets as `False`. All other values are treated as `True`.

There are three logical operators in Python. They are "and", "or" and "not". They must be in lowercase.

Logical "and" Operator

For the compound Boolean expression to be `True`, both the operands must be `True`. If any or both operands evaluate to `False`, the expression returns `False`.

Logical "and" Operator Truth Table

The following table shows the scenarios.

a	b	a and b
F	F	F
F	T	F
T	F	F
T	T	T

Logical "or" Operator

In contrast, the `or` operator returns `True` if any of the operands is `True`. For the compound Boolean expression to be `False`, both the operands have to be `False`.

Logical "or" Operator Truth Table

The following table shows the result of the "or" operator with different conditions:

a	b	a or b
F	F	F
F	T	T
T	F	T
T	T	T

Logical "not" Operator

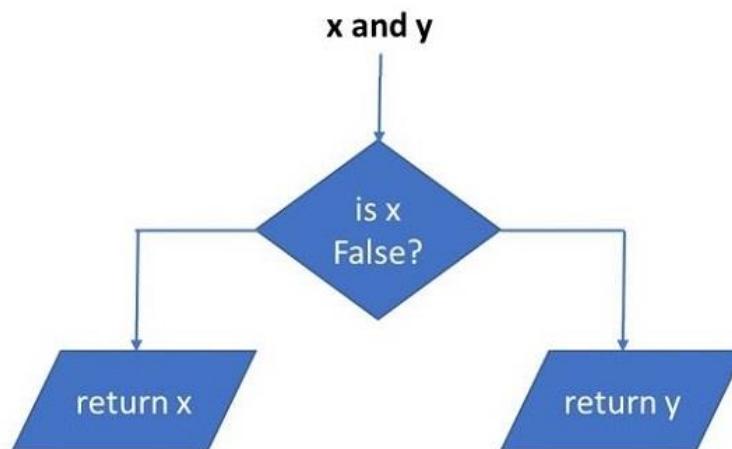
This is a unary operator. The state of Boolean operand that follows, is reversed. As a result, not True becomes False and not False becomes True.

Logical "not" Operator Truth Table

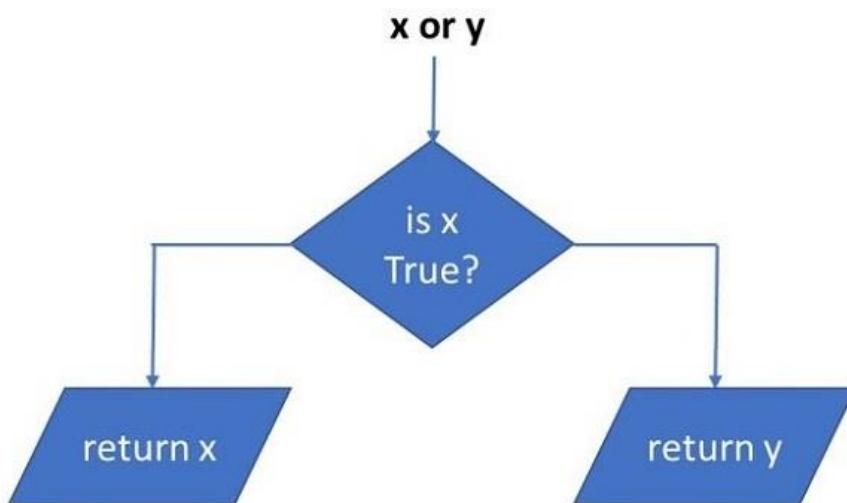
a	not(a)
F	T
T	F

How the Python interpreter evaluates the logical operators?

The expression "x and y" first evaluates "x". If "x" is false, its value is returned; otherwise, "y" is evaluated and the resulting value is returned.



The expression "x or y" first evaluates "x"; if "x" is true, its value is returned; otherwise, "y" is evaluated and the resulting value is returned.



Python Logical Operators Examples

Some use cases of logical operators are given below –

Example 1: Logical Operators with Boolean Conditions

```
x = 10
y = 20
print("x > 0 and x < 10:", x > 0 and x < 10)
print("x > 0 and y > 10:", x > 0 and y > 10)
print("x > 10 or y > 10:", x > 10 or y > 10)
print("x%2 == 0 and y%2 == 0:", x%2 == 0 and y%2 == 0)
print ("not (x+y>15):", not (x+y)>15)
```

It will produce the following output –

```
x > 0 and x < 10: False
x > 0 and y > 10: True
x > 10 or y > 10: True
x%2 == 0 and y%2 == 0: True
not (x+y>15): False
```

Example 2: Logical Operators with Non- Boolean Conditions

We can use non-boolean operands with logical operators. Here, we need to note that any non-zero numbers, and non-empty sequences evaluate to True. Hence, the same truth tables of logical operators apply.

In the following example, numeric operands are used for logical operators. The variables "x", "y" evaluate to True, "z" is False

```
x = 10
y = 20
z = 0
print("x and y:", x and y)
print("x or y:", x or y)
print("z or x:", z or x)
print("y or z:", y or z)
```

It will produce the following output –

```
x and y: 20
x or y: 10
z or x: 10
y or z: 20
```

Example 3: Logical Operators with Strings and Tuples

The string variable is treated as True and an empty tuple as False in the following example –

```
a="Hello"  
b=tuple()  
print("a and b:",a and b)  
print("b or a:",b or a)
```

It will produce the following output –

```
a and b: ()  
b or a: Hello
```

Example 4: Logical Operators to Compare Sequences (Lists)

Finally, two list objects below are non-empty. Hence x and y returns the latter, and x or y returns the former.

```
x=[1,2,3]  
y=[10,20,30]  
print("x and y:",x and y)  
print("x or y:",x or y)
```

It will produce the following output –

```
x and y: [10, 20, 30]  
x or y: [1, 2, 3]
```

21. Python - Bitwise Operators

Python Bitwise Operators

Python bitwise operators are normally used to perform bitwise operations on integer-type objects. However, instead of treating the object as a whole, it is treated as a string of bits. Different operations are done on each bit in the string.

Python has six bitwise operators - &, |, ^, ~, << and >>. All these operators (except ~) are binary in nature, in the sense they operate on two operands. Each operand is a binary digit (bit) 1 or 0.

The following are the bitwise operators in Python -

- Bitwise AND Operator
- Bitwise OR Operator
- Bitwise XOR Operator
- Bitwise NOT Operator
- Bitwise Left Shift Operator
- Bitwise Right Shift Operator

Python Bitwise AND Operator (&)

Bitwise AND operator is somewhat similar to logical and operator. It returns True only if both the bit operands are 1 (i.e. True). All the combinations are –

```
0 & 0 is 0
1 & 0 is 0
0 & 1 is 0
1 & 1 is 1
```

When you use integers as the operands, both are converted in equivalent binary, the & operation is done on corresponding bit from each number, starting from the least significant bit and going towards most significant bit.

Example of Bitwise AND Operator in Python

Let us take two integers 60 and 13, and assign them to variables a and b respectively.

```
a=60
b=13
print ("a:",a, "b:",b, "a&b:",a&b)
```

It will produce the following output –

```
a: 60 b: 13 a&b: 12
```

To understand how Python performs the operation, obtain the binary equivalent of each variable.

```
print ("a:", bin(a))
```

```
print ("b:", bin(b))
```

It will produce the following output –

```
a: 0b111100
b: 0b1101
```

For the sake of convenience, use the standard 8-bit format for each number, so that "a" is 00111100 and "b" is 00001101. Let us manually perform and operation on each corresponding bits of these two numbers.

```
0011 1100
&
0000 1101
-----
0000 1100
```

Convert the resultant binary back to integer. You'll get 12, which was the result obtained earlier.

```
>>> int('00001100',2)
12
```

Python Bitwise OR Operator (|)

The "|" symbol (called pipe) is the bitwise OR operator. If any bit operand is 1, the result is 1 otherwise it is 0.

```
0 | 0 is 0
0 | 1 is 1
1 | 0 is 1
1 | 1 is 1
```

Example of Bitwise OR Operator in Python

Take the same values of a=60, b=13. The "|" operation results in 61. Obtain their binary equivalents.

```
a=60
b=13
print ("a:",a, "b:",b, "a|b:",a|b)
print ("a:", bin(a))
print ("b:", bin(b))
```

It will produce the following output –

```
a: 60 b: 13 a|b: 61
a: 0b111100
```

```
b: 0b1101
```

To perform the "|" operation manually, use the 8-bit format.

```
0011 1100
|
0000 1101
-----
0011 1101
```

Convert the binary number back to integer to tally the result –

```
>>> int('00111101',2)
61
```

Python Bitwise XOR Operator (^)

The term XOR stands for exclusive OR. It means that the result of OR operation on two bits will be 1 if only one of the bits is 1.

```
0 ^ 0 is 0
0 ^ 1 is 1
1 ^ 0 is 1
1 ^ 1 is 0
```

Example of Bitwise XOR Operator in Python

Let us perform XOR operation on a=60 and b=13.

```
a=60
b=13
print ("a:",a, "b:",b, "a^b:",a^b)
```

It will produce the following output –

```
a: 60 b: 13 a^b: 49
```

We now perform the bitwise XOR manually.

```
0011 1100
 ^
0000 1101
-----
0011 0001
```

The int() function shows 00110001 to be 49.

```
>>> int('00110001',2)
49
```

Python Bitwise NOT Operator (~)

This operator is the binary equivalent of logical NOT operator. It flips each bit so that 1 is replaced by 0, and 0 by 1, and returns the complement of the original number. Python uses 2's complement method. For positive integers, it is obtained simply by reversing the bits. For negative number, $-x$, it is written using the bit pattern for $(x-1)$ with all of the bits complemented (switched from 1 to 0 or 0 to 1). Hence: (for 8 bit representation)

```
-1 is complement(1 - 1) = complement(0) = "11111111"
-10 is complement(10 - 1) = complement(9) = complement("00001001") = "11110110".
```

Example of Bitwise NOT Operator in Python

For $a=60$, its complement is –

```
a=60
print ("a:",a, "~a:", ~a)
```

It will produce the following output –

```
a: 60 ~a: -61
```

Python Bitwise Left Shift Operator (<<)

Left shift operator shifts most significant bits to right by the number on the right side of the "<<" symbol. Hence, " $x << 2$ " causes two bits of the binary representation of to right.

Example of Bitwise Left Shift Operator in Python

Let us perform left shift on 60.

```
a=60
print ("a:",a, "a<<2:", a<<2)
```

It will produce the following output –

```
a: 60 a<<2: 240
```

How does this take place? Let us use the binary equivalent of 60, and perform the left shift by 2.

```
0011 1100
<<
  2
-----
1111 0000
```

Convert the binary to integer. It is 240.

```
>>> int('11110000',2)
240
```

Python Bitwise Right Shift Operator (>>)

Right shift operator shifts least significant bits to left by the number on the right side of the ">>" symbol. Hence, "x >> 2" causes two bits of the binary representation of to left.

Example of Bitwise Right Shift Operator in Python

Let us perform right shift on 60.

```
a=60  
print ("a:",a, "a>>2:", a>>2)
```

It will produce the following output –

```
a: 60 a>>2: 15
```

Manual right shift operation on 60 is shown below –

```
0011 1100  
>>  
2  
-----  
0000 1111
```

Use int() function to convert the above binary number to integer. It is 15.

```
>>> int('00001111',2)  
15
```

22. Python - Membership Operators

Python Membership Operators

The membership operators in Python help us determine whether an item is present in a given container type object, or in other words, whether an item is a member of the given container type object.

Types of Python Membership Operators

Python has two membership operators: `in` and `not in`. Both return a Boolean result. The result of `in` operator is opposite to that of `not in` operator.

The '`in`' Operator

The "`in`" operator is used to check whether a substring is present in a bigger string, any item is present in a list or tuple, or a sub-list or sub-tuple is included in a list or tuple.

Example of Python Membership "`in`" Operator

In the following example, different substrings are checked whether they belong to the string `var="TutorialsPoint"`. Python differentiates characters on the basis of their Unicode value. Hence "To" is not the same as "to". Also note that if the "`in`" operator returns True, the "`not in`" operator evaluates to False.

```
var = "TutorialsPoint"
a = "P"
b = "tor"
c = "in"
d = "To"

print (a, "in", var, ":", a in var)
print (b, "in", var, ":", b in var)
print (c, "in", var, ":", c in var)
print (d, "in", var, ":", d in var)
```

It will produce the following output –

```
P in TutorialsPoint : True
tor in TutorialsPoint : True
in in TutorialsPoint : True
To in TutorialsPoint : False
```

The '`not in`' Operator

The "`not in`" operator is used to check a sequence with the given value is not present in the object like string, list, tuple, etc.

Example of Python Membership "not in" Operator

```
var = "TutorialsPoint"
a = "P"
b = "tor"
c = "in"
d = "To"

print (a, "not in", var, ":", a not in var)
print (b, "not in", var, ":", b not in var)
print (c, "not in", var, ":", c not in var)
print (d, "not in", var, ":", d not in var)
```

It will produce the following output –

```
P not in TutorialsPoint : False
tor not in TutorialsPoint : False
in not in TutorialsPoint : False
To not in TutorialsPoint : True
```

Membership Operator with Lists and Tuples

You can use the "in/not in" operator to check the membership of an item in the list or tuple.

```
var = [10,20,30,40]
a = 20
b = 10
c = a-b
d = a/2

print (a, "in", var, ":", a in var)
print (b, "not in", var, ":", b not in var)
print (c, "in", var, ":", c in var)
print (d, "not in", var, ":", d not in var)
```

It will produce the following output –

```
20 in [10, 20, 30, 40] : True
10 not in [10, 20, 30, 40] : False
10 in [10, 20, 30, 40] : True
10.0 not in [10, 20, 30, 40] : False
```

In the last case, "d" is a float but still it compares to True with 10 (an int) in the list. Even if a number expressed in other formats like binary, octal or hexadecimal are given, the membership operators tell if it is inside the sequence.

```
>>> 0x14 in [10, 20, 30, 40]
True
```

Example

However, if you try to check if two successive numbers are present in a list or tuple, the `in` operator returns `False`. If the list/tuple contains the successive numbers as a sequence itself, then it returns `True`.

```
var = (10,20,30,40)
a = 10
b = 20
print ((a,b), "in", var, ":", (a,b) in var)
var = ((10,20),30,40)
a = 10
b = 20
print ((a,b), "in", var, ":", (a,b) in var)
```

It will produce the following output –

```
(10, 20) in (10, 20, 30, 40) : False
(10, 20) in ((10, 20), 30, 40) : True
```

Membership Operator with Sets

Python's membership operators also work well with the set objects.

```
var = {10,20,30,40}
a = 10
b = 20
print (b, "in", var, ":", b in var)
var = {(10,20),30,40}
a = 10
b = 20
print ((a,b), "in", var, ":", (a,b) in var)
```

It will produce the following output –

```
20 in {40, 10, 20, 30} : True
(10, 20) in {40, 30, (10, 20)} : True
```

Membership Operator with Dictionaries

Use of `in` as well as `not in` operators with dictionary object is allowed. However, Python checks the membership only with the collection of keys and not values.

```
var = {1:10, 2:20, 3:30}  
a = 2  
b = 20  
print (a, "in", var, ":", a in var)  
print (b, "in", var, ":", b in var)
```

It will produce the following output –

```
2 in {1: 10, 2: 20, 3: 30} : True  
20 in {1: 10, 2: 20, 3: 30} : False
```

23. Python - Identity Operators

Python Identity Operators

The identity operators compare the objects to determine whether they share the same memory and refer to the same object type (data type).

Python provided two identity operators; we have listed them as follows:

- 'is' Operator
- 'is not' Operator

Python 'is' Operator

The 'is' operator evaluates to True if both the operand objects share the same memory location. The memory location of the object can be obtained by the "id()" function. If the "id()" of both variables is same, the "is" operator returns True.

Example of Python Identity 'is' Operator

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
c = a

# Comparing and printing return values
print(a is c)
print(a is b)

# Printing IDs of a, b, and c
print("id(a) : ", id(a))
print("id(b) : ", id(b))
print("id(c) : ", id(c))
```

It will produce the following output –

```
True
False
id(a) : 140114091859456
id(b) : 140114091906944
id(c) : 140114091859456
```

Python 'is not' Operator

The 'is not' operator evaluates to True if both the operand objects do not share the same memory location or both operands are not the same objects.

Example of Python Identity 'is not' Operator

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
c = a

# Comparing and printing return values
print(a is not c)
print(a is not b)

# Printing IDs of a, b, and c
print("id(a) : ", id(a))
print("id(b) : ", id(b))
print("id(c) : ", id(c))
```

It will produce the following output –

```
False
True
id(a) : 140559927442176
id(b) : 140559925598080
id(c) : 140559927442176
```

Python Identity Operators Examples with Explanations

Example 1

```
a="TutorialsPoint"
b=a
print ("id(a), id(b):", id(a), id(b))
print ("a is b:", a is b)
print ("b is not a:", b is not a)
```

It will produce the following output –

```
id(a), id(b): 2739311598832 2739311598832
a is b: True
b is not a: False
```

The list and tuple objects behave differently, which might look strange in the first instance. In the following example, two lists "a" and "b" contain same items. But their id() differs.

Example 2

```
a=[1,2,3]
b=[1,2,3]
print ("id(a), id(b):", id(a), id(b))
print ("a is b:", a is b)
print ("b is not a:", b is not a)
```

It will produce the following output –

```
id(a), id(b): 1552612704640 1552567805568
a is b: False
b is not a: True
```

The list or tuple contains the memory locations of individual items only and not the items itself. Hence "a" contains the addresses of 10,20 and 30 integer objects in a certain location which may be different from that of "b".

Example 3

```
print (id(a[0]), id(a[1]), id(a[2]))
print (id(b[0]), id(b[1]), id(b[2]))
```

It will produce the following output –

```
140734682034984 140734682035016 140734682035048
140734682034984 140734682035016 140734682035048
```

Because of two different locations of "a" and "b", the "is" operator returns False even if the two lists contain same numbers.

24. Python Operator Precedence

Python Operator Precedence

An expression may have multiple operators to be evaluated. The operator precedence defines the order in which operators are evaluated. In other words, the order of operator evaluation is determined by the operator precedence.

If a certain expression contains multiple operators, their order of evaluation is determined by the order of precedence. For example, consider the following expression

```
>>> a = 2+3*5
```

Here, what will be the value of a? - yes it will be 17 (multiply 3 by 5 first and then add 2) or 25 (adding 2 and 3 and then multiply with 5)? Python's operator precedence rule comes into picture here.

If we consider only the arithmetic operators in Python, the traditional BODMAS rule is also employed by Python interpreter, where the brackets are evaluated first, the division and multiplication operators next, followed by addition and subtraction operators. Hence, a will become 17 in the above expression.

In addition to the operator precedence, the associativity of operators is also important. If an expression consists of operators with same level of precedence, the associativity determines the order. Most of the operators have left to right associativity. It means, the operator on the left is evaluated before the one on the right.

Let us consider another expression:

```
>>> b = 10/5*4
```

In this case, both * (multiplication) and / (division) operators have same level of precedence. However, the left to right associativity rule performs the division first ($10/5 = 2$) and then the multiplication ($2*4 = 8$).

Python Operator Precedence Table

The following table lists all the operators in Python in their decreasing order of precedence. Operators in the same cell under the Operators column have the same precedence.

Sr.No.	Operator & Description
1	(), [], { }
	Parentheses and braces
2	[index], [index:index]
	Subscription, slicing,
3	await x
	Await expression
4	**
	Exponentiation
5	+x, -x, ~x
	Positive, negative, bitwise NOT

	* , @ , / , // , %
6	Multiplication, matrix multiplication, division, floor division, remainder
7	+ , - Addition and subtraction
8	<< , >> Left Shifts, Right Shifts
9	& Bitwise AND
10	^ Bitwise XOR
11	 Bitwise OR
12	in , not in , is , is not , < , <= , > , >= , != , == Comparisons, including membership tests and identity tests
13	not x Boolean NOT
14	and Boolean AND
15	or Boolean OR
16	if – else Conditional expression
17	lambda Lambda expression
18	:= Walrus operator

Python Operator Precedence Example

```
a = 20
b = 10
c = 15
d = 5
e = 0
e = (a + b) * c / d      #( 30 * 15 ) / 5
print ("Value of (a + b) * c / d is ", e)

e = ((a + b) * c) / d    # (30 * 15 ) / 5
print ("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);      # (30) * (15/5)
print ("Value of (a + b) * (c / d) is ", e)

e = a + (b * c) / d;       # 20 + (150/5)
print ("Value of a + (b * c) / d is ", e)
```

When you execute the above program, it produces the following result –

```
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

25. Python - Comments

Python Comments

Python comments are programmer-readable explanation or annotations in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter. Comments enhance the readability of the code and help the programmers to understand the code very carefully.

Example

If we execute the code given below, the output produced will simply print "Hello, World!" to the console, as comments are ignored by the Python interpreter and do not affect the execution of the program –

```
# This is a comment  
print("Hello, World!")
```

Python supports three types of comments as shown below –

- Single-line comments
- Multi-line comments
- Docstring Comments

Single Line Comments in Python

Single-line comments in Python start with a hash symbol (#) and extend to the end of the line. They are used to provide short explanations or notes about the code. They can be placed on their own line above the code they describe, or at the end of a line of code (known as an inline comment) to provide context or clarification about that specific line.

Example: Standalone Single-Line Comment

A standalone single-line comment is a comment that occupies an entire line by itself, starting with a hash symbol (#). It is placed above the code it describes or annotates.

In this example, the standalone single-line comment is placed above the "greet" function "_"

```
# Standalone single line comment is placed here  
def greet():  
    print("Hello, World!")  
greet()
```

Example: Inline Single-Line Comment

An inline single-line comment is a comment that appears on the same line as a piece of code, following the code and preceded by a hash symbol (#).

Here the inline single-line comment follows the print("Hello, World!") statement –

```
print("Hello, World!") # Inline single line comment is placed here
```

Multi Line Comments in Python

In Python, multi-line comments are used to provide longer explanations or notes that span multiple lines. While Python does not have a specific syntax for multi-line comments, there are two common ways to achieve this: consecutive single-line comments and triple-quoted strings –

Consecutive Single-Line Comments

Consecutive single-line comments refer to using the hash symbol (#) at the beginning of each line. This method is often used for longer explanations or to sections of parts of the code.

Example

In this example, multiple lines of comments are used to explain the purpose and logic of the factorial function –

```
# This function calculates the factorial of a number
# using an iterative approach. The factorial of a number
# n is the product of all positive integers less than or
# equal to n. For example, factorial(5) is 5*4*3*2*1 = 120.

def factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers"
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

number = 5
print(f"The factorial of {number} is {factorial(number)}")
```

Multi Line Comment Using Triple Quoted Strings

We can use triple-quoted strings ("'" or "'''") to create multi-line comments. These strings are technically string literals but can be used as comments if they are not assigned to any variable or used in expressions.

This pattern is often used for block comments or when documenting sections of code that require detailed explanations.

Example

Here, the triple-quoted string provides a detailed explanation of the "gcd" function, describing its purpose and the algorithm used –

```
"""
This function calculates the greatest common divisor (GCD)
of two numbers using the Euclidean algorithm. The GCD of
```

```

two numbers is the largest number that divides both of them
without leaving a remainder.

"""

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

result = gcd(48, 18)
print("The GCD of 48 and 18 is:", result)

```

Using Comments for Documentation

In Python, documentation comments, also known as docstrings, provide a way to incorporate documentation within your code. This can be useful for explaining the purpose and usage of modules, classes, functions, and methods. Effective use of documentation comments helps other developers understand your code and its purpose without needing to read through all the details of the implementation.

Python Docstrings

In Python, docstrings are a special type of comment that is used to document modules, classes, functions, and methods. They are written using triple quotes (" or """) and are placed immediately after the definition of the entity they document.

Docstrings can be accessed programmatically, making them an integral part of Python's built-in documentation tools.

Example of a Function Docstring

```

def greet(name):
    """
    This function greets the person whose name is passed as a parameter.

    Parameters:
    name (str): The name of the person to greet

    Returns:
    None
    """

    print(f"Hello, {name}!")
greet("Alice")

```

Accessing Docstrings

Docstrings can be accessed using the `.__doc__` attribute or the `help()` function. This makes it easy to view the documentation for any module, class, function, or method directly from the interactive Python shell or within the code.

Example: Using the `.__doc__` attribute

```
def greet(name):
    """
    This function greets the person whose name is passed as a parameter.

    Parameters:
    name (str): The name of the person to greet

    Returns:
    None
    """

print(greet.__doc__)
```

Example: Using the `help()` Function

```
def greet(name):
    """
    This function greets the person whose name is passed as a parameter.

    Parameters:
    name (str): The name of the person to greet

    Returns:
    None
    """

help(greet)
```

26. Python - User Input

Provide User Input in Python

In this chapter, we will learn how Python accepts the user input from the console, and displays the output on the same console.

Every computer application should have a provision to accept input from the user when it is running. This makes the application interactive. Depending on how it is developed, an application may accept the user input in the form of text entered in the console (**sys.stdin**), a graphical layout, or a web-based interface.

Python User Input Functions

Python provides us with two built-in functions to read the input from the keyboard.

- The `input()` Function
- The `raw_input()` Function

Python interpreter works in interactive and scripted mode. While the interactive mode is good for quick evaluations, it is less productive. For repeated execution of same set of instructions, scripted mode should be used.

Let us write a simple Python script to start with.

```
#!/usr/bin/python3.11
name = "Kiran"
city = "Hyderabad"
print ("Hello My name is", name)
print ("I am from", city)
```

Save the above code as `hello.py` and run it from the command-line. Here's the output

```
C:\python311> python hello.py
Hello My name is Kiran
I am from Hyderabad
```

The program simply prints the values of the two variables in it. If you run the program repeatedly, the same output will be displayed every time. To use the program for another name and city, you can edit the code, change name to say "Ravi" and city to "Chennai". Every time you need to assign different value, you will have to edit the program, save and run, which is not the ideal way.

The `input()` Function

Obviously, you need some mechanism to assign different values to the variable in the runtime – while the program is running. Python's `input()` function does the same job.

Following is the syntax of Python's standard library `input()` function.

```
>>> var = input()
```

When the interpreter encounters `input()` function, it waits for the user to enter data from the standard input stream (keyboard) till the Enter key is pressed. The sequence of characters may be stored in a string variable for further use.

On reading the Enter key, the program proceeds to the next statement. Let change our program to store the user input in `name` and `city` variables.

```
#!/usr/bin/python3.11
name = input()
city = input()

print ("Hello My name is", name)
print ("I am from ", city)
```

When you run, you will find the cursor waiting for user's input. Enter values for `name` and `city`. Using the entered data, the output will be displayed.

```
Ravi
Chennai
Hello My name is Ravi
I am from Chennai
```

Now, the variables are not assigned any specific value in the program. Every time you run, different values can be input. So, your program has become truly interactive.

Inside the `input()` function, you may give a prompt text, which will appear before the cursor when you run the code.

```
#!/usr/bin/python3.11
name = input("Enter your name : ")
city = input("Enter your city : ")
print ("Hello My name is", name)
print ("I am from ", city)
```

When you run the program displays the prompt message, basically helping the user what to enter.

```
Enter your name: Praveen Rao
Enter your city: Bengaluru
Hello My name is Praveen Rao
I am from Bengaluru
```

The `raw_input()` Function

The `raw_input()` function works similar to `input()` function. Here only point is that this function was available in Python 2.7, and it has been renamed to `input()` in Python 3.6

Following is the syntax of the raw_input() function:

```
>>> var = raw_input ([prompt text])
```

Let's re-write the above program using raw_input() function:

```
#!/usr/bin/python3.11

name = raw_input("Eneter your name - ")
city = raw_input("Enter city name - ")

print ("Hello My name is", name)
print ("I am from ", city)
```

When you run, you will find the cursor waiting for user's input. Enter values for name and city. Using the entered data, the output will be displayed.

```
Eneter your name - Ravi
Enter city name - Chennai
Hello My name is Ravi
I am from Chennai
```

Taking Numeric Input in Python

Let us write a Python code that accepts width and height of a rectangle from the user and computes the area.

```
#!/usr/bin/python3.11

width = input("Enter width : ")
height = input("Enter height : ")

area = width*height
print ("Area of rectangle = ", area)
```

Run the program, and enter width and height.

```
Enter width: 20
Enter height: 30
Traceback (most recent call last):
  File "C:\Python311\var1.py", line 5, in <module>
    area = width*height
TypeError: can't multiply sequence by non-int of type 'str'
```

Why do you get a `TypeError` here? The reason is, Python always read the user input as a string. Hence, `width="20"` and `height="30"` are the strings and obviously you cannot perform multiplication of two strings.

To overcome this problem, we shall use `int()`, another built-in function from Python's standard library. It converts a string object to an integer.

To accept an integer input from the user, read the input in a string, and type cast it to integer with `int()` function –

```
w = input("Enter width : ")
width = int(w)

h = input("Enter height : ")
height = int(h)
```

You can combine the input and type cast statements in one –

```
#!/usr/bin/python3.11

width = int(input("Enter width : "))
height = int(input("Enter height : "))

area = width*height
print ("Area of rectangle = ", area)
```

Now you can input any integer value to the two variables in the program –

```
Enter width: 20
Enter height: 30
Area of rectangle = 600
```

Python's `float()` function converts a string into a float object. The following program accepts the user input and parses it to a float variable – `rate`, and computes the interest on an amount which is also input by the user.

```
#!/usr/bin/python3.11

amount = float(input("Enter Amount : "))
rate = float(input("Enter rate of interest : "))

interest = amount*rate/100
print ("Amount: ", amount, "Interest: ", interest)
```

The program ask user to enter amount and rate; and displays the result as follows –

```
Enter Amount: 12500
```

```
Enter rate of interest: 6.5
Amount: 12500.0 Interest: 812.5
```

The print() Function

Python's print() function is a built-in function. It is the most frequently used function, that displays value of Python expression given in parenthesis, on Python's console, or standard output (**sys.stdout**).

```
print ("Hello World ")
```

Any number of Python expressions can be there inside the parenthesis. They must be separated by comma symbol. Each item in the list may be any Python object, or a valid Python expression.

```
#!/usr/bin/python3.11

a = "Hello World"
b = 100
c = 25.50
d = 5+6j
print ("Message: a")
print (b, c, b-c)
print(pow(100, 0.5), pow(c,2))
```

The first call to print() displays a string literal and a string variable. The second prints value of two variables and their subtraction. The pow() function computes the square root of a number and square value of a variable.

```
Message Hello World
100 25.5 74.5
10.0 650.25
```

If there are multiple comma separated objects in the print() function's parenthesis, the values are separated by a white space " ". To use any other character as a separator, define a sep parameter for the print() function. This parameter should follow the list of expressions to be printed.

In the following output of print() function, the variables are separated by comma.

```
#!/usr/bin/python3.11

city="Hyderabad"
state="Telangana"
country="India"
print(city, state, country, sep=',')
```

The effect of `sep=','` can be seen in the result –

```
Hyderabad,Telangana,India
```

The `print()` function issues a newline character ('\n') at the end, by default. As a result, the output of the next `print()` statement appears in the next line of the console.

```
city="Hyderabad"  
state="Telangana"  
print("City:", city)  
print("State:", state)
```

Two lines are displayed as the output –

```
City: Hyderabad  
State: Telangana
```

To make these two lines appear in the same line, define `end` parameter in the first `print()` function and set it to a whitespace string " ".

```
city="Hyderabad"  
state="Telangana"  
country="India"  
  
print("City:", city, end=" ")  
print("State:", state)
```

Output of both the `print()` functions appear in continuation.

```
City: Hyderabad State: Telangana
```

27. Python - Numbers

Python has built-in support to store and process numeric data (**Python Numbers**). Most of the times you work with numbers in almost every Python application. Obviously, any computer application deals with numbers. This tutorial will discuss about different types of Python Numbers and their properties.

Python - Number Types

There are three built-in number types available in Python:

- integers (int)
- floating point numbers (float)
- complex numbers

Python also has a built-in Boolean data type called bool. It can be treated as a sub-type of int type, since its two possible values True and False represent the integers 1 and 0 respectively.

Python – Integer Numbers

In Python, any number without the provision to store a fractional part is an integer. (Note that if the fractional part in a number is 0, it doesn't mean that it is an integer. For example, a number 10.0 is not an integer, it is a float with 0 fractional part whose numeric value is 10.) An integer can be zero, positive or a negative whole number. For example, 1234, 0, -55 all represent integers in Python.

There are three ways to form an integer object. With (a) literal representation, (b) any expression evaluating to an integer, and (c) using int() function.

Literal is a notation used to represent a constant directly in the source code. For example –

```
>>> a =10
```

However, look at the following assignment of the integer variable c.

```
a = 10
b = 20
c = a + b

print ("a:", a, "type:", type(a))
print ("c:", c, "type:", type(c))
```

It will produce the following output –

```
a: 10 type: <class 'int'>
c: 30 type: <class 'int'>
```

Here, c is indeed an integer variable, but the expression a + b is evaluated first, and its value is indirectly assigned to c.

The third method of forming an integer object is with the return value of int() function. It converts a floating point number or a string to an integer.

```
>>> a=int(10.5)
>>> b=int("100")
```

You can represent an integer as a binary, octal or hexa-decimal number. However, internally the object is stored as an integer.

Binary Numbers in Python

A number consisting of only the binary digits (1 and 0) and prefixed with "0b" is a binary number. If you assign a binary number to a variable, it still is an int variable.

A represent an integer in binary form, store it directly as a literal, or use int() function, in which the base is set to 2

```
a=0b101
print ("a:",a, "type:",type(a))

b=int("0b101011", 2)
print ("b:",b, "type:",type(b))
```

It will produce the following output –

```
a: 5 type: <class 'int'>
b: 43 type: <class 'int'>
```

There is also a bin() function in Python. It returns a binary string equivalent of an integer.

```
a=43
b=bin(a)
print ("Integer:",a, "Binary equivalent:",b)
```

It will produce the following output –

```
Integer: 43 Binary equivalent: 0b101011
```

Octal Numbers in Python

An octal number is made up of digits 0 to 7 only. In order to specify that the integer uses octal notation, it needs to be prefixed by "0o" (lowercase O) or "0O" (uppercase O). A literal representation of octal number is as follows –

```
a=00107
print (a, type(a))
```

It will produce the following output –

```
71 <class 'int'>
```

Note that the object is internally stored as integer. Decimal equivalent of octal number 107 is 71.

Since octal number system has 8 symbols (0 to 7), its base is 7. Hence, while using int() function to convert an octal string to integer, you need to set the base argument to 8.

```
a=int('20',8)
print (a, type(a))
```

It will produce the following output –

```
16 <class 'int'>
```

Decimal equivalent of octal 30 is 16.

In the following code, two int objects are obtained from octal notations and their addition is performed.

```
a=0056
print ("a:",a, "type:",type(a))

b=int("0031",8)
print ("b:",b, "type:",type(b))

c=a+b
print ("addition:", c)
```

It will produce the following output –

```
a: 46 type: <class 'int'>
b: 25 type: <class 'int'>
addition: 71
```

To obtain the octal string for an integer, use oct() function.

```
a=oct(71)
print (a, type(a))
```

Hexa-decimal Numbers in Python

As the name suggests, there are 16 symbols in the Hexadecimal number system. They are 0-9 and A to F. The first 10 digits are same as decimal digits. The alphabets A, B, C, D, E and F are equivalents of 11, 12, 13, 14, 15, and 16 respectively. Upper or lower cases may be used for these letter symbols.

For the literal representation of an integer in Hexadecimal notation, prefix it by "0x" or "OX".

```
a=0XA2
print (a, type(a))
```

It will produce the following output –

```
162 <class 'int'>
```

To convert a hexadecimal string to integer, set the base to 16 in the int() function.

```
a=int('0X1e', 16)
print (a, type(a))
```

Try out the following code snippet. It takes a Hexadecimal string, and returns the integer.

```
num_string = "A1"
number = int(num_string, 16)
print ("Hexadecimal:", num_string, "Integer:", number)
```

It will produce the following output –

```
Hexadecimal: A1 Integer: 161
```

However, if the string contains any symbol apart from the Hexadecimal symbol chart an error will be generated.

```
num_string = "A1X001"
print (int(num_string, 16))
```

The above program generates the following error –

```
Traceback (most recent call last):
  File "/home/main.py", line 2, in
    print (int(num_string, 16))
ValueError: invalid literal for int() with base 16: 'A1X001'
```

Python's standard library has hex() function, with which you can obtain a hexadecimal equivalent of an integer.

```
a=hex(161)
print (a, type(a))
```

It will produce the following output –

```
0xa1 <class 'str'>
```

Though an integer can be represented as binary or octal or hexadecimal, internally it is still integer. So, when performing arithmetic operation, the representation doesn't matter.

```
a=10 #decimal
b=0b10 #binary
c=0010 #octal
d=0XA #Hexadecimal
e=a+b+c+d
```

```
print ("addition:", e)
```

It will produce the following output –

```
addition: 30
```

Python – Floating Point Numbers

A floating point number has an integer part and a fractional part, separated by a decimal point symbol (.). By default, the number is positive, prefix a dash (-) symbol for a negative number.

A floating point number is an object of Python's float class. To store a float object, you may use a literal notation, use the value of an arithmetic expression, or use the return value of float() function.

Using literal is the most direct way. Just assign a number with fractional part to a variable. Each of the following statements declares a float object.

```
>>> a=9.99
>>> b=0.999
>>> c=-9.99
>>> d=-0.999
```

In Python, there is no restriction on how many digits after the decimal point can a floating point number have. However, to shorten the representation, the E or e symbol is used. E stands for Ten raised to. For example, E4 is 10 raised to 4 (or 4th power of 10), e-3 is 10 raised to -3.

In scientific notation, number has a coefficient and exponent part. The coefficient should be a float greater than or equal to 1 but less than 10. Hence, 1.23E+3, 9.9E-5, and 1E10 are the examples of floats with scientific notation.

```
>>> a=1E10
>>> a
10000000000.0
>>> b=9.90E-5
>>> b
9.9e-05
>>> 1.23E3
1230.0
```

The second approach of forming a float object is indirect, using the result of an expression. Here, the quotient of two floats is assigned to a variable, which refers to a float object.

```
a=10.33
b=2.66
c=a/b
```

```
print ("c:", c, "type", type(c))
```

It will produce the following output –

```
c: 3.8834586466165413 type <class 'float'>
```

Python's float() function returns a float object, parsing a number or a string if it has the appropriate contents. If no arguments are given in the parenthesis, it returns 0.0, and for an int argument, fractional part with 0 is added.

```
>>> a=float()
>>> a
0.0
>>> a=float(10)
>>> a
10.0
```

Even if the integer is expressed in binary, octal or hexadecimal, the float() function returns a float with fractional part as 0.

```
a=float(0b10)
b=float(0010)
c=float(0xA)

print (a,b,c, sep=",")
```

It will produce the following output –

```
2.0,8.0,10.0
```

The float() function retrieves a floating point number out of a string that encloses a float, either in standard decimal point format, or having scientific notation.

```
a=float("-123.54")
b=float("1.23E04")
print ("a=",a,"b=",b)
```

It will produce the following output –

```
a= -123.54 b= 12300.0
```

In mathematics, infinity is an abstract concept. Physically, infinitely large number can never be stored in any amount of memory. For most of the computer hardware configurations, however, a very large number with 400th power of 10 is represented by Inf. If you use "Infinity" as argument for float() function, it returns Inf.

```
a=1.00E400
print (a, type(a))
a=float("Infinity")
```

```
print (a, type(a))
```

It will produce the following output –

```
inf <class 'float'>
inf <class 'float'>
```

One more such entity is Nan (stands for Not a Number). It represents any value that is undefined or not representable.

```
>>> a=float('Nan')
>>> a
Nan
```

Python – Complex Numbers

In this section, we shall know in detail about Complex data type in Python. Complex numbers find their applications in mathematical equations and laws in electromagnetism, electronics, optics, and quantum theory. Fourier transforms use complex numbers. They are used in calculations with wavefunctions, designing filters, signal integrity in digital electronics, radio astronomy, etc.

A complex number consists of a real part and an imaginary part, separated by either "+" or "-". The real part can be any floating point (or itself a complex number) number. The imaginary part is also a float/complex, but multiplied by an imaginary number.

In mathematics, an imaginary number "i" is defined as the square root of -1 ($\sqrt{-1}$). Therefore, a complex number is represented as " $x+yi$ ", where x is the real part, and " y " is the coefficient of imaginary part.

Quite often, the symbol "j" is used instead of "I" for the imaginary number, to avoid confusion with its usage as current in theory of electricity. Python also uses "j" as the imaginary number. Hence, " $x+yj$ " is the representation of complex number in Python.

Like int or float data type, a complex object can be formed with literal representation or using complex() function. All the following statements form a complex object.

```
>>> a=5+6j
>>> a
(5+6j)
>>> type(a)
<class 'complex'>
>>> a=2.25-1.2j
>>> a
(2.25-1.2j)
>>> type(a)
<class 'complex'>
>>> a=1.01E-2+2.2e3j
```

```
>>> a
(0.0101+2200j)
>>> type(a)
<class 'complex'>
```

Note that the real part as well as the coefficient of imaginary part have to be floats, and they may be expressed in standard decimal point notation or scientific notation.

Python's `complex()` function helps in forming an object of complex type. The function receives arguments for real and imaginary part, and returns the complex number.

There are two versions of `complex()` function, with two arguments and with one argument. Use of `complex()` with two arguments is straightforward. It uses first argument as real part and second as coefficient of imaginary part.

```
a=complex(5.3,6)
b=complex(1.01E-2, 2.2E3)
print ("a:", a, "type:", type(a))
print ("b:", b, "type:", type(b))
```

It will produce the following output –

```
a: (5.3+6j) type: <class 'complex'>
b: (0.0101+2200j) type: <class 'complex'>
```

In the above example, we have used `x` and `y` as float parameters. They can even be of complex data type.

```
a=complex(1+2j, 2-3j)
print (a, type(a))
```

It will produce the following output –

```
(4+4j) <class 'complex'>
```

Surprised by the above example? Put "`x`" as `1+2j` and "`y`" as `2-3j`. Try to perform manual computation of "`x+yj`" and you'll come to know.

```
complex(1+2j, 2-3j)
=(1+2j)+(2-3j)*j
=1+2j +2j+3
=4+4j
```

If you use only one numeric argument for `complex()` function, it treats it as the value of real part; and imaginary part is set to 0.

```
a=complex(5.3)
print ("a:", a, "type:", type(a))
```

It will produce the following output –

```
a: (5.3+0j) type: <class 'complex'>
```

The complex() function can also parse a string into a complex number if its only argument is a string having complex number representation.

In the following snippet, user is asked to input a complex number. It is used as argument. Since Python reads the input as a string, the function extracts the complex object from it.

```
a= "5.5+2.3j"
b=complex(a)
print ("Complex number:", b)
```

It will produce the following output –

```
Complex number: (5.5+2.3j)
```

Python's built-in complex class has two attributes real and imag – they return the real and coefficient of imaginary part from the object.

```
a=5+6j
print ("Real part:", a.real, "Coefficient of Imaginary part:", a.imag)
```

It will produce the following output –

```
Real part: 5.0 Coefficient of Imaginary part: 6.0
```

The complex class also defines a conjugate() method. It returns another complex number with the sign of imaginary component reversed. For example, conjugate of $x+yj$ is $x-yj$.

```
>>> a=5-2.2j
>>> a.conjugate()
(5+2.2j)
```

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type int(x) to convert x to a plain integer.
- Type long(x) to convert x to a long integer.
- Type float(x) to convert x to a floating-point number.
- Type complex(x) to convert x to a complex number with real part x and imaginary part zero. In the same way type complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Let us see various numeric and math-related functions.

Theoretic and Representation Functions

Python includes following theoretic and representation functions in the math module –

Sr.No.	Function & Description
1	math.ceil(x)

	The ceiling of x: the smallest integer not less than x
2	math.comb(n,k) This function is used to find the returns the number of ways to choose "x" items from "y" items without repetition and without order.
3	math.copysign(x, y) This function returns a float with the magnitude (absolute value) of x but the sign of y.
4	math.cmp(x, y) This function is used to compare the values of two objects. This function is deprecated in Python3.
5	math.fabs(x) This function is used to calculate the absolute value of a given integer.
6	math.factorial(n) This function is used to find the factorial of a given integer.
7	math.floor(x) This function calculates the floor value of a given integer.
8	math.fmod(x, y) The fmod() function in math module returns same result as the "%" operator. However fmod() gives more accurate result of modulo division than modulo operator.
9	math.frexp(x) This function is used to calculate the mantissa and exponent of a given number.
10	math.fsum(iterable) This function returns the floating point sum of all numeric items in an iterable i.e. list, tuple, array.
11	math.gcd(*integers) This function is used to calculate the greatest common divisor of all the given integers.
12	math.isclose() This function is used to determine whether two given numeric values are close to each other.
13	math.isfinite(x)

	This function is used to determine whether the given number is a finite number.
14	math.isinf(x) This function is used to determine whether the given value is infinity (+ve or, -ve).
15	math.isnan(x) This function is used to determine whether the given number is "NaN".
16	math.isqrt(n) This function calculates the integer square-root of the given non negative integer.
17	math.lcm(*integers) This function is used to calculate the least common factor of the given integer arguments.
18	math.ldexp(x, i) This function returns product of first number with exponent of second number. So, ldexp(x,y) returns $x \cdot 2^y$. This is inverse of frexp() function.
19	math.modf(x) This returns the fractional and integer parts of x in a two-item tuple.
20	math.nextafter(x, y, steps) This function returns the next floating-point value after x towards y.
21	math.perm(n, k) This function is used to calculate the permutation. It returns the number of ways to choose x items from y items without repetition and with order.
22	math.prod(iterable, *, start) This function is used to calculate the product of all numeric items in the iterable (list, tuple) given as argument.
23	math.remainder(x,y) This function returns the remainder of x with respect to y. This is the difference $x - n \cdot y$, where n is the integer closest to the quotient x / y .
24	math.trunk(x)

	This function returns integral part of the number, removing the fractional part. <code>trunc()</code> is equivalent to <code>floor()</code> for positive x , and equivalent to <code>ceil()</code> for negative x .
25	<p><u>math.ulp(x)</u></p> <p>This function returns the value of the least significant bit of the float x. <code>trunc()</code> is equivalent to <code>floor()</code> for positive x, and equivalent to <code>ceil()</code> for negative x.</p>

Power and Logarithmic Functions

Sr.No.	Function & Description
1	<p><u>math.cbrt(x)</u></p> <p>This function is used to calculate the cube root of a number.</p>
2	<p><u>math.exp(x)</u></p> <p>This function calculate the exponential of x: e^x</p>
3	<p><u>math.exp2(x)</u></p> <p>This function returns 2 raised to power x. It is equivalent to 2^{**x}.</p>
4	<p><u>math.expm1(x)</u></p> <p>This function returns e raised to the power x, minus 1. Here e is the base of natural logarithms.</p>
5	<p><u>math.log(x)</u></p> <p>This function calculates the natural logarithm of x, for $x > 0$.</p>
6	<p><u>math.log1p(x)</u></p> <p>This function returns the natural logarithm of $1+x$ (base e). The result is calculated in a way which is accurate for x near zero.</p>
7	<p><u>math.log2(x)</u></p> <p>This function returns the base-2 logarithm of x. This is usually more accurate than $\log(x, 2)$.</p>
8	<p><u>math.log10(x)</u></p>

	The base-10 logarithm of x for $x > 0$.
9	math.pow(x, y) The value of $x^{**}y$.
10	math.sqrt(x) The square root of x for $x > 0$

Trigonometric Functions

Python includes following functions that perform trigonometric calculations in the math module –

Sr.No.	Function & Description
1	math.acos(x) This function returns the arc cosine of x, in radians.
2	math.asin(x) This function returns the arc sine of x, in radians.
3	math.atan(x) This function returns the arc tangent of x, in radians.
4	math.atan2(y, x) This function returns atan(y / x), in radians.
5	math.cos(x) This function returns the cosine of x radians.
6	math.sin(x) This function returns the sine of x radians.
7	math.tan(x) This function returns the tangent of x radians.
8	math.hypot(x, y) This function returns the Euclidean norm, $\sqrt{x^*x + y^*y}$.

Angular conversion Functions

Following are the angular conversion function provided by Python math module –

Sr.No.	Function & Description
1	math.degrees(x) This function converts the given angle from radians to degrees.
2	math.radians(x)

This function converts the given angle from degrees to radians.

Mathematical Constants

The Python math module defines the following mathematical constants –

Sr.No.	Constants & Description
1	math.pi This represents the mathematical constant pi, which equals to "3.141592..." to available precision.
2	math.e This represents the mathematical constant e, which is equal to "2.718281..." to available precision.
3	math. number This represents the mathematical constant Tau (denoted by τ). It is equivalent to the ratio of circumference to radius, and is equal to 2π .
4	math.inf This represents positive infinity. For negative infinity use " -math.inf ".
5	math. in This constant is a floating-point "not a number" (NaN) value. Its value is equivalent to the output of <code>float('nan')</code> .

Hyperbolic Functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles. Following are the hyperbolic functions of the Python math module –

Sr.No.	Function & Description
1	math.acosh(x) This function is used to calculate the inverse hyperbolic cosine of the given value.
2	math.asinh(x)

	This function is used to calculate the inverse hyperbolic sine of a given number.
3	math.atanh(x) This function is used to calculate the inverse hyperbolic tangent of a number.
4	math.cosh(x) This function is used to calculate the hyperbolic cosine of the given value.
5	math.sinh(x) This function is used to calculate the hyperbolic sine of a given number.
6	math.tanh(x) This function is used to calculate the hyperbolic tangent of a number.

Special Functions

Following are the special functions provided by the Python math module –

Sr.No.	Function & Description
1	math.erf(x) This function returns the value of the Gauss error function for the given parameter.
2	math.erfc(x) This function is the complementary for the error function. Value of erf(x) is equivalent to 1-erf(x) .
3	math.gamma(x) This is used to calculate the factorial of the complex numbers. It is defined for all the complex numbers except the non-positive integers.
4	math.lgamma(x) This function is used to calculate the natural logarithm of the absolute value of the Gamma function at x.

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions in the random module.

Sr.No.	Function & Description
1	random.choice(seq) A random item from a list, tuple, or string.
2	random.randrange([start,] stop [,step]) A randomly selected element from range(start, stop, step)
3	random.random() A random float r, such that 0 is less than or equal to r and r is less than 1
4	random.seed([x]) This function sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
5	random.shuffle(seq) This function is used to randomize the items of the given sequence.
6	random.uniform(a, b) This function returns a random floating point value r, such that a is less than or equal to r and r is less than b.

Built-in Mathematical Functions

Following mathematical functions are built into the Python interpreter, hence you don't need to import them from any module.

Sr.No.	Function & Description
1	Python abs() function The abs() function returns the absolute value of x, i.e. the positive distance between x and zero.
2	Python max() function

	The max() function returns the largest of its arguments or largest number from the iterable (list or tuple).
3	Python min() function The function min() returns the smallest of its arguments i.e. the value closest to negative infinity, or smallest number from the iterable (list or tuple)
4	Python pow() function The pow() function returns x raised to y. It is equivalent to $x^{**}y$.
5	Python round() Function round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.
6	Python sum() function The sum() function returns the sum of all numeric items in any iterable (list or tuple). It has an optional <i>start</i> argument which is 0 by default. If given, the numbers in the list are added to start value.

28. Python - Booleans

Python Booleans (bool)

In Python, bool is a sub-type of int type. A bool object has two possible values, and it is initialized with Python keywords, True and False.

Example

```
>>> a=True  
>>> b=False  
>>> type(a), type(b)  
(<class 'bool'>, <class 'bool'>)
```

A bool object is accepted as argument to type conversion functions. With True as argument, the int() function returns 1, float() returns 1.0; whereas for False, they return 0 and 0.0 respectively. We have a one argument version of complex() function.

If the argument is a complex object, it is taken as real part, setting the imaginary coefficient to 0.

Example

```
a=int(True)  
print ("bool to int:", a)  
a=float(False)  
print ("bool to float:", a)  
a=complex(True)  
print ("bool to complex:", a)
```

On running this code, you will get the following output –

```
bool to int: 1  
bool to float: 0.0  
bool to complex: (1+0j)
```

Python Boolean Expression

Python boolean expression is an expression that evaluates to a Boolean value. It almost always involves a comparison operator. In the below example, we will see how the comparison operators can give us the Boolean values. The bool() method is used to return the truth value of an expression.

```
Syntax: bool([x])  
Returns True if X evaluates to true else false.  
Without parameters it returns false.
```

Below we have examples which use numbers streams and Boolean values as parameters to the bool function. The results come out as true or false depending on the parameter.

Example

```
# Check true
a = True
print(bool(a))

# Check false
a = False
print(bool(a))

# Check 0
a = 0.0
print(bool(a))

# Check 1
a = 1.0
print(bool(a))

# Check Equality
a = 5
b = 10
print(bool( a==b))

# Check None
a = None
print(bool(a))

# Check an empty sequence
a = ()
print(bool(a))

# Check an empty mapping
a = {}
print(bool(a))

# Check a non empty string
a = 'Tutorialspoint'
print(bool(a))
```

Python Control Statements

29. Python - Control Flow

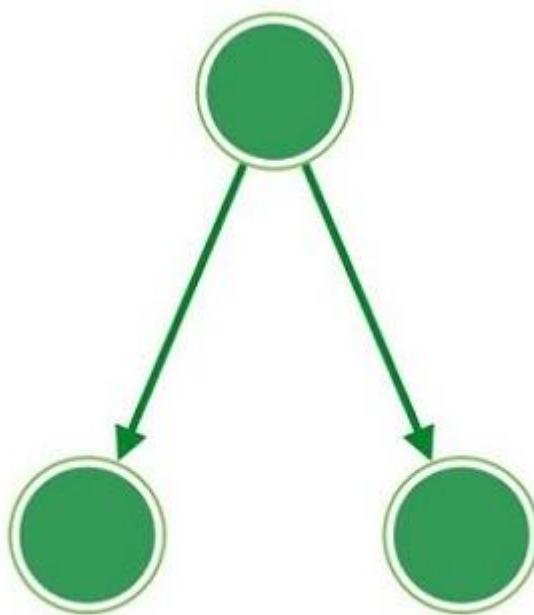
Python program control flow is regulated by various types of conditional statements, loops, and function calls. By default, the instructions in a computer program are executed in a sequential manner, from top to bottom, or from start to end. However, such sequentially executing programs can perform only simplistic tasks. We would like the program to have a decision-making ability, so that it performs different steps depending on different conditions.

Most programming languages including Python provide functionality to control the flow of execution of instructions. Normally, there are two type of control flow statements in any programming language and Python also supports them.

Decision Making Statements

Decision making statements are used in the Python programs to make them able to decide which of the alternative group of instructions to be executed, depending on value of a certain Boolean expression.

The following diagram illustrates how decision-making statements work –



The if Statements

Python provides if..elif..else control statements as a part of decision marking. It consists of three different blocks, which are if block, elif (short of else if) block and else block.

Example

Following is a simple example which makes use of if..elif..else. You can try to run this program using different marks and verify the result.

```
marks = 80
result = ""
if marks < 30:
    result = "Failed"
elif marks > 75:
    result = "Passed with distinction"
else:
    result = "Passed"

print(result)
```

This will produce following result:

```
Passed with distinction
```

The match Statement

Python supports Match-Case statement, which can also be used as a part of decision making. If a pattern matches the expression, the code under that case will execute.

Example

Following is a simple example which makes use of match statement.

```
def checkVowel(n):
    match n:
        case 'a': return "Vowel alphabet"
        case 'e': return "Vowel alphabet"
        case 'i': return "Vowel alphabet"
        case 'o': return "Vowel alphabet"
        case 'u': return "Vowel alphabet"
        case _: return "Simple alphabet"
print (checkVowel('a'))
print (checkVowel('m'))
print (checkVowel('o'))
```

This will produce following result:

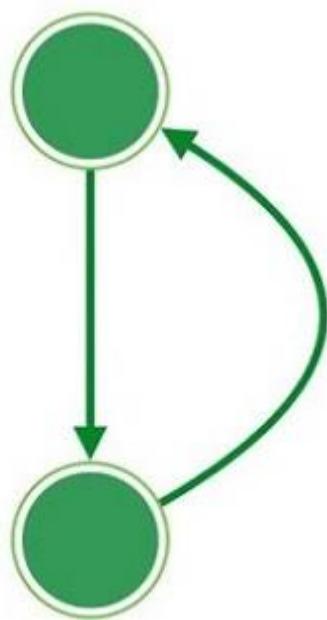
```
Vowel alphabet
```

Simple alphabet
Vowel alphabet

Loops or Iteration Statements

Most of the processes require a group of instructions to be repeatedly executed. In programming terminology, it is called a loop. Instead of the next step, if the flow is redirected towards any earlier step, it constitutes a loop.

The following diagram illustrates how the looping works –



If the control goes back unconditionally, it forms an infinite loop which is not desired as the rest of the code would never get executed.

In a conditional loop, the repeated iteration of block of statements goes on till a certain condition is met. Python supports a number of loops like for loop, while loop which we will study in next chapters.

The for Loop

The for loop iterates over the items of any sequence, such as a list, tuple or a string.

Example

Following is an example which makes use of For Loop to iterate through an array in Python:

<pre>words = ["one", "two", "three"] for x in words:</pre>
--

```
print(x)
```

This will produce following result:

```
one
two
three
```

The while Loop

The while loop repeatedly executes a target statement as long as a given boolean expression is true.

Example

Following is an example which makes use of While Loop to print first 5 numbers in Python:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

This will produce following result:

```
1
2
3
4
5
```

Jump Statements

The jump statements are used to jump on a specific statement by breaking the current flow of the program. In Python, there are two jump statements break and continue.

The break Statement

It terminates the current loop and resumes execution at the next statement.

Example

The following example demonstrates the use of break statement –

```
x = 0
```

```

while x < 10:
    print("x:", x)
    if x == 5:
        print("Breaking...")
        break
    x += 1

print("End")

```

This will produce following result:

```

x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
Breaking...
End

```

The continue Statement

It skips the execution of the program block and returns the control to the beginning of the current loop to start the next iteration.

Example

The following example demonstrates the use of continue statement –

```

for letter in "Python":
    # continue when letter is 'h'
    if letter == "h":
        continue
    print("Current Letter :", letter)

```

This will produce following result:

```

Current Letter : P
Current Letter : y
Current Letter : t

```

```
Current Letter : o
```

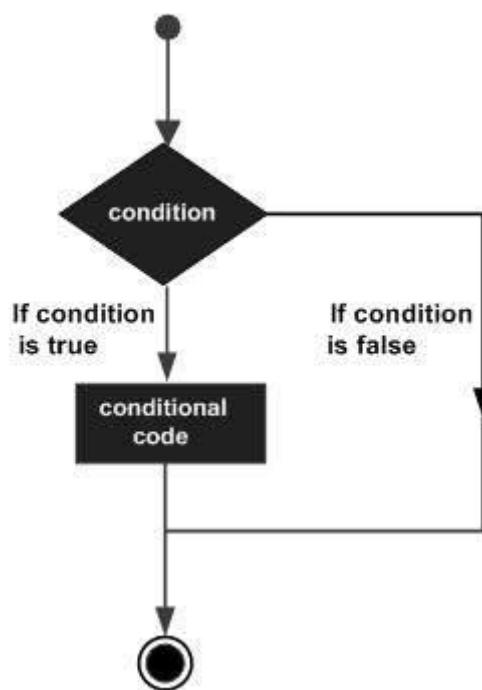
```
Current Letter : n
```

30. Python - Decision Making

Python's decision making functionality is in its keywords – if..elif...else. The if keyword requires a boolean expression, followed by colon (:) symbol. The colon (:) symbol starts an indented block. The statements with the same level of indentation are executed if the boolean expression in if statement is True. If the expression is not True (False), the interpreter bypasses the indented block and proceeds to execute statements at earlier indentation level.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages –



Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.

Types of Decision Making Statements in Python

Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	if statements

	An if statement consists of a boolean expression followed by one or more statements.
2	<u>if...else statements</u>
3	An if statement can be followed by an optional else statement , which executes when the boolean expression is FALSE. <u>nested if statements</u>

Let us go through each decision making briefly –

Single Statement Suites

If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.

Example

Here is an example of a one-line if clause –

```
var = 100
if ( var == 100 ) : print ("Value of expression is 100")
print ("Good bye!")
```

When the above code is executed, it produces the following result –

```
Value of expression is 100
Good bye!
```

if...else statement

In this decision making statement, if the **if condition** is true, then the statements within this block are executed, otherwise, the **else block** is executed.

The program will choose which block of code to execute based on whether the condition in the if statement is true or false.

Example

The following example shows the use of if...else statement.

```
var = 100
if ( var == 100 ):
    print ("Value of var is equal to 100")
```

```
else:
    print("Value of var is not equal to 100")
```

On running the above code, it will show the following output –

```
Value of var is equal to 100
```

Nested if statements

A nested if is another decision making statement in which one if statement resides inside another. It allows us to check multiple conditions sequentially.

Example

In this example, we will see the use of nested-if statement.

```
var = 100
if ( var == 100 ):
    print("The number is equal to 100")
    if var % 2 == 0:
        print("The number is even")
    else:
        print("The given number is odd")
elif var == 0:
    print("The given number is zero")
else:
    print("The given number is negative")
```

On executing the above code, it will display the below output –

```
The number is equal to 100
The number is even
```

31. Python - if Statement

Python If Statement

The if statement in Python evaluates whether a condition is true or false. It contains a logical expression that compares data, and a decision is made based on the result of the comparison.

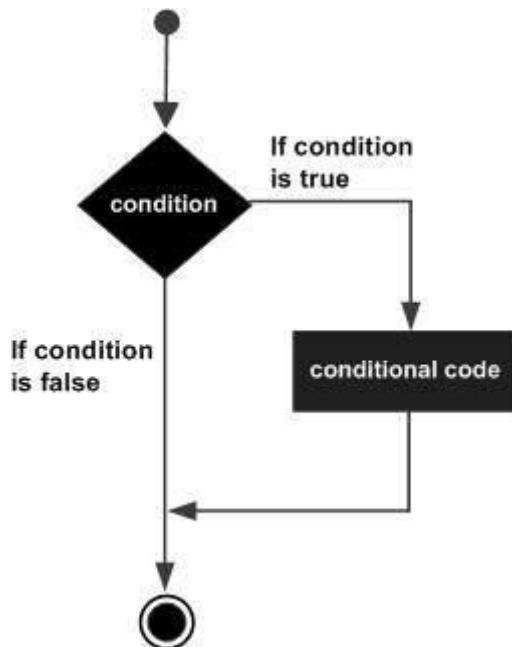
Syntax of the if Statement

```
if expression:  
    # statement(s) to be executed
```

If the boolean expression evaluates to TRUE, then the statement(s) inside the if block is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if block is executed.

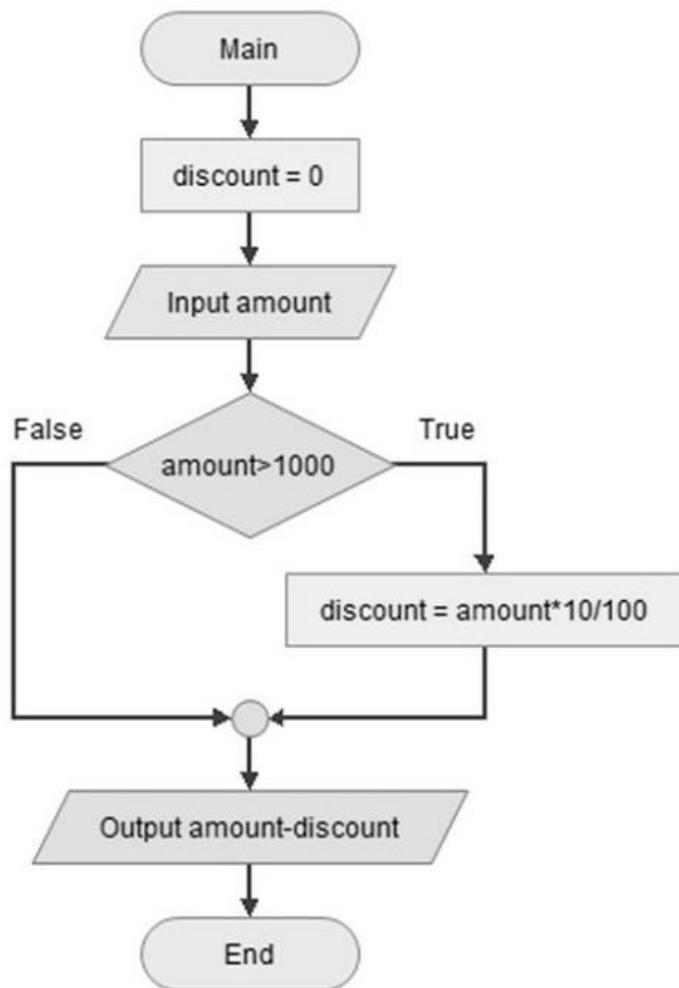
Flow Diagram (Flowchart) of the if Statement

The below diagram shows flowchart of the if statement –



Example of Python if Statement

Let us consider an example of a customer entitled to 10% discount if his purchase amount is > 1000; if not, then no discount is applicable. The following flowchart shows the whole decision making process –



First, set a discount variable to 0 and an amount variable to 1200. Then, use an if statement to check whether the amount is greater than 1000. If this condition is true, calculate the discount amount. If a discount is applicable, deduct it from the original amount.

Python code for the above flowchart can be written as follows –

```

discount = 0
amount = 1200

# Check he amount value
if amount > 1000:
    discount = amount * 10 / 100

print("amount = ", amount - discount)
  
```

Here the amount is 1200, hence discount 120 is deducted. On executing the code, you will get the following output –

```
amount = 1080.0
```

Change the variable amount to 800, and run the code again. This time, no discount is applicable. And, you will get the following output –

```
amount = 800
```

32. Python if-else Statement

Python if else Statement

The if-else statement in Python is used to execute a block of code when the condition in the if statement is true, and another block of code when the condition is false.

Syntax of if-else Statement

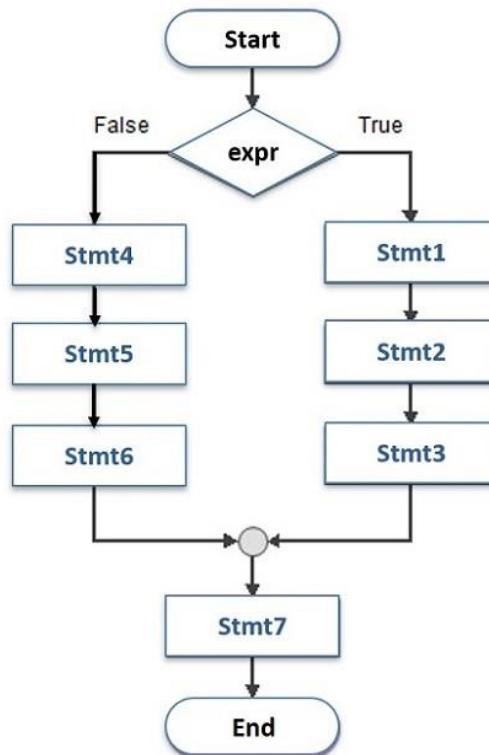
The syntax of an if-else statement in Python is as follows –

```
if boolean_expression:  
    # code block to be executed  
    # when boolean_expression is true  
else:  
    # code block to be executed  
    # when boolean_expression is false
```

If the boolean expression evaluates to TRUE, then the statement(s) inside the if block will be executed otherwise statements of the else block will be executed.

Flowchart of if-else Statement

This flowchart shows how if-else statement is used –



If the expr is True, block of stmt1, 2, 3 is executed then the default flow continues with stmt7. However, if the expr is False, block stmt4, 5, 6 runs then the default flow continues.

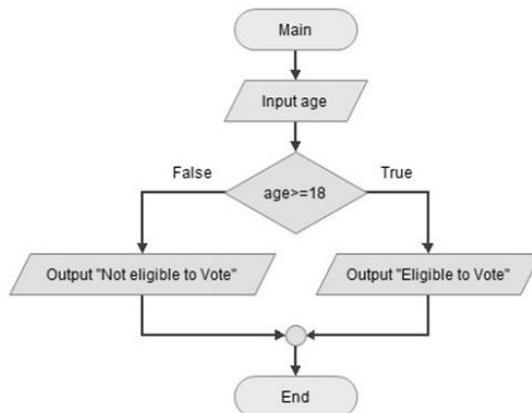
Python implementation of the above flowchart is as follows –

```

if expr==True:
    stmt1
    stmt2
    stmt3
else:
    stmt4
    stmt5
    stmt6
stmt7
  
```

Python if-else Statement Example

Let us understand the use of if-else statements with the following example. Here, variable age can take different values. If the expression age > 18 is true, then eligible to vote message will be displayed otherwise not eligible to vote message will be displayed. Following flowchart illustrates this logic –



Now, let's see the Python implementation the above flowchart.

```

age=25
print ("age: ", age)
if age >=18:
    print ("eligible to vote")
else:
    print ("not eligible to vote")
  
```

On executing this code, you will get the following output –

```

age: 25
eligible to vote
  
```

To test the else block, change the age to 12, and run the code again.

```

age: 12
not eligible to vote
  
```

Python if elif else Statement

The if elif else statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the else block, the elif block is also optional. However, a program can contain only one else block whereas there can be an arbitrary number of elif blocks following an if block.

Syntax of Python if elif else Statement

```
if expression1:  
    statement(s)  
elif expression2:  
    statement(s)  
elif expression3:  
    statement(s)  
else:  
    statement(s)
```

How if elif else Works?

The keyword elif is a short form of else if. It allows the logic to be arranged in a cascade of elif statements after the first if statement. If the first if statement evaluates to false, subsequent elif statements are evaluated one by one and comes out of the cascade if any one is satisfied.

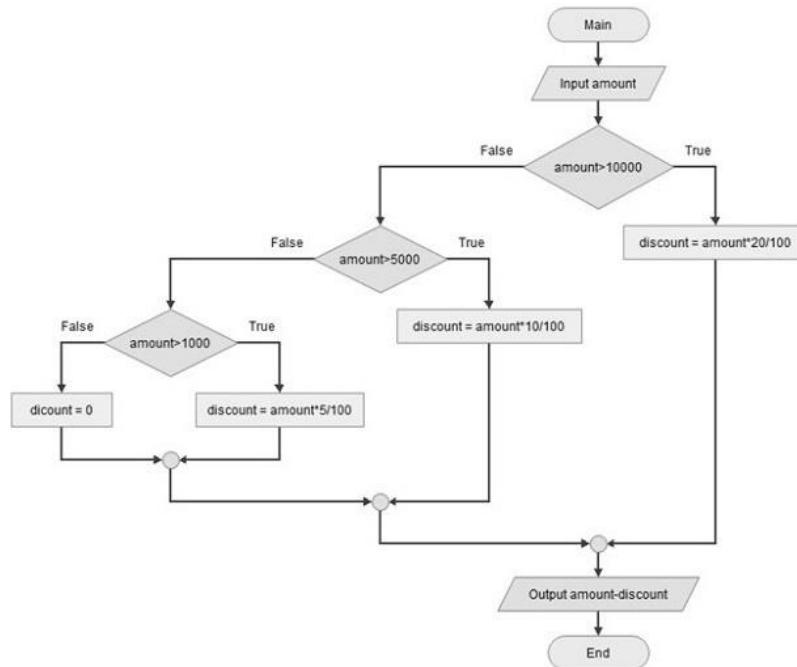
Last in the cascade is the else block which will come in picture when all preceding if/elif conditions fails.

Example

Suppose there are different slabs of discount on a purchase –

- 20% on amount exceeding 10000,
- 10% for amount between 5-10000,
- 5% if it is between 1 to 5000.
- no discount if amount<1000

The following flowchart illustrates these conditions –



We can write a Python code for the above logic with if-else statements –

```

amount = 2500
print('Amount = ',amount)
if amount > 10000:
    discount = amount * 20 / 100
else:
    if amount > 5000:
        discount = amount * 10 / 100
    else:
        if amount > 1000:
            discount = amount * 5 / 100
        else:
            discount = 0
print('Payable amount = ',amount - discount)
  
```

Set amount to test all possible conditions: 800, 2500, 7500 and 15000. The outputs will vary accordingly –

```

Amount: 800
Payable amount = 800
Amount: 2500
  
```

```
Payable amount = 2375.0
Amount: 7500
Payable amount = 6750.0
Amount: 15000
Payable amount = 12000.0
```

While the code will work perfectly fine, if you look at the increasing level of indentation at each if and else statement, it will become difficult to manage if there are still more conditions.

Python if elif else Statement Example

The elif statement makes the code easy to read and comprehend. Following is the Python code for the same logic with if elif else statements –

```
amount = 2500
print('Amount = ',amount)
if amount > 10000:
    discount = amount * 20 / 100
elif amount > 5000:
    discount = amount * 10 / 100
elif amount > 1000:
    discount = amount * 5 / 100
else:
    discount=0

print('Payable amount = ',amount - discount)
```

The output of the above code is as follows –

```
Amount: 2500
Payable amount = 2375.0
```

33. Python - Nested if Statement

Python supports nested if statements which means we can use a conditional if and if...else statement inside an existing if statement.

There may be a situation when you want to check for additional conditions after the initial one resolves to true. In such a situation, you can use the nested if construct.

Additionally, within a nested if construct, you can include an if...elif...else construct inside another if...elif...else construct.

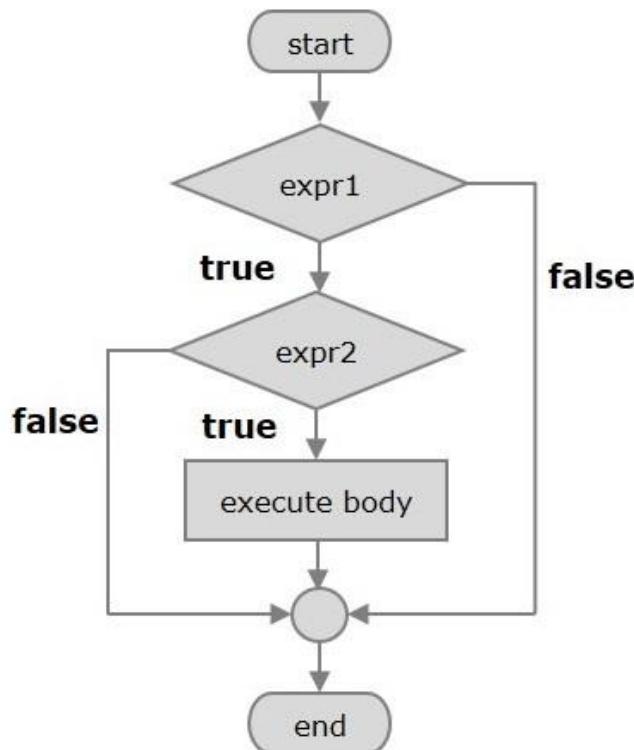
Syntax of Nested if Statement

The syntax of the nested if construct with else condition will be like this –

```
if boolean_expression1:  
    statement(s)  
    if boolean_expression2:  
        statement(s)
```

Flowchart of Nested if Statement

Following is the flowchart of Python nested if statement –



Example of Nested if Statement

The below example shows the working of nested if statements –

```

num = 36
print ("num = ", num)
if num % 2 == 0:
    if num % 3 == 0:
        print ("Divisible by 3 and 2")
print("....execution ends....")

```

When you run the above code, it will display the following result –

```

num = 36
Divisible by 3 and 2
....execution ends....

```

Nested if Statement with else Condition

As mentioned earlier, we can nest if-else statement within an if statement. If the if condition is true, the first if-else statement will be executed otherwise, statements inside the else block will be executed.

Syntax

The syntax of the nested if construct with else condition will be like this –

```

if expression1:
    statement(s)
    if expression2:
        statement(s)
    else:
        statement(s)
else:
    if expression3:
        statement(s)
    else:
        statement(s)

```

Example

Now let's take a Python code to understand how it works –

```

num=8
print ("num = ",num)
if num%2==0:
    if num%3==0:

```

```
    print ("Divisible by 3 and 2")
else:
    print ("divisible by 2 not divisible by 3")
else:
    if num%3==0:
        print ("divisible by 3 not divisible by 2")
    else:
        print ("not Divisible by 2 not divisible by 3")
```

When the above code is executed, it produces the following output –

```
num = 8
divisible by 2 not divisible by 3
num = 15
divisible by 3 not divisible by 2
num = 12
Divisible by 3 and 2
num = 5
not Divisible by 2 not divisible by 3
```

34. Python - Match-Case Statement

Python match-case Statement

A Python match-case statement takes an expression and compares its value to successive patterns given as one or more case blocks. Only the first pattern that matches gets executed. It is also possible to extract components (sequence elements or object attributes) from the value into variables.

With the release of Python 3.10, a pattern matching technique called match-case has been introduced, which is similar to the switch-case construct available in C/C++/Java etc. Its basic use is to compare a variable against one or more values. It is more similar to pattern matching in languages like Rust or Haskell than a switch statement in C or C++.

Syntax

The following is the syntax of match-case statement in Python -

```
match variable_name:  
    case 'pattern 1' : statement 1  
    case 'pattern 2' : statement 2  
    ...  
    case 'pattern n' : statement n
```

Example

The following code has a function named weekday(). It receives an integer argument, matches it with all possible weekday number values, and returns the corresponding name of day.

```
def weekday(n):  
    match n:  
        case 0: return "Monday"  
        case 1: return "Tuesday"  
        case 2: return "Wednesday"  
        case 3: return "Thursday"  
        case 4: return "Friday"  
        case 5: return "Saturday"  
        case 6: return "Sunday"  
        case _: return "Invalid day number"  
  
    print (weekday(3))  
    print (weekday(6))  
    print (weekday(7))
```

On executing, this code will produce the following output –

```
Thursday
Sunday
Invalid day number
```

The last case statement in the function has "_" as the value to compare. It serves as the wildcard case, and will be executed if all other cases are not true.

Combined Cases in Match Statement

Sometimes, there may be a situation where for more than one cases, a similar action has to be taken. For this, you can combine cases with the OR operator represented by "|" symbol.

Example

The code below shows how to combine cases in match statement. It defines a function named access() and has one string argument, representing the name of the user. For admin or manager user, the system grants full access; for Guest, the access is limited; and for the rest, there's no access.

```
def access(user):
    match user:
        case "admin" | "manager": return "Full access"
        case "Guest": return "Limited access"
        case _: return "No access"
print (access("manager"))
print (access("Guest"))
print (access("Ravi"))
```

On running the above code, it will show the following result –

```
Full access
Limited access
No access
```

List as the Argument in Match Case Statement

Since Python can match the expression against any literal, you can use a list as a case value. Moreover, for variable number of items in the list, they can be parsed to a sequence with "*" operator.

Example

In this code, we use list as argument in match case statement.

```
def greeting(details):
    match details:
        case [time, name]:
```

```

        return f'Good {time} {name}!'
case [time, *names]:
    msg=''
    for name in names:
        msg+=f'Good {time} {name}!\n'
    return msg

print (greeting(["Morning", "Ravi"]))
print (greeting(["Afternoon", "Guest"]))
print (greeting(["Evening", "Kajal", "Praveen", "Lata"]))

```

On executing, this code will produce the following output –

```

Good Morning Ravi!
Good Afternoon Guest!
Good Evening Kajal!
Good Evening Praveen!
Good Evening Lata!

```

Using "if" in "Case" Clause

Normally Python matches an expression against literal cases. However, it allows you to include if statement in the case clause for conditional computation of match variable.

Example

In the following example, the function argument is a list of amount and duration, and the interest is to be calculated for amount less than or more than 10000. The condition is included in the case clause.

```

def intr(details):
    match details:
        case [amt, duration] if amt<10000:
            return amt*10*duration/100
        case [amt, duration] if amt>=10000:
            return amt*15*duration/100
    print ("Interest = ", intr([5000,5]))
    print ("Interest = ", intr([15000,3]))

```

On executing, this code will produce the following output –

```

Interest = 2500.0
Interest = 6750.0

```

35. Python - Loops

Python Loops

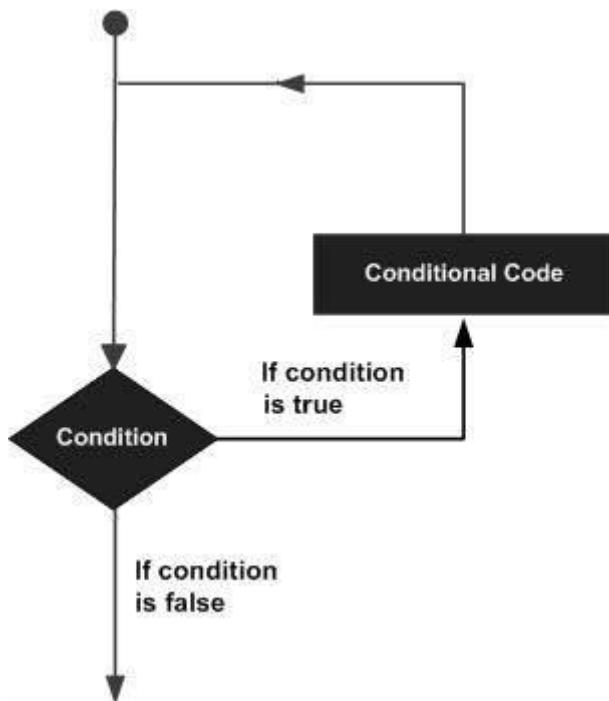
Python loops allow us to execute a statement or a group of statements multiple times.

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

Flowchart of a Loop

The following diagram illustrates a loop statement –



Types of Loops in Python

Python programming language provides following types of loops to handle looping requirements –

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop

	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	<p>nested loops</p> <p>You can use one or more loop inside any another while, for or do..while loop.</p>

Python Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	<p>break statement</p> <p>Terminates the loop statement and transfers execution to the statement immediately following the loop.</p>
2	<p>continue statement</p> <p>Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.</p>
3	<p>pass statement</p> <p>The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.</p>

36. Python - For Loops

The for loop in Python provides the ability to loop over the items of any sequence, such as a list, tuple or a string. It performs the same action on each item of the sequence. This loop starts with the for keyword, followed by a variable that represents the current item in the sequence.

The **in** keyword links the variable to the sequence you want to iterate over. A **colon (:) is used at the end of the loop header, and the indented block of code beneath it is executed once for each item in the sequence.**

Syntax of Python for Loop

```
for iterating_var in sequence:  
    statement(s)
```

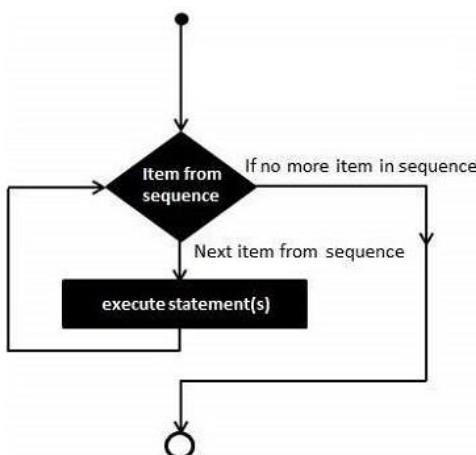
Here, the `iterating_var` is a variable to which the value of each sequence item will be assigned during each iteration. `Statements` represents the block of code that you want to execute repeatedly.

Before the loop starts, the sequence is evaluated. If it's a list, the expression list (if any) is evaluated first. Then, the first item (at index 0) in the sequence is assigned to `iterating_var` variable.

During each iteration, the block of statements is executed with the current value of `iterating_var`. After that, the next item in the sequence is assigned to `iterating_var`, and the loop continues until the entire sequence is exhausted.

Flowchart of Python for Loop

The following flow diagram illustrates the working of for loop –



Python for Loop with Strings

A string is a sequence of Unicode letters, each having a positional index. Since, it is a sequence, you can iterate over its characters using the for loop.

Example

The following example compares each character and displays if it is not a vowel ('a', 'e', 'i', 'o', 'u').

```
zen = ''

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
...
for char in zen:
    if char not in 'aeiou':
        print (char, end='')
```

On executing, this code will produce the following output –

```
Btfl s bttr thn gly.
Explct s bttr thn mplct.
Smpl s bttr thn cmplx.
Cmplx s bttr thn cmplctd.
```

Python for Loop with Tuples

Python's tuple object is also an indexed sequence, and hence you can traverse its items with a for loop.

Example

In the following example, the for loop traverses a tuple containing integers and returns the total of all numbers.

```
numbers = (34,54,67,21,78,97,45,44,80,19)
total = 0
for num in numbers:
    total += num
print ("Total =", total)
```

On running this code, it will produce the following output –

```
Total = 539
```

Python for Loop with Lists

Python's list object is also an indexed sequence, and hence you can iterate over its items using a for loop.

Example

In the following example, the for loop traverses a list containing integers and prints only those which are divisible by 2.

```
numbers = [34,54,67,21,78,97,45,44,80,19]
total = 0
for num in numbers:
    if num%2 == 0:
        print (num)
```

When you execute this code, it will show the following result –

```
34
54
78
44
80
```

Python for Loop with Range Objects

Python's built-in range() function returns an iterator object that streams a sequence of numbers. This object contains integers from start to stop, separated by step parameter. You can run a for loop with range as well.

Syntax

The range() function has the following syntax –

```
range(start, stop, step)
```

Where,

- **Start** – Starting value of the range. Optional. Default is 0
- **Stop** – The range goes upto stop-1
- **Step** – Integers in the range increment by the step value. The default is 1.

Example

In this example, we will see the use of range with for loop.

```
for num in range(5):
    print (num, end=' ')
print()
for num in range(10, 20):
    print (num, end=' ')
print()
for num in range(1, 10, 2):
    print (num, end=' ')
```

When you run the above code, it will produce the following output –

```
0 1 2 3 4
10 11 12 13 14 15 16 17 18 19
1 3 5 7 9
```

Python for Loop with Dictionaries

Unlike a list, tuple or a string, dictionary data type in Python is not a sequence, as the items do not have a positional index. However, traversing a dictionary is still possible with the for loop.

Example

Running a simple for loop over the dictionary object traverses the keys used in it.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers:
    print (x)
```

On executing, this code will produce the following output –

```
10
20
30
40
```

Once we are able to get the key, its associated value can be easily accessed either by using square brackets operator or with the get() method.

Example

The following example illustrates the above mentioned approach.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers:
    print (x,":",numbers[x])
```

It will produce the following output –

```
10 : Ten
20 : Twenty
30 : Thirty
40 : Forty
```

The items(), keys() and values() methods of dict class return the view objects dict_items, dict_keys and dict_values respectively. These objects are iterators, and hence we can run a for loop over them.

Example

The dict_items object is a list of key-value tuples over which a for loop can be run as follows –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers.items():
    print (x)
```

It will produce the following output –

```
(10, 'Ten')
(20, 'Twenty')
(30, 'Thirty')
(40, 'Forty')
```

Using else Statement with For Loop

Python supports else statements associated with a loop statement. However, the else statement is executed when the loop has exhausted iterating the list.

Example

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 to 20.

```
#For loop to iterate between 10 to 20
for num in range(10, 20):
    #For loop to iterate on the factors
    for i in range(2,num):
        #If statement to determine the first factor
        if num%i == 0:
            #To calculate the second factor
            j=num/i
            print ("%d equals %d * %d" % (num,i,j))
            #To move to the next number
            break
    else:
        print (num, "is a prime number")
        break
```

When the above code is executed, it produces the following result –

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
```

```
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

37. Python for-else Loops

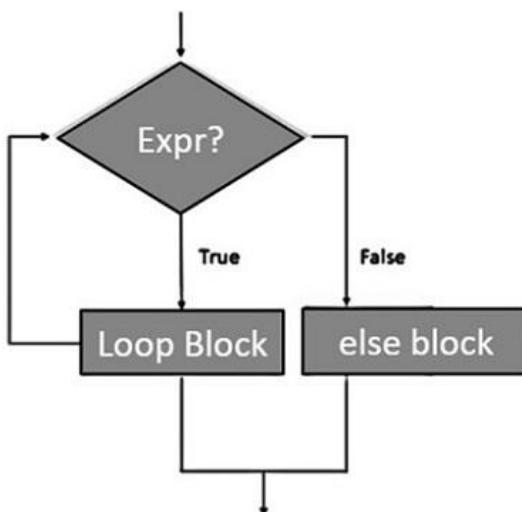
Python - For Else Loop

Python supports an optional else block to be associated with a for loop. If a else block is used with a for loop, it is executed only when the for loop terminates normally.

The for loop terminates normally when it completes all its iterations without encountering a break statement, which allows us to exit the loop when a certain condition is met.

Flowchart of For Else Loop

The following flowchart illustrates use of for-else loop –



Syntax of For Else Loop

Following is the syntax of for loop with optional else block –

```
for variable_name in iterable:  
    #stmts in the loop  
    .  
    .  
    .  
else:  
    #stmts in else clause  
    .  
    .
```

Example of For Else Loop

The following example illustrates the combination of an else statement with a for statement in Python. Till the count is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

```
for count in range(6):
    print ("Iteration no. {}".format(count))
else:
    print ("for loop over. Now in else block")
print ("End of for loop")
```

On executing, this code will produce the following output –

```
Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
for loop over. Now in else block
End of for loop
```

For-Else Construct without break statement

As mentioned earlier in this tutorial, the else block executes only when the loop terminates normally i.e. without using break statement.

In the following program, we use the for-else loop without break statement.

```
for i in ['T','P']:
    print(i)
else:
    # Loop else statement
    # there is no break statement in for loop, hence else part gets executed
    # directly
    print("ForLoop-else statement successfully executed")
```

On executing, the above program will generate the following output –

```
T
P
ForLoop-else statement successfully executed
```

For-Else Construct with break statement

In case of forceful termination (by using break statement) of the loop, else statement is overlooked by the interpreter and hence its execution is skipped.

Example

The following program shows how else conditions work in case of a break statement.

```
for i in ['T','P']:
    print(i)
    break
else:
    # Loop else statement
    # terminated after 1st iteration due to break statement in for loop
    print("Loop-else statement successfully executed")
```

On executing, the above program will generate the following output –

T

For-Else with break statement and if conditions

If we use for-else construct with break statement and if condition, the for loop will iterate over the iterators and within this loop, you can use an if block to check for a specific condition. If the loop completes without encountering a break statement, the code in the else block is executed.

Example

The following program shows how else conditions works in case of break statement and conditional statements.

```
# creating a function to check whether the list item is a positive
# or a negative number
def positive_or_negative():
    # traversing in a list
    for i in [5,6,7]:
        # checking whether the list element is greater than 0
        if i>=0:
            # printing positive number if it is greater than or equal to 0
            print ("Positive number")
        else:
            # Else printing Negative number and breaking the loop
            print ("Negative number")
            break
    # Else statement of the for loop
```

```
else:  
    # Statement inside the else block  
    print ("Loop-else Executed")  
# Calling the above-created function  
positive_or_negative()
```

On executing, the above program will generate the following output –

```
Positive number  
Positive number  
Positive number  
Loop-else Executed
```

38. Python - While Loops

Python while Loop

A while loop in Python programming language repeatedly executes a target statement as long as the specified boolean expression is true. This loop starts with while keyword followed by a boolean expression and colon symbol (:). Then, an indented block of statements starts.

Here, statement(s) may be a single statement or a block of statements with uniform indent. The condition may be any expression, and true is any non-zero value. As soon as the expression becomes false, the program control passes to the line immediately following the loop.

If it fails to turn false, the loop continues to run, and doesn't stop unless forcefully stopped. Such a loop is called infinite loop, which is undesired in a computer program.

Syntax of while Loop

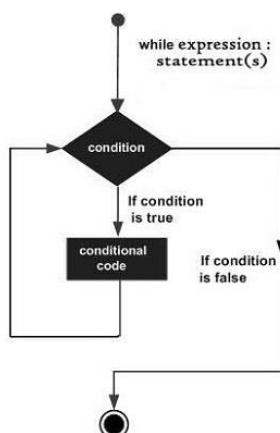
The syntax of a while loop in Python programming language is –

```
while expression:  
    statement(s)
```

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flowchart of While loop

The following flow diagram illustrates the while loop –



Example 1

The following example illustrates the working of while loop. Here, the iteration run till value of count will become 5.

```
count=0
```

```

while count<5:
    count+=1
    print ("Iteration no. {}".format(count))

print ("End of while loop")

```

On executing, this code will produce the following output –

```

Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
End of while loop

```

Example 2

Here is another example of using the while loop. For each iteration, the program asks for user input and keeps repeating till the user inputs a non-numeric string. The isnumeric() function returns true if input is an integer, false otherwise.

```

var = '0'
while var.isnumeric() == True:
    var = "test"
    if var.isnumeric() == True:
        print ("Your input", var)
print ("End of while loop")

```

On running the code, it will produce the following output –

```

enter a number..10
Your input 10
enter a number..100
Your input 100
enter a number..543
Your input 543
enter a number..qwer
End of while loop

```

Python Infinite while Loop

A loop becomes infinite if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

Example

Let's take an example to understand how the infinite loop works in Python –

```
var = 1

while var == 1 : # This constructs an infinite loop
    num = int(input("Enter a number :"))

    print ("You entered: ", num)
print ("Good bye!")
```

On executing, this code will produce the following output –

```
Enter a number :20
You entered: 20
Enter a number :29
You entered: 29
Enter a number :3
You entered: 3
Enter a number :11
You entered: 11
Enter a number :22
You entered: 22
Enter a number :Traceback (most recent call last):
  File "examples\test.py", line 5, in
    num = int(input("Enter a number :"))
KeyboardInterrupt
```

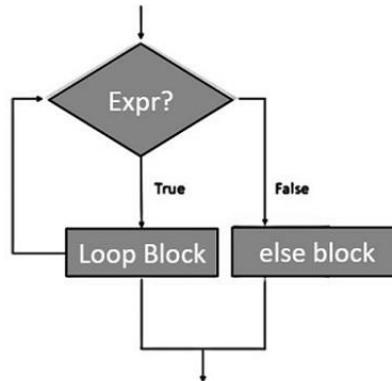
The above example goes in an infinite Loop and you need to use CTRL+C to exit the program.

Python while-else Loop

Python supports having an else statement associated with a while loop. If the else statement is used with a while loop, the else statement is executed when the condition becomes false before the control shifts to the main line of execution.

Flowchart of While loop with else Statement

The following flow diagram shows how to use else statement with while loop –

**Example**

The following example illustrates the combination of an else statement with a while statement. Till the count is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

```

count=0
while count<5:
    count+=1
    print ("Iteration no. {}".format(count))
else:
    print ("While loop over. Now in else block")
print ("End of while loop")
  
```

On running the above code, it will print the following output –

```

Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
While loop over. Now in else block
End of while loop
  
```

Single Statement Suites

Similar to the if statement syntax, if your while clause consists only of a single statement, it may be placed on the same line as the while header.

Example

The following example shows how to use one-line while clause.

```

flag = 0
while (flag): print ("Given flag is really true!")
  
```

```
print ("Good bye!")
```

When you run this code, it will display the following output –

```
Good bye!
```

Change the flag value to "1" and try the above program. If you do so, it goes into infinite loop and you need to press CTRL+C keys to exit.

39. Python - break Statement

Python break Statement

Python break statement is used to terminate the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for Python break statement is when some external condition is triggered requiring a sudden exit from a loop. The break statement can be used in both Python while and for loops.

If you are using nested loops in Python, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

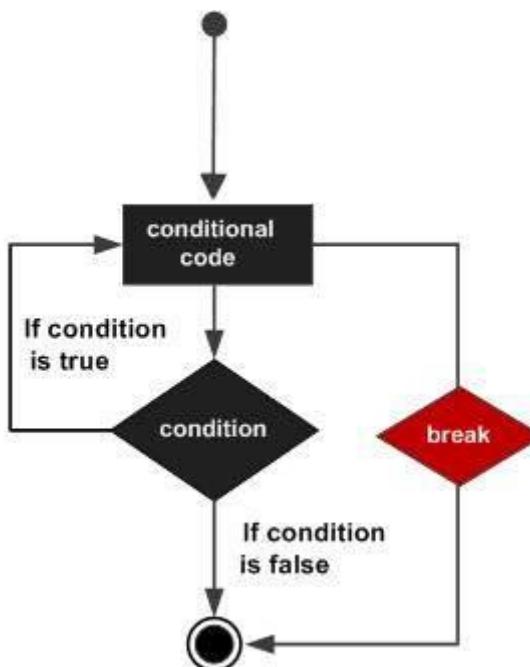
Syntax of break Statement

The syntax for a break statement in Python is as follows –

```
looping statement:  
    condition check:  
        break
```

Flow Diagram of break Statement

Following is the flowchart of the break statement –



break Statement with for loop

If we use break statement inside a for loop, it interrupts the normal flow of program and exit the loop before completing the iteration.

Example

In this example, we will see the working of break statement in for loop.

```
for letter in 'Python':
    if letter == 'h':
        break
    print ("Current Letter :", letter)
print ("Good bye!")
```

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
Good bye!
break Statement with while loop
```

Similar to the for loop, we can use the break statement to skip the code inside while loop after the specified condition becomes TRUE.

Example

The code below shows how to use break statement with while loop.

```
var = 10
while var > 0:
    print ('Current variable value :', var)
    var = var -1
    if var == 5:
        break

print ("Good bye!")
```

On executing the above code, it produces the following result –

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

break Statement with Nested Loops

In nested loops, one loop is defined inside another. The loop that enclose another loop (i.e. inner loop) is called as outer loop.

When we use a break statement with nested loops, it behaves as follows –

When break statement is used inside the inner loop, only the inner loop will be skipped and the program will continue executing statements after the inner loop

And, when the break statement is used in the outer loop, both the outer and inner loops will be skipped and the program will continue executing statements immediate to the outer loop.

Example

The following program demonstrates the use of break in a for loop iterating over a list. Here, the specified number will be searched in the list. If it is found, then the loop terminates with the "found" message.

```
no = 33

numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:

    if num == no:

        print ('number found in list')

        break

    else:

        print ('number not found in list')
```

The above program will produce the following output –

```
number found in list
```

40. Python - Continue Statement

Python continue Statement

Python continue statement is used to skip the execution of the program block and returns the control to the beginning of the current loop to start the next iteration. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

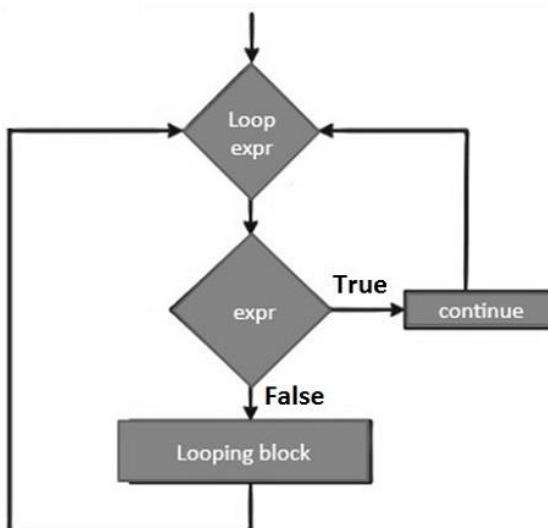
The continue statement is just the opposite to that of break. It skips the remaining statements in the current loop and starts the next iteration.

Syntax of continue Statement

```
looping statement:  
    condition check:  
        continue
```

Flow Diagram of continue Statement

The flow diagram of the continue statement looks like this –



Python continue Statement with for Loop

In Python, the continue statement is allowed to be used with a for loop. Inside the for loop, you should include an **if statement** to check for a specific condition. If the condition becomes TRUE, the continue statement will skip the current iteration and proceed with the next iteration of the loop.

Example

Let's see an example to understand how the continue statement works in for loop.

```

for letter in 'Python':
    if letter == 'h':
        continue
    print ('Current Letter :', letter)
print ("Good bye!")

```

When the above code is executed, it produces the following output –

```

Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Good bye!

```

Python continue Statement with while Loop

Python continue statement is used with 'for' loops as well as 'while' loops to skip the execution of the current iteration and transfer the program's control to the next iteration.

Example: Checking Prime Factors

Following code uses continue statement to find the prime factors of a given number. To find prime factors, we need to successively divide the given number starting with 2, increment the divisor and continue the same process till the input reduces to 1.

```

num = 60
print ("Prime factors for: ", num)
d=2
while num > 1:
    if num%d==0:
        print (d)
        num=num/d
        continue
    d=d+1

```

On executing, this code will produce the following output –

```

Prime factors for: 60
2
2
3
5

```

Assign different value (say 75) to num in the above program and test the result for its prime factors.

```
Prime factors for: 75
```

```
3
```

```
5
```

```
5
```

41. Python - pass Statement

Python pass Statement

Python pass statement is used when a statement is required syntactically but you do not want any command or code to execute. It is a null which means nothing happens when it executes. This is also useful in places where piece of code will be added later, but a placeholder is required to ensure the program runs without errors.

For instance, in a function or class definition where the implementation is yet to be written, pass statement can be used to avoid the SyntaxError. Additionally, it can also serve as a placeholder in control flow statements like for and while loops.

Syntax of pass Statement

Following is the syntax of Python pass statement –

```
pass
```

Example of pass Statement

The following code shows how you can use the pass statement in Python –

```
for letter in 'Python':  
    if letter == 'h':  
        pass  
        print ('This is pass block')  
        print ('Current Letter :', letter)  
    print ("Good bye!")
```

When the above code is executed, it produces the following output –

```
Current Letter : P  
Current Letter : y  
Current Letter : t  
This is pass block  
Current Letter : h  
Current Letter : o  
Current Letter : n  
Good bye!
```

Dumpy Infinite Loop with pass Statement

This is simple enough to create an infinite loop using pass statement in Python.

Example

If you want to code an infinite loop that does nothing each time through, do it as shown below –

```
while True: pass  
# Type Ctrl-C to stop
```

Because the body of the loop is just an empty statement, Python gets stuck in this loop.

Using Ellipses (...) as pass Statement Alternative

Python 3.X allows ellipses (coded as three consecutive dots ...) to be used in place of pass statement. Both serve as placeholders for code that are going to be written later.

Example

For example if we create a function which does not do anything especially for code to be filled in later, then we can make use of ...

```
def func1():  
    # Alternative to pass  
    ...  
  
    # Works on same line too  
def func2(): ...  
    # Does nothing if called  
func1()  
func2()
```

42. Python - Nested Loops

In Python, when you write one or more loops within a loop statement that is known as a nested loop. The main loop is considered as outer loop and loop(s) inside the outer loop are known as inner loops.

The Python programming language allows the usage of one loop inside another loop. A loop is a code block that executes specific instructions repeatedly. There are two types of loops, namely for and while, using which we can create nested loops.

You can put any type of Loop inside of any other type of Loop. For example, a for Loop can be inside a while Loop or vice versa.

Python Nested for Loop

The for loop with one or more inner for loops is called nested for loop. A for loop is used to loop over the items of any sequence, such as a list, tuple or a string and performs the same action on each item of the sequence.

Python Nested for Loop Syntax

The syntax for a Python nested for loop statement in Python programming language is as follows –

```
for iterating_var in sequence:  
    for iterating_var in sequence:  
        statements(s)  
    statements(s)
```

Python Nested for Loop Example

The following program uses a nested for loop to iterate over months and days lists.

```
months = ["jan", "feb", "mar"]  
days = ["sun", "mon", "tue"]  
  
for x in months:  
    for y in days:  
        print(x, y)  
  
print("Good bye!")
```

When the above code is executed, it produces following result –

```
jan sun  
jan mon
```

```

jan tue
feb sun
feb mon
feb tue
mar sun
mar mon
mar tue
Good bye!

```

Python Nested while Loop

The while loop having one or more inner while loops are nested while loop. A while loop is used to repeat a block of code for an unknown number of times until the specified boolean expression becomes TRUE.

Python Nested while Loop Syntax

The syntax for a nested while loop statement in Python programming language is as follows –

```

while expression:
    while expression:
        statement(s)
    statement(s)

```

Python Nested while Loop Example

The following program uses a nested while loop to find the prime numbers from 2 to 100 –

```

i = 2
while(i < 25):
    j = 2
    while(j <= (i/j)):
        if not(i%j): break
        j = j + 1
    if (j > i/j) : print (i, " is prime")
    i = i + 1

print ("Good bye!")

```

On executing, the above code produces following result –

```

2 is prime
3 is prime

```

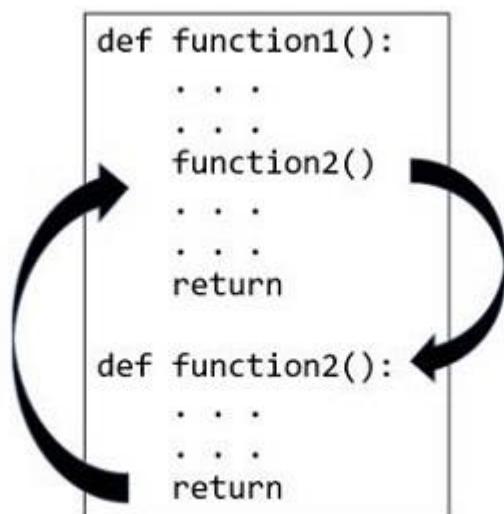
```
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
Good bye!
```

Python Functions & Modules

43. Python - Functions

A Python function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

A top-to-down approach towards building the processing logic involves defining blocks of independent reusable functions. A Python function may be invoked from any other function by passing required data (called parameters or arguments). The called function returns its result back to the calling environment.



Types of Python Functions

Python provides the following types of functions –

Sr.No	Type & Description
1	<u>Built-in functions</u> Python's standard library includes number of built-in functions. Some of Python's built-in functions are <code>print()</code> , <code>int()</code> , <code>len()</code> , <code>sum()</code> , etc. These functions are always available, as they are loaded into computer's memory as soon as you start Python interpreter.
2	<u>Functions defined in built-in modules</u> The standard library also bundles a number of modules. Each module defines a group of functions. These functions are not readily available. You need to import them into the memory from their respective modules.
3	<u>User-defined functions</u> In addition to the built-in functions and functions in the built-in modules, you can also create your own functions. These functions are called user-defined functions.

Defining a Python Function

You can define custom functions to provide the required functionality. Here are simple rules to define a function in Python –

- Function blocks begin with the keyword def followed by the function name and parentheses ().
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement; the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax to Define a Python Function

```
def function_name( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

By default, parameters have a positional behavior and you need to add the values in the same order that they were defined.

Once the function is defined, you can execute it by calling it from another function or directly from the Python prompt.

Example to Define a Python Function

The following example shows how to define a function greetings(). The bracket is empty so there aren't any parameters. Here, the first line is a docstring and the function block ends with return statement.

```
def greetings():
    "This is docstring of greetings function"
    print ("Hello World")
    return
```

When this function is called, Hello world message will be printed.

Calling a Python Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. Once the basic structure of a function is finalized, you can call it by using the function name itself. If the function requires any parameters, they should be passed within parentheses. If the function doesn't require any parameters, the parentheses should be left empty.

Example to Call a Python Function

Following is the example to call printme() function –

```
# Function definition is here
```

```

def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;

# Now you can call the function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")

```

When the above code is executed, it produces the following output –

```

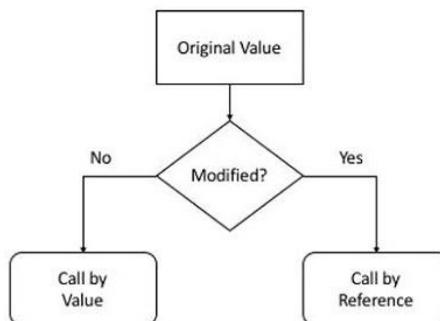
I'm first call to user defined function!
Again second call to the same function

```

Pass by Reference vs Value

In programming languages like C and C++, there are two main ways to pass variables to a function, which are Call by Value and Call by Reference (also known as pass by reference and pass by value). However, the way we pass variables to functions in Python differs from others.

- **call by value** – When a variable is passed to a function while calling, the value of actual arguments is copied to the variables representing the formal arguments. Thus, any changes in formal arguments does not get reflected in the actual argument. This way of passing variable is known as call by value.
- **call by reference** – In this way of passing variable, a reference to the object in memory is passed. Both the formal arguments and the actual arguments (variables in the calling code) refer to the same object. Hence, any changes in formal arguments does get reflected in the actual argument.



Python uses pass by reference mechanism. As variable in Python is a label or reference to the object in the memory, both the variables used as actual argument as well as formal arguments really refer to the same object in the memory. We can verify this fact by checking the `id()` of the passed variable before and after passing.

Example

In the following example, we are checking the `id()` of a variable.

```
def testfunction(arg):
    print ("ID inside the function:", id(arg))

var = "Hello"
print ("ID before passing:", id(var))
testfunction(var)
```

If the above code is executed, the `id()` before passing and inside the function will be displayed.

```
ID before passing: 1996838294128
ID inside the function: 1996838294128
```

The behavior also depends on whether the passed object is mutable or immutable. Python numeric object is immutable. When a numeric object is passed, and the function changes the value of the formal argument, it actually creates a new object in the memory, leaving the original variable unchanged.

Example

The following example shows how an immutable object behaves when it is passed to a function.

```
def testfunction(arg):
    print ("ID inside the function:", id(arg))
    arg = arg + 1
    print ("new object after increment", arg, id(arg))

var=10
print ("ID before passing:", id(var))
testfunction(var)
print ("value after function call", var)
```

It will produce the following output –

```
ID before passing: 140719550297160
ID inside the function: 140719550297160
new object after increment 11 140719550297192
value after function call 10
```

Let us now pass a mutable object (such as a list or dictionary) to a function. It is also passed by reference, as the `id()` of list before and after passing is same. However, if we modify the list inside the function, its global representation also reflects the change.

Example

Here we pass a list, append a new item, and see the contents of original list object, which we will find has changed.

```

def testfunction(arg):
    print ("Inside function:",arg)
    print ("ID inside the function:", id(arg))
    arg.append(100)

var=[10, 20, 30, 40]
print ("ID before passing:", id(var))
testfunction(var)
print ("list after function call", var)

```

It will produce the following output –

```

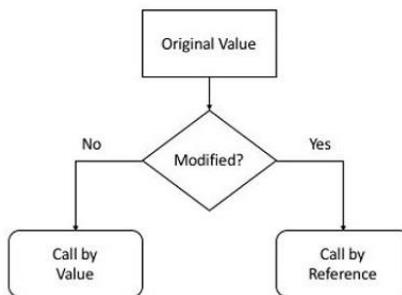
ID before passing: 2716006372544
Inside function: [10, 20, 30, 40]
ID inside the function: 2716006372544
list after function call [10, 20, 30, 40, 100]

```

Python Function Arguments

Function arguments are the values or variables passed into a function when it is called. The behavior of a function often depends on the arguments passed to it.

While defining a function, you specify a list of variables (known as formal parameters) within the parentheses. These parameters act as placeholders for the data that will be passed to the function when it is called. When the function is called, value to each of the formal arguments must be provided. Those are called actual arguments.



Example

Let's modify greetings function and use **name** as an argument. A string passed to the function as actual argument becomes name variable inside the function.

```

def greetings(name):
    "This is docstring of greetings function"
    print ("Hello {}".format(name))
    return

```

```
greetings("Samay")
greetings("Pratima")
greetings("Steven")
```

This code will produce the following output –

```
Hello Samay
Hello Pratima
Hello Steven
```

Types of Python Function Arguments

Based on how the arguments are declared while defining a Python function, they are classified into the following categories –

- Positional or Required Arguments
- Keyword Arguments
- Default Arguments
- Positional-only Arguments
- Keyword-only arguments
- Arbitrary or Variable-length Arguments

Positional or Required Arguments

Required arguments are the ones passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition, otherwise the code gives a syntax error.

Example

In the code below, we call the function printme() without any parameters which will give error.

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result –

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
```

```
printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example 1

The following example shows how to use keyword arguments in Python.

```
# Function definition is here

def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;

# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result –

```
My string
```

Example 2

The following example gives more clear picture. Note that the order of parameters does not matter.

```
# Function definition is here

def printinfo( name, age ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result –

```
Name: miki
Age 50
```

Default Arguments

A default argument assumes a default value if a value is not provided in the function call for that argument.

Example

The following example gives an idea on default arguments, it prints default age if it is not passed –

```
# Function definition is here

def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result –

```
Name: miki
Age 50
Name: miki
Age 35
```

Positional-only arguments

Those arguments that can only be specified by their position in the function call is called as Positional-only arguments. They are defined by placing a "/" in the function's parameter list after all positional-only parameters. This feature was introduced with the release of Python 3.8.

The benefit of using this type of argument is that it ensures the functions are called with the correct arguments in the correct order. The positional-only arguments should be passed to a function as positional arguments, not keyword arguments.

Example

In the following example, we have defined two positional-only arguments namely "x" and "y". This method should be called with positional arguments in the order in which the arguments are declared, otherwise, we will get an error.

```
def posFun(x, y, /, z):
    print(x + y + z)

print("Evaluating positional-only arguments: ")
```

```
posFun(33, 22, z=11)
```

It will produce the following output –

```
Evaluating positional-only arguments:
```

```
66
```

Keyword-only arguments

Those arguments that must be specified by their name while calling the function is known as Keyword-only arguments. They are defined by placing an asterisk ("*") in the function's parameter list before any keyword-only parameters. This type of argument can only be passed to a function as a keyword argument, not a positional argument.

Example

In the code below, we have defined a function with three keyword-only arguments. To call this method, we need to pass keyword arguments, otherwise, we will encounter an error.

```
def posFun(*, num1, num2, num3):
    print(num1 * num2 * num3)

print("Evaluating keyword-only arguments: ")
posFun(num1=6, num2=8, num3=5)
```

It will produce the following output –

```
Evaluating keyword-only arguments:
```

```
240
```

Arbitrary or Variable-length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example

Following is a simple example of Python variable-length arguments.

```
# Function definition is here

def printinfo( arg1, *vartuple ):

    "This prints a variable passed arguments"
    print ("Output is: ")
    print (arg1)
    for var in vartuple:
        print (var)

    return;

# Now you can call printinfo function

printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result –

```
Output is:
10
Output is:
70
60
50
```

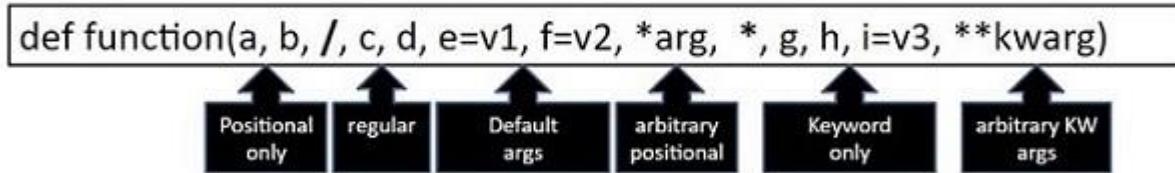
In the next few chapters, we will discuss these function arguments at length.

Order of Python Function Arguments

A function can have arguments of any of the types defined above. However, the arguments must be declared in the following order –

- The argument list begins with the positional-only args, followed by the slash (/) symbol.
- It is followed by regular positional args that may or may not be called as keyword arguments.
- Then there may be one or more args with default values.
- Next, arbitrary positional arguments represented by a variable prefixed with single asterisk, that is treated as tuple. It is the next.
- If the function has any keyword-only arguments, put an asterisk before their names start. Some of the keyword-only arguments may have a default value.
- Last in the bracket is argument with two asterisks ** to accept arbitrary number of keyword arguments.

The following diagram shows the order of formal arguments –



Python Function with Return Value

The return keyword as the last statement in function definition indicates end of function block, and the program flow goes back to the calling function. Although reduced indent after the last statement in the block also implies return but using explicit return is a good practice.

Along with the flow control, the function can also return value of an expression to the calling function. The value of returned expression can be stored in a variable for further processing.

Example

Let us define the add() function. It adds the two values passed to it and returns the addition. The returned value is stored in a variable called result.

```

def add(x,y):
    z=x+y
    return z
a=10
b=20
result = add(a,b)
print ("a = {} b = {} a+b = {}".format(a, b, result))
  
```

It will produce the following output –

```
a = 10 b = 20 a+b = 30
```

The Anonymous Functions

The functions are called anonymous when they are not declared in the standard manner by using the def keyword. Instead, they are defined using the lambda keyword.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda is a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of lambda functions contains only a single statement, which is as follows –

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Example

Following is the example to show how lambda form of function works –

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

When the above code is executed, it produces the following result –

```
Value of total : 30
Value of total : 40
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python –

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Example

Following is a simple example of local and global scope –

```
total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):
    # Add both the parameters and return them.

    total = arg1 + arg2; # Here total is local variable.

    print ("Inside the function local total : ", total)
```

```
return total;

# Now you can call sum function
sum( 10, 20 );
print ("Outside the function global total : ", total)
```

When the above code is executed, it produces the following result –

```
Inside the function local total : 30
Outside the function global total : 0
```

44. Python - Default Arguments

Python Default Arguments

Python allows a function definition with default value assigned to one or more formal arguments. Python uses the default value for such an argument to which no value has been assigned. If any value is passed, the default value is overridden with the actual value passed.

Default arguments in Python are the function arguments that will be used if no arguments are passed to the function call.

Example of Default Arguments

The following example shows use of Python default arguments. Here, the second call to the function will not pass value to "city" argument, hence its default value "Hyderabad" will be used.

```
# Function definition

def showinfo( name, city = "Hyderabad" ):

    "This prints a passed info into this function"
    print ("Name:", name)
    print ("City:", city)
    return

# Now call showinfo function
showinfo(name = "Ansh", city = "Delhi")
showinfo(name = "Shrey")
```

It will produce the following output –

```
Name: Ansh
City: Delhi
Name: Shrey
City: Hyderabad
```

Example: Calling Function Without Keyword Arguments

Let us look at another example that assigns default value to a function argument. The function percent() has a default argument named "maxmarks" which is set to 200. Hence, we can omit the value of third argument while calling the function.

```
# function definition
```

```

def percent(phy, maths, maxmarks=200):
    val = (phy + maths) * 100/maxmarks
    return val

phy = 60
maths = 70
# function calling with default argument
result = percent(phy, maths)
print ("percentage:", result)

phy = 40
maths = 46
result = percent(phy, maths, 100)
print ("percentage:", result)

```

On executing, this code will produce the following output –

```

percentage: 65.0
percentage: 86.0

```

Mutable Objects as Default Argument

Python evaluates default arguments once when the function is defined, not each time the function is called. Therefore, if you use a mutable default argument and modify it within the given function, the same values are referenced in the subsequent function calls.

Those Python objects that can be changed after creation are called as mutable objects.

Example

The code below explains how to use mutable objects as default argument in Python.

```

def fcn(nums, numericlist = []):
    numericlist.append(nums + 1)
    print(numericlist)

# function calls
fcn(66)
fcn(68)
fcn(70)

```

On executing the above code, it will produce the following output –

```
[67]  
[67, 69]  
[67, 69, 71]
```

45. Python - Keyword Arguments

Keyword Arguments

Python allows to pass function arguments in the form of keywords which are also called **named arguments**. Variables in the function definition are used as keywords. When the function is called, you can explicitly mention the name and its value.

Calling Function with Keyword Arguments

The following example demonstrates keyword arguments in Python. In the second function call, we have used keyword arguments.

```
# Function definition is here

def printinfo( name, age ):

    "This prints a passed info into this function"

    print ("Name: ", name)
    print ("Age ", age)

    return

# Now you can call printinfo function

# by positional arguments

printinfo ("Naveen", 29)

# by keyword arguments

printinfo(name="miki", age = 30)
```

It will produce the following output –

```
Name: Naveen
Age 29
Name: miki
Age 30
```

Order of Keyword Arguments

By default, the function assigns the values to arguments in the order of appearance. However, while using keyword arguments, it is not necessary to follow the order of formal arguments in function definition. Use of keyword arguments is optional. You can use mixed calling. You can pass values to some arguments without keywords, and for others with keyword.

Example

Let us try to understand with the help of following function definition –

```
def division(num, den):
    quotient = num/den
    print ("num:{} den:{} quotient:{}".format(num, den, quotient))

division(10,5)
division(5,10)
```

Since the values are assigned as per the position, the output is as follows –

```
num:10 den:5 quotient:2.0
num:5 den:10 quotient:0.5
```

Example

Instead of passing the values with positional arguments, let us call the function with keyword arguments –

```
def division(num, den):
    quotient = num/den
    print ("num:{} den:{} quotient:{}".format(num, den, quotient))

division(num=10, den=5)
division(den=5, num=10)
```

Unlike positional arguments, the order of keyword arguments does not matter. Hence, it will produce the following output –

```
num:10 den:5 quotient:2.0
num:10 den:5 quotient:2.0
```

However, the positional arguments must be before the keyword arguments while using mixed calling.

Example

Try to call the division() function with the keyword arguments as well as positional arguments.

```
def division(num, den):
    quotient = num/den
    print ("num:{} den:{} quotient:{}".format(num, den, quotient))

division(num = 5, 10)
```

As the Positional argument cannot appear after keyword arguments, Python raises the following error message –

```
division(num=5, 10)
^
SyntaxError: non-keyword arg after keyword arg
```

46. Python - Keyword-Only Arguments

Keyword-Only Arguments

You can use the variables in formal argument list as keywords to pass value. Use of keyword arguments is optional. But, you can force the function to accept arguments by keyword only. You should put an asterisk (*) before the keyword-only arguments list.

Let us say we have a function with three arguments, out of which we want second and third arguments to be keyword-only. For that, put * after the first argument.

Example of Keyword-Only Arguments

The built-in print() function is an example of keyword-only arguments. You can give list of expressions to be printed in the parentheses. The printed values are separated by a white space by default. You can specify any other separation character instead using "sep" argument.

```
print ("Hello", "World", sep="-")
```

It will print –

```
Hello-World
```

Example: Using "sep" as non-keyword Argument

The sep argument of the print() function is keyword-only. Try using it as non-keyword argument.

```
print ("Hello", "World", "-")
```

You'll get different output, not as desired –

```
Hello World -
```

Using Keyword-Only argument in User-Defined Method

To make an argument keyword-only, put the asterisk (*) before it while creating the user-defined function.

Those Python functions that are defined by us within a given class to perform certain actions are called as user-defined function. They are not predefined by Python.

Example

In the following user defined function "intr()" the "rate" argument is keyword-only. To call this function, the value for rate must be passed by keyword.

```
def intr(amt,*, rate):
    val = amt*rate/100
    return val
```

```
interest = intr(1000, rate=10)
print(interest)
100.0
```

However, if you try to use the default positional way of calling the above function, you will encounter an error.

Example

The code below shows it is not possible to use positional arguments when keyword-only arguments are required.

```
def intr(amt, *, rate):
    val = amt * rate / 100
    return val

interest = intr(1000, 10)
print(interest)
```

On executing, this code will show the following result –

```
interest = intr(1000, 10)
               ^
TypeError: intr() takes 1 positional argument but 2 were given
```

47. Python - Positional Arguments

Positional Arguments

The list of variables declared in the parentheses at the time of defining a function are the formal arguments. These arguments are also known as positional arguments. A function may be defined with any number of formal arguments.

While calling a function –

- All the arguments are required.
- The number of actual arguments must be equal to the number of formal arguments.
- They Pick up values in the order of definition.
- The type of arguments must match.
- Names of formal and actual arguments need not be same.

Positional Arguments Examples

Let's discuss some examples of Positional arguments –

Example 1

The following example shows the use of positional argument.

```
def add(x,y):  
    z = x+y  
    print ("x={} y={} x+y={}".format(x,y,z))  
  
a = 10  
b = 20  
  
add(a, b)
```

It will produce the following output –

```
x=10 y=20 x+y=30
```

Here, the `add()` function has two formal arguments, both are numeric. When integers 10 and 20 passed to it. The variable "a" takes 10 and "b" takes 20, in the order of declaration. The `add()` function displays the addition.

Example 2

Python also raises error when the number of arguments don't match. If you give only one argument and check the result you can see an error.

```
def add(x,y):  
    z=x+y  
    print (z)  
  
a=10;  
  
add(a)
```

The error generated will be as shown below –

```
TypeError: add() missing 1 required positional argument: 'y'
```

Example 3

Similarly, if you pass more than the number of formal arguments an error will be generated stating the same –

```
def add(x,y):
    z=x+y
    print ("x={} y={} x+y={}".format(x,y,z))
add(10, 20, 30)
```

Following is the output –

```
TypeError: add() takes 2 positional arguments but 3 were given
```

Example 4

Data type of corresponding actual and formal arguments must match. Change a to a string value and see the result.

```
def add(x,y):
    z=x+y
    print (z)
a="Hello"
b=20
add(a,b)
```

It will produce the following error –

```
z=x+y
^~^
TypeError: can only concatenate str (not "int") to str
```

Difference between Positional and Keyword argument

The below table explains the difference between positional and keyword argument –

Positional Argument	Keyword Argument
Only the names of arguments are used to pass data to the given function.	Keyword arguments are passed to a function in name=value form.
Arguments are passed in the order defined in function declaration.	While passing arguments, their order can be changed.
Syntax: function(param1, param2,...)	Syntax: function(param1 = value1,...)

48. Python - Positional-Only Arguments

Positional Only Arguments

It is possible in Python to define a function in which one or more arguments can not accept their value with keywords. Such arguments are called positional-only arguments.

To make an argument positional-only, use the forward slash (/) symbol. All the arguments before this symbol will be treated as positional-only.

Python's built-in `input()` function is an example of positional-only arguments. The syntax of `input` function is –

```
input(prompt = "")
```

Prompt is an explanatory string for the benefit of the user. However, you cannot use the `prompt` keyword inside the parentheses.

Example

In this example, we are using `prompt` keyword, which will lead to error.

```
name = input(prompt="Enter your name ")
```

On executing, this code will show the following error message –<>

```
name = input (prompt="Enter your name ")
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: input() takes no keyword arguments
```

Positional-Only Arguments Examples

Let's understand positional-only arguments with the help of some examples –

Example 1

In this example, we make both the arguments of `intr()` function as positional-only by putting "/" at the end.

```
def intr(amt, rate, /):
    val = amt * rate / 100
    return val

print(intr(316200, 4))
```

When you run the code, it will show the following result –

```
12648.0
```

Example 2

If we try to use the arguments as keywords, Python raises errors as shown in the below example.

```
def intr(amt, rate, /):
    val = amt * rate / 100
    return val

print(intr(amt=1000, rate=10))
```

On running this code, it will show following error message –

```
interest = intr(amt=1000, rate=10)
^~~~~~
TypeError: intr() got some positional-only arguments passed as keyword
arguments: 'amt, rate'
```

Example 3

A function may be defined in such a way that it has some keyword-only and some positional-only arguments. Here, x is a required positional-only argument, y is a regular positional argument, and z is a keyword-only argument.

```
def myfunction(x, /, y, *, z):
    print (x, y, z)

myfunction(10, y=20, z=30)
myfunction(10, 20, z=30)
```

The above code will show the following output –

```
10 20 30
10 20 30
```

49. Python - Arbitrary or, Variable-length Arguments

Arbitrary Arguments (*args)

You may want to define a function that is able to accept arbitrary or variable number of arguments. Moreover, the arbitrary number of arguments might be positional or keyword arguments.

- An argument prefixed with a single asterisk * for arbitrary positional arguments.
- An argument prefixed with two asterisks ** for arbitrary keyword arguments.

Arbitrary Arguments Example

Given below is an example of arbitrary or variable length positional arguments –

```
# sum of numbers
def add(*args):
    s=0
    for x in args:
        s=s+x
    return s
result = add(10,20,30,40)
print (result)

result = add(1,2,3)
print (result)
```

The args variable prefixed with "*" stores all the values passed to it. Here, args becomes a tuple. We can run a loop over its items to add the numbers.

It will produce the following output –

```
100
6
```

Required Arguments with Arbitrary Arguments

It is also possible to have a function with some required arguments before the sequence of variable number of values.

Example

The following example has avg() function. Assume that a student can take any number of tests. First test is mandatory. He can take as many tests as he likes to better his score. The function calculates the average of marks in first test and his maximum score in the rest of tests.

The function has two arguments, first is the required argument and second to hold any number of values.

```
#avg of first test and best of following tests
def avg(first, *rest):
    second=max(rest)
    return (first+second)/2

result=avg(40,30,50,25)
print (result)
```

Following call to avg() function passes first value to the required argument first, and the remaining values to a tuple named rest. We then find the maximum and use it to calculate the average.

It will produce the following output –

```
45.0
```

Arbitrary Keyword Arguments (**kwargs)

If a variable in the argument list has two asterisks prefixed to it, the function can accept arbitrary number of keyword arguments. The variable becomes a dictionary of keyword:value pairs.

Example

The following code is an example of a function with arbitrary keyword arguments. The addr() function has an argument **kwargs which is able to accept any number of address elements like name, city, phno, pin, etc. Inside the function kwargs dictionary of kw:value pairs is traversed using items() method.

```
def addr(**kwargs):
    for k,v in kwargs.items():
        print ("{}:{}".format(k,v))

    print ("pass two keyword args")
    addr(Name="John", City="Mumbai")
    print ("pass four keyword args")

# pass four keyword args
addr(Name="Raam", City="Mumbai", ph_no="9123134567", PIN="400001")
```

It will produce the following output –

```
pass two keyword args
Name:John
City:Mumbai
```

```
pass four keyword args
Name:Raam
City:Mumbai
ph_no:9123134567
PIN:400001
```

Multiple Arguments with Arbitrary Keyword Arguments

If the function uses mixed types of arguments, the arbitrary keyword arguments should be after positional, keyword and arbitrary positional arguments in the argument list.

Example

Imagine a case where science and maths are mandatory subjects, in addition to which student may choose any number of elective subjects.

The following code defines a percent() function where marks in science and maths are stored in required arguments, and the marks in variable number of elective subjects in **optional argument.

```
def percent(math, sci, **optional):
    print ("maths:", math)
    print ("sci:", sci)
    s=math+sci
    for k,v in optional.items():
        print ("{}:{}".format(k,v))
        s=s+v
    return s/(len(optional)+2)

result=percent(math=80, sci=75, Eng=70, Hist=65, Geo=72)
print ("percentage:", result)
```

It will produce the following output –

```
maths: 80
sci: 75
Eng:70
Hist:65
Geo:72
percentage: 72.4
```

50. Python Variable Scope

The scope of a variable in Python is defined as the specific area or region where the variable is accessible to the user. The scope of a variable depends on where and how it is defined. In Python, a variable can have either a global or a local scope.

Types of Scope for Variables in Python

On the basis of scope, the Python variables are classified in three categories –

- Local Variables
- Global Variables
- Nonlocal Variables

Local Variables

A local variable is defined within a specific function or block of code. It can only be accessed by the function or block where it was defined, and it has a limited scope. In other words, the scope of local variables is limited to the function they are defined in and attempting to access them outside of this function will result in an error. Always remember, multiple local variables can exist with the same name.

Example

The following example shows the scope of local variables.

```
def myfunction():
    a = 10
    b = 20
    print("variable a:", a)
    print("variable b:", b)
    return a+b

print (myfunction())
```

In the above code, we have accessed the local variables through its function. Hence, the code will produce the following output –

```
variable a: 10
variable b: 20
30
```

Global Variables

A global variable can be accessed from any part of the program, and it is defined outside any function or block of code. It is not specific to any block or function.

Example

The following example shows the scope of global variable. We can access them inside as well as outside of the function scope.

```
#global variables
name = 'TutorialsPoint'
marks = 50
def myfunction():
    # accessing inside the function
    print("name:", name)
    print("marks:", marks)
# function call
myfunction()
```

The above code will produce the following output –

```
name: TutorialsPoint
marks: 50
```

Nonlocal Variables

The Python variables that are not defined in either local or global scope are called nonlocal variables. They are used in nested functions.

Example

The following example demonstrates the how nonlocal variables works.

```
def yourfunction():
    a = 5
    b = 6
    # nested function
    def myfunction():
        # nonlocal function
        nonlocal a
        nonlocal b
        a = 10
        b = 20
        print("variable a:", a)
        print("variable b:", b)
        return a+b
    print (myfunction())
yourfunction()
```

The above code will produce the below output –

```
variable a: 10
variable b: 20
30
```

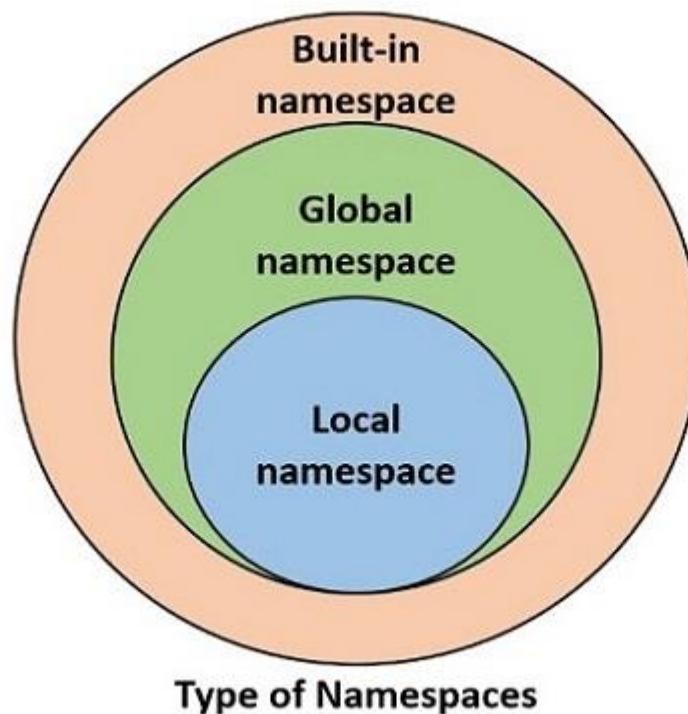
Namespace and Scope of Python Variables

A namespace is a collection of identifiers, such as variable names, function names, class names, etc. In Python, namespace is used to manage the scope of variables and to prevent naming conflicts.

Python provides the following types of namespaces –

- **Built-in namespace** contains built-in functions and built-in exceptions. They are loaded in the memory as soon as Python interpreter is loaded and remain till the interpreter is running.
- **Global namespace** contains any names defined in the main program. These names remain in memory till the program is running.
- **Local namespace** contains names defined inside a function. They are available till the function is running.

These namespaces are nested one inside the other. Following diagram shows relationship between namespaces.



The life of a certain variable is restricted to the namespace in which it is defined. As a result, it is not possible to access a variable present in the inner namespace from any outer namespace.

Python `globals()` Function

Python's standard library includes a built-in function `globals()`. It returns a dictionary of symbols currently available in global namespace.

Run the `globals()` function directly from the Python prompt.

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```

It can be seen that the built-in module which contains definitions of all built-in functions and built-in exceptions is loaded.

Example

Save the following code that contains few variables and a function with few more variables inside it.

```
name = 'TutorialsPoint'
marks = 50
result = True
def myfunction():
    a = 10
    b = 20
    return a+b

print (globals())
```

Calling `globals()` from inside this script returns following dictionary object –

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x00000263E7255250>,
 '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'C:\\Users\\\\user\\\\examples\\\\main.py', '__cached__':
None, 'name': 'TutorialsPoint', 'marks': 50, 'result': True, 'myfunction':
<function myfunction at 0x00000263E72004A0>}
```

The global namespace now contains variables in the program and their values and the function object in it (and not the variables in the function).

Python `locals()` Function

Python's standard library includes a built-in function called `locals()`. It returns a dictionary of symbols currently available in the local namespace of the function.

Example

Modify the above script to print dictionary of global and local namespaces from within the function.

```
name = 'TutorialsPoint'
marks = 50
result = True
def myfunction():
```

```

a = 10
b = 20
c = a+b
print ("globals():", globals())
print ("locals():", locals())
return c
myfunction()

```

The output shows that locals() returns a dictionary of variables and their values currently available in the function.

```

globals(): {'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x00000169AE265250>, '__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>, '__file__':
'C:\\\\Users\\\\mlath\\\\examples\\\\main.py', '__cached__': None, 'name':
'TutorialsPoint', 'marks': 50, 'result': True, 'myfunction': <function
myfunction at 0x00000169AE2104A0>}
locals(): {'a': 10, 'b': 20, 'c': 30}

```

Since both globals() and locals() functions return dictionary, you can access value of a variable from respective namespace with dictionary get() method or index operator.

```

print (globals()['name']) # displays TutorialsPoint
print (locals().get('a')) # displays 10

```

Namespace Conflict in Python

If a variable of same name is present in global as well as local scope, Python interpreter gives priority to the one in local namespace.

Example

In the following example, we define a local and a global variable.

```

marks = 50 # this is a global variable
def myfunction():
    marks = 70 # this is a local variable
    print (marks)

myfunction()
print (marks) # prints global value

```

It will produce the following output –

```

70
50

```

Example

If you try to manipulate value of a global variable from inside a function, Python raises `UnboundLocalError` as shown in example below –

```
# this is a global variable
marks = 50

def myfunction():
    marks = marks + 20
    print (marks)

myfunction()
# prints global value
print (marks)
```

It will produce the following error message –

```
marks = marks + 20
^^^^^
UnboundLocalError: cannot access local variable 'marks' where it is not
associated with a value
```

Example

To modify a global variable, you can either update it with a dictionary syntax, or use the `global` keyword to refer it before modifying.

```
var1 = 50 # this is a global variable
var2 = 60 # this is a global variable

def myfunction():
    "Change values of global variables"
    globals()['var1'] = globals()['var1']+10
    global var2
    var2 = var2 + 20

myfunction()

print ("var1:",var1, "var2:",var2) #shows global variables with changed values
```

On executing the code, it will produce the following output –

```
var1: 60 var2: 80
```

Example

Lastly, if you try to access a local variable in global scope, Python raises `NameError` as the variable in local scope can't be accessed outside it.

```
var1 = 50 # this is a global variable
var2 = 60 # this is a global variable

def myfunction(x, y):
    total = x+y
    print ("Total is a local variable: ", total)

myfunction(var1, var2)
print (total) # This gives NameError
```

It will produce the following error message —

```
Total is a local variable: 110
Traceback (most recent call last):
  File "C:\Users\user\examples\main.py", line 9, in <module>
    print (total) # This gives NameError
               ^
NameError: name 'total' is not defined
```

51. Python - Function Annotations

Function Annotations

The function annotation feature of Python enables you to add additional explanatory metadata about the arguments declared in a function definition, and also the return data types. They are not considered by Python interpreter while executing the function. Python IDEs use them for providing a detailed documentation to the programmer.

Although you can use the docstring feature of Python for documentation of a function, it may be obsolete if certain changes in the function's prototype are made. Hence, the annotation feature was introduced in Python as a result of PEP 3107.

Annotations are any valid Python expressions added to the arguments or return data type. Simplest example of annotation is to prescribe the data type of the arguments. Annotation is mentioned as an expression after putting a colon in front of the argument.

Example

Remember that Python is a dynamically typed language, and doesn't enforce any type checking at runtime. Hence annotating the arguments with data types doesn't have any effect while calling the function. Even if non-integer arguments are given, Python doesn't detect any error.

```
def myfunction(a: int, b: int):
    c = a+b
    return c

print (myfunction(10,20))
print (myfunction("Hello ", "Python"))
```

It will produce the following output –

```
30
Hello Python
```

Function Annotations with Return Type

Annotations are ignored at runtime, but are helpful for the IDEs and static type checker libraries such as mypy.

You can give annotation for the return data type as well. After the parentheses and before the colon symbol, put an arrow (->) followed by the annotation.

Example

In this example, we are providing annotation for return type.

```
def myfunction(a: int, b: int) -> int:
```

```
c = a+b
return c

print(myfunction(56,88))
print(myfunction.__annotations__)
```

This will generate the following output –

```
144
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Function Annotations with Expression

As using the data type as annotation is ignored at runtime, you can put any expression which acts as the metadata for the arguments. Hence, function may have any arbitrary expression as annotation.

Example

In the below example, we are using expression as a function annotation.

```
def total(x : 'marks in Physics', y: 'marks in chemistry'):
    return x+y
print(total(86, 88))
print(total.__annotations__)
```

Following is the output –

```
174
{'x': 'marks in Physics', 'y': 'marks in chemistry'}
```

Function Annotations with Default Arguments

If you want to specify a default argument along with the annotation, you need to put it after the annotation expression. Default arguments must come after the required arguments in the argument list.

Example 1

The following example demonstrates how to provide annotation for default arguments of a function.

```
def myfunction(a: "physics", b:"Maths" = 20) -> int:
    c = a+b
    return c
print (myfunction(10))
```

The function in Python is also an object, and one of its attributes is `__annotations__`. You can check with `dir()` function.

```
print (dir(myfunction))
```

This will print the list of myfunction object containing `__annotations__` as one of the attributes.

```
[ '__annotations__', '__builtins__', '__call__', '__class__', '__closure__',
  '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__',
  '__eq__', '__format__', '__ge__', '__get__', '__getattribute__', '__getattr__',
  '__globals__', '__gt__', '__hash__', '__init__', '__init_subclass__',
  '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__',
  '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
  '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Example 2

The `__annotations__` attribute itself is a dictionary in which arguments are keys and annotations their values.

```
def myfunction(a: "physics", b:"Maths" = 20) -> int:
    c = a+b
    return c
print (myfunction.__annotations__)
```

It will produce the following output –

```
{'a': 'physics', 'b': 'Maths', 'return': <class 'int'>}
```

Example 3

You may have arbitrary positional and/or arbitrary keyword arguments for a function. Annotations can be given for them also.

```
def myfunction(*args: "arbitrary args", **kwargs: "arbitrary keyword args") ->
    int:
    pass
print (myfunction.__annotations__)
```

It will produce the following output –

```
{'args': 'arbitrary args', 'kwargs': 'arbitrary keyword args', 'return': <class 'int'>}
```

Example 4

In case you need to provide more than one annotation expressions to a function argument, give it in the form of a dictionary object in front of the argument itself.

```
def division(num: dict(type=float, msg='numerator'), den: dict(type=float,
    msg='denominator')) -> float:
    return num/den
print (division.__annotations__)
```

It will produce the following output –

```
{'num': {'type': <class 'float'>, 'msg': 'numerator'}, 'den': {'type': <class 'float'>, 'msg': 'denominator'}, 'return': <class 'float'>}
```

52. Python - Modules

Python Modules

The concept of module in Python further enhances the modularity. You can define more than one related functions together and load required functions. A module is a file containing definition of functions, classes, variables, constants or any other Python object. Contents of this file can be made available to any other program. Python has the **import** keyword for this purpose.

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Example of Python Module

```
import math  
print ("Square root of 100:", math.sqrt(100))
```

It will produce the following output –

```
Square root of 100: 10.0
```

Python Built-in Modules

Python's standard library comes bundled with a large number of modules. They are called built-in modules. Most of these built-in modules are written in C (as the reference implementation of Python is in C), and pre-compiled into the library. These modules come with useful functionality like system-specific OS management, disk IO, networking, etc.

Here is a select list of built-in modules –

Sr.No.	Name & Brief Description
1	<u>os</u> This module provides a unified interface to a number of operating system functions.
2	<u>string</u> This module contains a number of functions for string processing
3	<u>re</u> This module provides a set of powerful regular expression facilities. Regular expression (RegEx), allows powerful string search and matching for a pattern in a string
4	<u>math</u>

	This module implements a number of mathematical operations for floating point numbers. These functions are generally thin wrappers around the platform C library functions.
5	cmath This module contains a number of mathematical operations for complex numbers.
6	datetime This module provides functions to deal with dates and the time within a day. It wraps the C runtime library.
7	gc This module provides an interface to the built-in garbage collector.
8	asyncio This module defines functionality required for asynchronous processing
9	Collections This module provides advanced Container datatypes.
10	functools This module has Higher-order functions and operations on callable objects. Useful in functional programming
11	operator Functions corresponding to the standard operators.
12	pickle Convert Python objects to streams of bytes and back.
13	socket Low-level networking interface.
14	sqlite3 A DB-API 2.0 implementation using SQLite 3.x.
15	statistics Mathematical statistics functions
16	typing Support for type hints
17	venv Creation of virtual environments.
18	json Encode and decode the JSON format.
19	wsgiref WSGI Utilities and Reference Implementation.

20	unittest
	Unit testing framework for Python.
21	random
	Generate pseudo-random numbers
22	sys
	Provides functions that acts strongly with the interpreter.
23	requests
	It simplifies HTTP requests by offering a user-friendly interface for sending and handling responses.

It simplifies HTTP requests by offering a user-friendly interface for sending and handling responses.

Python User-defined Modules

Any text file with .py extension and containing Python code is basically a module. It can contain definitions of one or more functions, variables, constants as well as classes. Any Python object from a module can be made available to interpreter session or another Python script by import statement. A module can also include runnable code.

Creating a Python Module

Creating a module is nothing but saving a Python code with the help of any editor. Let us save the following code as mymodule.py

```
def SayHello(name):
    print ("Hi {}! How are you?".format(name))
    return
```

You can now import mymodule in the current Python terminal.

```
>>> import mymodule
>>> mymodule.SayHello("Harish")
Hi Harish! How are you?
```

You can also import one module in another Python script. Save the following code as example.py

```
import mymodule
mymodule.SayHello("Harish")
```

Run this script from command terminal

```
Hi Harish! How are you?
```

The import Statement

In Python, the import keyword has been provided to load a Python object from one module. The object may be a function, a class, a variable etc. If a module contains multiple definitions, all of them will be loaded in the namespace.

Let us save the following code having three functions as mymodule.py.

```
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

The import mymodule statement loads all the functions in this module in the current namespace. Each function in the imported module is an attribute of this module object.

```
>>> dir(mymodule)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'average', 'power', 'sum']
```

To call any function, use the module object's reference. For example, mymodule.sum().

```
import mymodule
print ("sum:",mymodule.sum(10,20))
print ("average:",mymodule.average(10,20))
print ("power:",mymodule.power(10, 2))
```

It will produce the following output –

```
sum:30
average:15.0
power:100
```

The from ... import Statement

The import statement will load all the resources of the module in the current namespace. It is possible to import specific objects from a module by using this syntax. For example –

Out of three functions in mymodule, only two are imported in following executable script example.py

```
from mymodule import sum, average
print ("sum:",sum(10,20))
```

```
print ("average:",average(10,20))
```

It will produce the following output –

```
sum: 30
average: 15.0
```

Note that function need not be called by prefixing name of its module to it.

The from...import * Statement

It is also possible to import all the names from a module into the current namespace by using the following import statement –

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

The import ... as Statement

You can assign an alias name to the imported module.

```
from modulename as alias
```

The alias should be prefixed to the function while calling.

Take a look at the following example –

```
import mymodule as x
print ("sum:",x.sum(10,20))
print ("average:", x.average(10,20))
print ("power:", x.power(10, 2))
```

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences –

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system –

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system –

```
set PYTHONPATH = /usr/local/lib/python
```

Namespaces and Scoping

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values).

- A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- In order to assign a value to a global variable within a function, you must first use the `global` statement.
- The statement `global VarName` tells Python that `VarName` is a global variable. Python stops searching the local namespace for the variable.

Example

For example, we define a variable `Money` in the global namespace. Within the function `Money`, we assign `Money` a value, therefore Python assumes `Money` as a local variable. However, we accessed the value of the local variable `Money` before setting it, so an `UnboundLocalError` is the result. Uncommenting the `global` statement fixes the problem.

```
Money = 2000

def AddMoney():
    # Uncomment the following line to fix the code:
    # global Money
    Money = Money + 1

    print (Money)
AddMoney()
print (Money)
```

Module Attributes

In Python, a module is an object of `module` class, and hence it is characterized by attributes.

Following are the module attributes –

- `__file__` returns the physical name of the module.
- `__package__` returns the package to which the module belongs.
- `__doc__` returns the docstring at the top of the module if any

- `__dict__` returns the entire scope of the module
- `__name__` returns the name of the module

Example

Assuming that the following code is saved as `mymodule.py`

```
"The docstring of mymodule"

def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

Let us check the attributes of `mymodule` by importing it in the following script –

```
import mymodule

print ("__file__ attribute:", mymodule.__file__)
print ("__doc__ attribute:", mymodule.__doc__)
print ("__name__ attribute:", mymodule.__name__)
```

It will produce the following output –

```
__file__ attribute: C:\math\examples\mymodule.py
__doc__ attribute: The docstring of mymodule
__name__ attribute: mymodule
```

The `__name__` Attribute

The `__name__` attribute of a Python module has great significance. Let us explore it in more detail.

In an interactive shell, `__name__` attribute returns '`__main__`'

```
>>> __name__
'__main__'
```

If you import any module in the interpreter session, it returns the name of the module as the `__name__` attribute of that module.

```
>>> import math
>>> math.__name__
'math'
```

From inside a Python script, the `__name__` attribute returns '`__main__`'

```
#example.py
print ("__name__ attribute within a script:", __name__)
Run this in the command terminal -
__name__ attribute within a script: __main__
```

This attribute allows a Python script to be used as executable or as a module. Unlike in C++, Java, C# etc., in Python, there is no concept of the `main()` function. The Python program script with `.py` extension can contain function definitions as well as executable statements.

Save `mymodule.py` and with the following code –

```
"The docstring of mymodule"
def sum(x,y):
    return x+y

print ("sum:",sum(10,20))
```

You can see that `sum()` function is called within the same script in which it is defined.

```
sum: 30
```

Now let us import this function in another script `example.py`.

```
import mymodule
print ("sum:",mymodule.sum(10,20))
```

It will produce the following output –

```
sum: 30
sum: 30
```

The output "sum:30" appears twice. Once when `mymodule` module is imported, the executable statements in imported module also run. Second output is from the calling script, i.e., `example.py` program.

What we want to explain is that when a module is imported, only the function should be imported, its executable statements should not run. This can be done by checking the value of `__name__`. If it is `__main__`, means it is being run and not imported, include the executable statements like function calls conditionally.

Add if statement in `mymodule.py` as shown –

```
"The docstring of mymodule"
def sum(x,y):
    return x+y

if __name__ == "__main__":
```

```
print ("sum:",sum(10,20))
```

Now if you run example.py program, you will find that the sum:30 output appears only once.

```
sum: 30
```

The dir() Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example –

```
# Import built-in module math
import math

content = dir(math)
print (content)
```

When the above code is executed, it produces the following result –

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

The reload() Function

Sometimes you may need to reload a module, especially when working with the interactive interpreter session of Python.

Assume that we have a test module (test.py) with the following function –

```
def SayHello(name):
    print ("Hi {}! How are you?".format(name))
    return
```

We can import the module and call its function from Python prompt as –

```
>>> import test
>>> test.SayHello("Deepak")
Hi Deepak! How are you?
```

However, suppose you need to modify the SayHello() function, such as –

```
def SayHello(name, course):
```

```

print ("Hi {}! How are you?".format(name))
print ("Welcome to {} Tutorial by TutorialsPoint".format(course))
return

```

Even if you edit the test.py file and save it, the function loaded in the memory won't update. You need to reload it, using reload() function in imp module.

```

>>> import imp
>>> imp.reload(test)
>>> test.SayHello("Deepak", "Python")
Hi Deepak! How are you?
Welcome to Python Tutorial by TutorialsPoint

```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules, subpackages and, sub-subpackages, and so on.

Consider a file Pots.py available in Phone directory. This file has following line of source code –

```

def Pots():
    print "I'm Pots Phone"

```

Similar way, we have another two files having different functions with the same name as above –

- Phone/Isdn.py file having function Isdn()
- Phone/G3.py file having function G3()

Now, create one more file __init__.py in Phone directory –

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows –

```

from Pots import Pots
from Isdn import Isdn
from G3 import G3

```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```

# Now import your Phone Package.

import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()

```

When the above code is executed, it produces the following result –

```
I'm Pots Phone  
I'm 3G Phone  
I'm ISDN Phone
```

In the above example, we have taken example of a single function in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

53. Python - Built-in Functions

Built-in Functions in Python?

Built-in functions are those functions that are pre-defined in the Python interpreter and you don't need to import any module to use them. These functions help to perform a wide variety of operations on strings, iterators, and numbers. For instance, the built-in functions like sum(), min(), and max() are used to simplify mathematical operations.

How to Use Built-in Function in Python?

To use built-in functions in your code, simply call the specific function by passing the required parameter (if any) inside the parentheses. Since these functions are pre-defined, you don't need to import any module or package.

Example of Using Built-in Functions

Consider the following example demonstrating the use of built-in functions in your code:

```
# Using print() and len() function

text = "Tutorials Point"

print(len(text)) # Prints 15
```

In the above example, we are using two built-in functions print() and len().

List of Python Built-in Functions

As of Python 3.12.2 version, the list of built-in functions is given below –

Sr.No.	Function & Description
1	Python aiter() function Returns an asynchronous iterator for an asynchronous iterable.
2	Python all() function Returns true when all elements in iterable is true.
3	Python anext() function Returns the next item from the given asynchronous iterator.
4	Python any() function Checks if any Element of an Iterable is True.
5	Python ascii() function Returns String Containing Printable Representation.
6	Python bin() function Converts integer to binary string.
7	Python bool() function

	Converts a Value to Boolean.
8	Python breakpoint() function This function drops you into the debugger at the call site and calls sys.breakpointhook().
9	Python bytearray() function Returns array of given byte size.
10	Python bytes() function Returns immutable bytes object.
11	Python callable() function Checks if the Object is Callable.
12	Python chr() function Returns a Character (a string) from an Integer.
13	Python classmethod() function Returns class method for given function.
14	Python compile() function Returns a code object.
15	Python complex() function Creates a Complex Number.
16	Python delattr() function Deletes Attribute From the Object.
17	Python dict() function Creates a Dictionary.
18	Python dir() function Tries to Return Attributes of Object.
19	Python divmod() function Returns a Tuple of Quotient and Remainder.
20	Python enumerate() function Returns an Enumerate Object.
21	Python eval() function Runs Code Within Program.
22	Python exec() function Executes Dynamically Created Program.
23	Python filter() function Constructs iterator from elements which are true.
24	Python float() function Returns floating point number from number, string.
25	Python format() function Returns formatted representation of a value.
26	Python frozenset() function Returns immutable frozenset object.
27	Python getattr() function Returns value of named attribute of an object.
28	Python globals() function Returns dictionary of current global symbol table.
29	Python hasattr() function

	Returns whether object has named attribute.
30	Python hash() function Returns hash value of an object.
31	Python help() function Invokes the built-in Help System.
32	Python hex() function Converts to Integer to Hexadecimal.
33	Python id() function Returns Identify of an Object.
34	Python input() function Reads and returns a line of string.
35	Python int() function Returns integer from a number or string.
36	Python isinstance() function Checks if a Object is an Instance of Class.
37	Python issubclass() function Checks if a Class is Subclass of another Class.
38	Python iter() function Returns an iterator.
39	Python len() function Returns Length of an Object.
40	Python list() function Creates a list in Python.
41	Python locals() function Returns dictionary of a current local symbol table.
42	Python map() function Applies Function and Returns a List.
43	Python memoryview() function Returns memory view of an argument.
44	Python next() function Retrieves next item from the iterator.
45	Python object() function Creates a featureless object.
46	Python oct() function Returns the octal representation of an integer.
47	Python open() function Returns a file object.
48	Python ord() function Returns an integer of the Unicode character.
49	Python print() function Prints the Given Object.
50	Python property() function Returns the property attribute.
51	Python range() function Returns a sequence of integers.
52	Python repr() function

	Returns a printable representation of the object.
53	Python reversed() function Returns the reversed iterator of a sequence.
54	Python set() function Constructs and returns a set.
55	Python setattr() function Sets the value of an attribute of an object.
56	Python slice() function Returns a slice object.
57	Python sorted() function Returns a sorted list from the given iterable.
58	Python staticmethod() function Transforms a method into a static method.
59	Python str() function Returns the string version of the object.
60	Python super() function Returns a proxy object of the base class.
61	Python tuple() function Returns a tuple.
62	Python type() function Returns the type of the object.
63	Python vars() function Returns the __dict__ attribute.
64	Python zip() function Returns an iterator of tuples.
65	Python import () function Function called by the import statement.
66	Python unichr() function Converts a Unicode code point to its corresponding Unicode character.
67	Python long() function Represents integers of arbitrary size.

Built-in Mathematical Functions

There are some additional built-in functions that are used for performing only mathematical operations in Python, they are listed below –

Sr.No.	Function & Description
1	Python abs() function The abs() function returns the absolute value of x, i.e. the positive distance between x and zero.
2	Python max() function The max() function returns the largest of its arguments or largest number from the iterable (list or tuple).
3	Python min() function

	The function min() returns the smallest of its arguments i.e. the value closest to negative infinity, or smallest number from the iterable (list or tuple)
4	Python pow() function The pow() function returns x raised to y . It is equivalent to $x^{**}y$. The function has third optional argument mod. If given, it returns $(x^{**}y) \% \text{mod}$ value
5	Python round() Function round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.
6	Python sum() function The sum() function returns the sum of all numeric items in any iterable (list or tuple). An optional start argument is 0 by default. If given, the numbers in the list are added to start value.

Advantages of Using Built-in Functions

The following are the advantages of using built-in functions:

- The use of the built-in functions simplifies and reduces the code length and enhances the readability of the code.
- Instead of writing the same logic repeatedly, you can use these functions across different sections of the program. This not only saves time but also helps in maintaining consistency of code.
- These functions provide a wide range of functionalities including mathematical operations, datatype conversion, and performing operations on iterators.
- These functions have descriptive names that make the code easier to understand and maintain. Developers need not write additional complex code for performing certain operations.

Frequently Asked Questions about Built-in Functions

How do I handle errors with built-in functions?

While working with built-in functions, you may encounter errors and to handle those errors you can use the try-except blocks. This may help you identify the type of error and exceptions raised.

Can we extend the functionality of built-in functions?

Yes, we can extend the functionality of built-in functions by using them with other methods and by applying our logic as per the need. However, it will not affect the pre-defined feature of the used function.

Can I create my built-in functions?

No, you cannot create your built-in function. But, Python allows a user to create user-defined functions.

How do I use built-in functions?

Using a built-in function is very simple, call it by its name followed by parentheses, and pass the required arguments inside the parentheses.

Python Strings

54. Python - Strings

In Python, a string is an immutable sequence of Unicode characters. Each character has a unique numeric value as per the UNICODE standard. But, the sequence as a whole, doesn't have any numeric value even if all the characters are digits. To differentiate the string from numbers and other identifiers, the sequence of characters is included within single, double or triple quotes in its literal representation. Hence, 1234 is a number (integer) but '1234' is a string.

Creating Python Strings

As long as the same sequence of characters is enclosed, single or double or triple quotes don't matter. Hence, following string representations are equivalent.

Example

```
>>> 'Welcome To TutorialsPoint'  
'Welcome To TutorialsPoint'  
>>> "Welcome To TutorialsPoint"  
'Welcome To TutorialsPoint'  
>>> '''Welcome To TutorialsPoint'''  
'Welcome To TutorialsPoint'  
>>> """Welcome To TutorialsPoint"""  
'Welcome To TutorialsPoint'
```

Looking at the above statements, it is clear that, internally Python stores strings as included in single quotes.

In older versions strings are stored internally as 8-bit ASCII, hence it is required to attach 'u' to make it Unicode. Since Python 3, all strings are represented in Unicode. Therefore, it is no longer necessary now to add 'u' after the string.

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example –

```
var1 = 'Hello World!'  
var2 = "Python Programming"  
  
print ("var1[0]: ", var1[0])  
print ("var2[1:5]: ", var2[1:5])
```

When the above code is executed, it produces the following result –

```
var1[0]: H
var2[1:5]: ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example –

```
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

When the above code is executed, it produces the following result –

```
Updated String :- Hello Python
```

Visit our [Python - Modify Strings](#) tutorial to know more about updating/modifying strings.

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0..7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0..9, a..f, or A..F

String Special Operators

Assume string variable a holds 'Hello' and variable b holds 'Python', then –

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0..7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0..9, a..f, or A..F

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the lack of having functions from C's printf() family. Following is a simple example –

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

When the above code is executed, it produces the following result –

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % –

Sr.No.	Format Symbol & Conversion
1	%c character
2	%s string conversion via str() prior to formatting
3	%i signed decimal integer
4	%d signed decimal integer
5	%u unsigned decimal integer
6	%o octal integer
7	%x

	hexadecimal integer (lowercase letters)
8	%X
	hexadecimal integer (UPPERcase letters)
9	%e
	exponential notation (with lowercase 'e')
10	%E
	exponential notation (with UPPERcase 'E')
11	%f
	floating point real number
12	%g
	the shorter of %f and %e
13	%G
	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table –

Sr.No.	Symbol & Functionality
1	*
	argument specifies width or precision
2	-
	left justification
3	+
	display the sign
4	<sp>
	leave a blank space before a positive number
5	#
	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
6	0
	pad from left with zeros (instead of spaces)
7	%
	'%%%' leaves you with a single literal '%'
8	(var)
	mapping variable (dictionary arguments)
9	m.n.
	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Visit our [Python - String Formatting](#) tutorial to learn about various ways to format strings.

Double Quotes in Python Strings

You want to embed some text in double quotes as a part of string, the string itself should be put in single quotes. To embed a single quoted text, string should be written in double quotes.

Example

```
var = 'Welcome to "Python Tutorial" from TutorialsPoint'
print ("var:", var)

var = "Welcome to 'Python Tutorial' from TutorialsPoint"
print ("var:", var)
```

It will produce the following output –

```
var: Welcome to "Python Tutorial" from TutorialsPoint
var: Welcome to 'Python Tutorial' from TutorialsPoint
```

Triple Quotes

To form a string with triple quotes, you may use triple single quotes, or triple double quotes – both versions are similar.

Example

```
var = '''Welcome to TutorialsPoint'''
print ("var:", var)

var = """Welcome to TutorialsPoint"""
print ("var:", var)
```

It will produce the following output –

```
var: Welcome to TutorialsPoint
var: Welcome to TutorialsPoint
```

Python Multiline Strings

Triple quoted string is useful to form a multi-line string.

Example

```
var = '''
Welcome To
Python Tutorial
from TutorialsPoint
'''
```

```
print ("var:", var)
```

It will produce the following output –

```
var:  
Welcome To  
Python Tutorial  
from TutorialsPoint
```

Arithmetic Operators with Strings

A string is a non-numeric data type. Obviously, we cannot use arithmetic operators with string operands. Python raises `TypeError` in such a case.

```
print ("Hello"- "World")
```

On executing the above program it will generate the following error –

```
>>> "Hello"- "World"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Getting Type of Python Strings

A string in Python is an object of `str` class. It can be verified with `type()` function.

Example

```
var = "Welcome To TutorialsPoint"  
print (type(var))
```

It will produce the following output –

```
<class 'str'>
```

Built-in String Methods

Python includes the following built-in methods to manipulate strings –

Sr.No.	Methods with Description
1	<u>capitalize()</u> Capitalizes first letter of string.
2	<u>casefold()</u> Converts all uppercase letters in string to lowercase. Similar to <code>lower()</code> , but works on UNICODE characters also.
3	<u>center(width, fillchar)</u> Returns a space-padded string with the original string centered to a total of width columns.

	<code>count(str, beg= 0,end=len(string))</code>
4	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
5	<code>decode(encoding='UTF-8',errors='strict')</code> Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
6	<code>encode(encoding='UTF-8',errors='strict')</code> Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
7	<code>endswith(suffix, beg=0, end=len(string))</code> Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
8	<code>expandtabs(tabsize=8)</code> Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
9	<code>find(str, beg=0 end=len(string))</code> Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
10	<code>format(*args, **kwargs)</code> This method is used to format the current string value.
11	<code>format_map(mapping)</code> This method is also use to format the current string the only difference is it uses a mapping object.
12	<code>index(str, beg=0, end=len(string))</code> Same as find(), but raises an exception if str not found.
13	<code>isalnum()</code> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
14	<code>isalpha()</code> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
15	<code>isascii()</code> Returns True is all the characters in the string are from the ASCII character set.
16	<code>isdecimal()</code>

	Returns true if a unicode string contains only decimal characters and false otherwise.
17	isdigit() Returns true if string contains only digits and false otherwise.
18	isidentifier() Checks whether the string is a valid Python identifier.
19	islower() Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
20	isnumeric() Returns true if a unicode string contains only numeric characters and false otherwise.
21	isprintable() Checks whether all the characters in the string are printable.
22	isspace() Returns true if string contains only whitespace characters and false otherwise.
23	istitle() Returns true if string is properly "titlecased" and false otherwise.
24	isupper() Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
25	join(seq) Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
26	ljust(width[, fillchar]) Returns a space-padded string with the original string left-justified to a total of width columns.
27	lower() Converts all uppercase letters in string to lowercase.
28	lstrip() Removes all leading white space in string.
29	maketrans() Returns a translation table to be used in translate function.
30	partition() Splits the string in three string tuple at the first occurrence of separator.
31	removeprefix()

	Returns a string after removing the prefix string.
32	<code>removesuffix()</code> Returns a string after removing the suffix string.
33	<code>replace(old, new [, max])</code> Replaces all occurrences of old in string with new or at most max occurrences if max given.
34	<code>rfind(str, beg=0,end=len(string))</code> Same as find(), but search backwards in string.
35	<code>rindex(str, beg=0, end=len(string))</code> Same as index(), but search backwards in string.
36	<code>rjust(width,[, fillchar])</code> Returns a space-padded string with the original string right-justified to a total of width columns.
37	<code>rpartition()</code> Splits the string in three string tuple at the last occurrence of separator.
38	<code>rsplit()</code> Splits the string from the end and returns a list of substrings.
39	<code>rstrip()</code> Removes all trailing whitespace of string.
40	<code>split(str="", num=string.count(str))</code> Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
41	<code>splitlines(num=string.count('\n'))</code> Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
42	<code>startswith(str, beg=0,end=len(string))</code> Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
43	<code>strip([chars])</code> Performs both lstrip() and rstrip() on string.
44	<code>swapcase()</code> Inverts case for all letters in string.
45	<code>title()</code> Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
46	<code>translate(table, deletechars="")</code>

	Translates string according to translation table str(256 chars), removing those in the del string.
47	upper() Converts lowercase letters in string to uppercase.
48	zfill (width) Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

Built-in Functions with Strings

Following are the built-in functions we can use with strings –

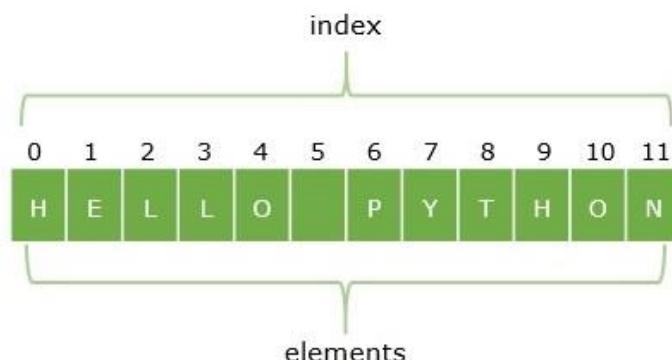
Sr.No.	Function with Description
1	len(list) Returns the length of the string.
2	max(list) Returns the max alphabetical character from the string str.
3	min(list) Returns the min alphabetical character from the string str.

55. Python Slicing Strings

Python String slicing is a way of creating a sub-string from a given string. In this process, we extract a portion or piece of a string. Usually, we use the slice operator "[:]" to perform slicing on a Python String. Before proceeding with string slicing let's understand string indexing.

In Python, a string is an ordered sequence of Unicode characters. Each character in the string has a unique index in the sequence. The index starts with 0. First character in the string has its positional index 0. The index keeps incrementing towards the end of string.

If a string variable is declared as var="HELLO PYTHON", index of each character in the string is as follows –



Python String Indexing

Python allows you to access any individual character from the string by its index. In this case, 0 is the lower bound and 11 is the upper bound of the string. So, var[0] returns H, var[6] returns P. If the index in square brackets exceeds the upper bound, Python raises IndexError.

Example

In the below example, we accessing the characters of a string through index.

```
var = "HELLO PYTHON"
print(var[0])
print(var[7])
print(var[11])
print(var[12])
```

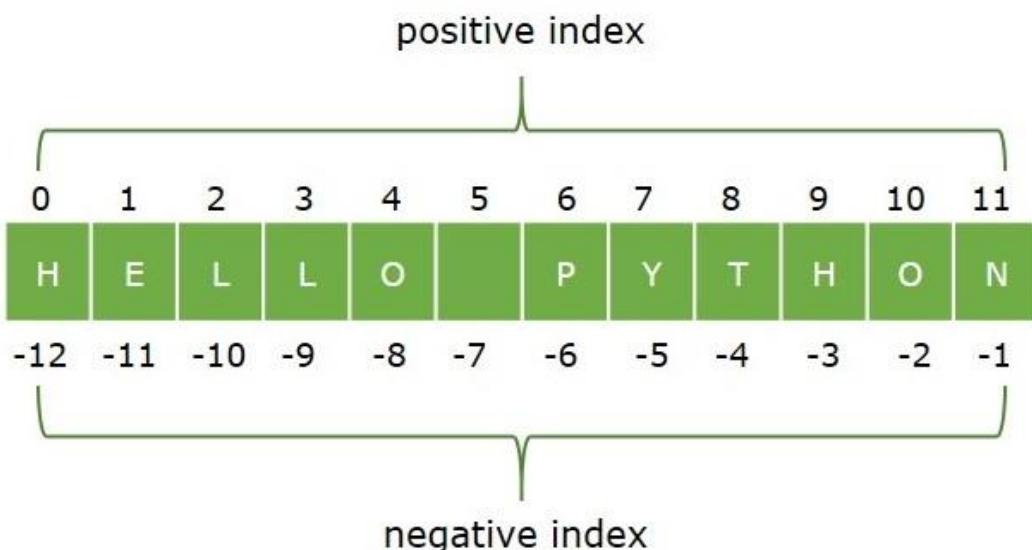
On running the code, it will produce the following output –

```
H
Y
N
ERROR!
```

```
Traceback (most recent call last):
  File "<main.py>", line 5, in <module>
    IndexError: string index out of range
```

Python String Negative & Positive Indexing

One of the unique features of Python sequence types (and therefore a string object) is that it has a negative indexing scheme also. In the example above, a positive indexing scheme is used where the index increments from left to right. In case of negative indexing, the character at the end has -1 index and the index decrements from right to left, as a result the first character H has -12 index.



Example

Let us use negative indexing to fetch N, Y, and H characters.

```
var = "HELLO PYTHON"
print(var[-1])
print(var[-5])
print(var[-12])
```

On executing the above code, it will give the following result –

```
N
Y
H
```

We can therefore use positive or negative index to retrieve a character from the string.

In Python, string is an immutable object. The object is immutable if it cannot be modified in-place, once stored in a certain memory location. You can retrieve any character from the string with the help of its index, but you cannot replace it with another character.

Example

In the following example, character Y is at index 7 in HELLO PYTHON. Try to replace Y with y and see what happens.

```
var="HELLO PYTHON"
var[7]="y"
print (var)
```

It will produce the following output –

```
Traceback (most recent call last):
  File "C:\Users\users\example.py", line 2, in <module>
    var[7]="y"
    ~~~^~~
TypeError: 'str' object does not support item assignment
```

The TypeError is because the string is immutable.

Python String Slicing

Python defines ":" as string slicing operator. It returns a substring from the original string. Its general usage is as follows –

```
substr=var[x:y]
```

The ":" operator needs two integer operands (both of which may be omitted, as we shall see in subsequent examples). The first operand x is the index of the first character of the desired slice. The second operand y is the index of the character next to the last in the desired string. So var(x:y] separates characters from xth position to (y-1)th position from the original string.

Example

```
var="HELLO PYTHON"

print ("var:",var)
print ("var[3:8]:", var[3:8])
```

It will produce the following output –

```
var: HELLO PYTHON
var[3:8]: LO PY
```

Python String Slicing With Negative Indexing

Like positive indexes, negative indexes can also be used for slicing.

Example

The below example shows how to slice a string using negative indexes.

```
var="HELLO PYTHON"
```

```
print ("var:",var)
print ("var[3:8]:", var[3:8])
print ("var[-9:-4]:", var[-9:-4])
```

It will produce the following output –

```
var: HELLO PYTHON
var[3:8]: LO PY
var[-9:-4]: LO PY
```

Default Values of Indexes with String Slicing

Both the operands for Python's Slice operator are optional. The first operand defaults to zero, which means if we do not give the first operand, the slice starts at 0th index, i.e. the first character. It slices the leftmost substring up to "y-1" characters.

Example

In this example, we are performing slice operation using default values.

```
var="HELLO PYTHON"
print ("var:",var)
print ("var[0:5]:", var[0:5])
print ("var[:5]:", var[:5])
```

It will produce the following output –

```
var: HELLO PYTHON
var[0:5]: HELLO
var[:5]: HELLO
```

Example

Similarly, y operand is also optional. By default, it is "-1", which means the string will be sliced from the xth position up to the end of string.

```
var="HELLO PYTHON"
print ("var:",var)
print ("var[6:12]:", var[6:12])
print ("var[6:]:", var[6:])
```

It will produce the following output –

```
var: HELLO PYTHON
var[6:12]: PYTHON
var[6:]: PYTHON
```

Example

Naturally, if both the operands are not used, the slice will be equal to the original string. That's because "x" is 0, and "y" is the last index+1 (or -1) by default.

```
var="HELLO PYTHON"
print ("var:",var)
print ("var[0:12]:", var[0:12])
print ("var[:]:", var[:])
```

It will produce the following output –

```
var: HELLO PYTHON
var[0:12]: HELLO PYTHON
var[:]: HELLO PYTHON
```

Example

The left operand must be smaller than the operand on right, for getting a substring of the original string. Python doesn't raise any error, if the left operand is greater, but returns a null string.

```
var="HELLO PYTHON"
print ("var:",var)
print ("var[-1:7]:", var[-1:7])
print ("var[7:0]:", var[7:0])
```

It will produce the following output –

```
var: HELLO PYTHON
var[-1:7]:
var[7:0]:
```

Return Type of String Slicing

Slicing returns a new string. You can very well perform string operations like concatenation, or slicing on the sliced string.

Example

```
var="HELLO PYTHON"

print ("var:",var)
print ("var[:6][:2]:", var[:6][:2])

var1=var[:6]
print ("slice:", var1)
print ("var1[:2]:", var1[:2])
```

It will produce the following output –

```
var: HELLO PYTHON
var[:6][:2]: HE
slice: HELLO
var1[:2]: HE
```

56. Python - Modify Strings

String modification refers to the process of changing the characters of a string. If we talk about modifying a string in Python, what we are talking about is creating a new string that is a variation of the original one.

In Python, a string (object of str class) is of immutable type. Here, immutable refers to an object that cannot be modified in place once it's created in memory. Unlike a list, we cannot overwrite any character in the sequence, nor can we insert or append characters to it directly. If we need to modify a string, we will use certain string methods that return a new string object. However, the original string remains unchanged.

We can use any of the following tricks as a workaround to modify a string.

Converting a String to a List

Both strings and lists in Python are sequence types, they are interconvertible. Thus, we can cast a string to a list, modify the list using methods like insert(), append(), or remove() and then convert the list back to a string to obtain a modified version.

Suppose, we have a string variable s1 with WORD as its value and we are required to convert it into a list. For this operation, we can use the list() built-in function and insert a character L at index 3. Then, we can concatenate all the characters using join() method of str class.

Example

The below example practically illustrates how to convert a string into a list.

```
s1="WORD"
print ("original string:", s1)
l1=list(s1)

l1.insert(3,"L")

print (l1)

s1=''.join(l1)
print ("Modified string:", s1)
```

It will produce the following output –

```
original string: WORD
['W', 'O', 'R', 'L', 'D']
Modified string: WORLD
```

Using the Array Module

To modify a string, construct an array object using the Python standard library named array module. It will create an array of Unicode type from a string variable.

Example

In the below example, we are using array module to modify the specified string.

```
import array as ar

# initializing a string
s1="WORD"
print ("original string:", s1)

# converting it to an array
sar=ar.array('u', s1)

# inserting an element
sar.insert(3,"L")

# getting back the modified string
s1=sar.tounicode()
print ("Modified string:", s1)
```

It will produce the following output –

```
original string: WORD
Modified string: WORLD
```

Using the StringIO Class

Python's io module defines the classes to handle streams. The StringIO class represents a text stream using an in-memory text buffer. A StringIO object obtained from a string behaves like a File object. Hence we can perform read/write operations on it. The getvalue() method of StringIO class returns a string.

Example

Let us use the above discussed principle in the following program to modify a string.

```
import io

s1="WORD"
print ("original string:", s1)
```

```
sio=io.StringIO(s1)
sio.seek(3)
sio.write("LD")
s1=sio.getvalue()

print ("Modified string:", s1)
```

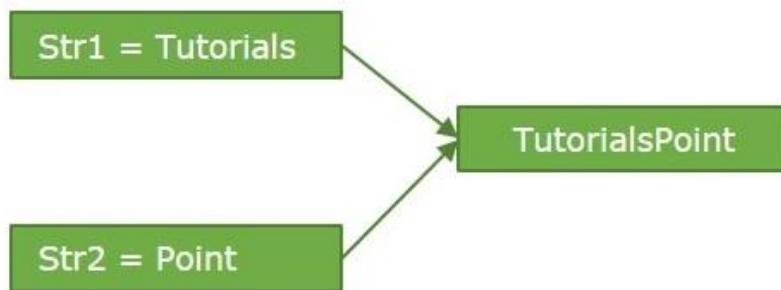
It will produce the following output –

```
original string: WORD
Modified string: WORLD
```

57. Python - String Concatenation

Concatenate Strings in Python

String concatenation in Python is the operation of joining two or more strings together. The result of this operation will be a new string that contains the original strings. The diagram below shows a general string concatenation operation –



In Python, there are numerous ways to concatenate strings. We are going to discuss the following –

- Using '+' operator
- Concatenating String with space
- Using multiplication operator
- Using '+' and '*' operators together

String Concatenation using '+' operator

The "+" operator is well-known as an addition operator, returning the sum of two numbers. However, the "+" symbol acts as string concatenation operator in Python. It works with two string operands, and results in the concatenation of the two.

The characters of the string on the right of plus symbol are appended to the string on its left. Result of concatenation is a new string.

Example

The following example shows string concatenation operation in Python using + operator.

```
str1="Hello"  
str2="World"  
print ("String 1:",str1)  
print ("String 2:",str2)  
str3=str1+str2  
print("String 3:",str3)
```

It will produce the following output –

```
String 1: Hello
String 2: World
String 3: HelloWorld
```

Concatenating String with space

To insert a whitespace between two strings, we can use a third empty string.

Example

In the below example, we are inserting space between two strings while concatenation.

```
str1="Hello"
str2="World"
blank=" "
print ("String 1:",str1)
print ("String 2:",str2)
str3=str1+blank+str2
print("String 3:",str3)
```

It will produce the following output –

```
String 1: Hello
String 2: World
String 3: Hello World
```

String Concatenation by Multiplying

Another symbol *, which we normally use for multiplication of two numbers, can also be used with string operands. Here, * acts as a repetition operator in Python. One of the operands must be an integer, and the second a string. The integer operand specifies the number of copies of the string operand to be concatenated.

In this example, the * operator concatenates multiple copies of the string.

```
newString = "Hello" * 3
print(newString)
```

The above code will produce the following output –

```
HelloHelloHello
```

String Concatenation with '+' and '*' Operators

Both the repetition operator (*) and the concatenation operator (+), can be used in a single expression to concatenate strings. The "*" operator has a higher precedence over the "+" operator.

Example

In the below example, we are concatenating strings using the + and * operator together.

```
str1="Hello"  
str2="World"  
print ("String 1:",str1)  
print ("String 2:",str2)  
str3=str1+str2*3  
print("String 3:",str3)  
str4=(str1+str2)*3  
print ("String 4:", str4)
```

To form str3 string, Python concatenates 3 copies of World first, and then appends the result to Hello

```
String 3: HelloWorldWorldWorld
```

In the second case, the strings str1 and str2 are inside parentheses, hence their concatenation takes place first. Its result is then replicated three times.

```
String 4: HelloWorldHelloWorldHelloWorld
```

Apart from + and *, no other arithmetic operators can be used with string

58. Python - String Formatting

String formatting in Python is the process of building a string representation dynamically by inserting the value of numeric expressions in an already existing string. Python's string concatenation operator doesn't accept a non-string operand. Hence, Python offers following string formatting techniques –

- Using % operator
- Using format() method of str class
- Using f-string
- Using String Template class

Using % operator

The "%" (modulo) operator often referred to as the string formatting operator. It takes a format string along with a set of variables and combine them to create a string that contain values of the variables formatted in the specified way.

Example

To insert a string into a format string using the "%" operator, we use "%s" as shown in the below example –

```
name = "Tutorialspoint"  
print("Welcome to %s!" % name)
```

It will produce the following output –

```
Welcome to Tutorialspoint!
```

Using format() method

It is a built-in method of str class. The format() method works by defining placeholders within a string using curly braces "{}". These placeholders are then replaced by the values specified in the method's arguments.

Example

In the below example, we are using format() method to insert values into a string dynamically.

```
str = "Welcome to {}"  
print(str.format("Tutorialspoint"))
```

On running the above code, it will produce the following output –

```
Welcome to Tutorialspoint
```

Using f-string

The f-strings, also known as formatted string literals, is used to embed expressions inside string literals. The "f" in f-strings stands for formatted and prefixing it with strings creates

an f-string. The curly braces "{}" within the string will then act as placeholders that is filled with variables, expressions, or function calls.

Example

The following example illustrates the working of f-strings with expressions.

```
item1_price = 2500
item2_price = 300
total = f'Total: {item1_price + item2_price}'
print(total)
```

The output of the above code is as follows –

```
Total: 2800
```

Using String Template class

The String Template class belongs to the string module and provides a way to format strings by using placeholders. Here, placeholders are defined by a dollar sign (\$) followed by an identifier.

Example

The following example shows how to use Template class to format strings.

```
from string import Template

# Defining template string
str = "Hello and Welcome to $name !"

# Creating Template object
templateObj = Template(str)

# now provide values
new_str = templateObj.substitute(name="Tutorialspoint")
print(new_str)
```

It will produce the following output –

```
Hello and Welcome to Tutorialspoint !
```

59. Python - Escape Characters

Escape Character

An escape character is a character followed by a backslash (\). It tells the Interpreter that this escape character (sequence) has a special meaning. For instance, \n is an escape sequence that represents a newline. When Python encounters this sequence in a string, it understands that it needs to start a new line.

Unless an 'r' or 'R' prefix is present, escape sequences in string and byte literals are interpreted according to rules similar to those used by Standard C. In Python, a string becomes a raw string if it is prefixed with "r" or "R" before the quotation symbols. Hence 'Hello' is a normal string whereas r'Hello' is a raw string.

Example

In the below example, we are practically demonstrating raw and normal string.

```
# normal string
normal = "Hello"
print (normal)

# raw string
raw = r"Hello"
print (raw)
```

Output of the above code is shown below –

```
Hello
Hello
```

In normal circumstances, there is no difference between the two. However, when the escape character is embedded in the string, the normal string actually interprets the escape sequence, whereas the raw string doesn't process the escape character.

Example

In the following example, when a normal string is printed the escape character '\n' is processed to introduce a newline. However, because of the raw string operator 'r' the effect of escape character is not translated as per its meaning.

```
normal = "Hello\nWorld"
print (normal)

raw = r"Hello\nWorld"
print (raw)
```

On running the above code, it will print the following result –

```
Hello
World
Hello\nWorld
```

Escape Characters in Python

The following table shows the different escape characters used in Python -

Sr.No	Escape Sequence & Meaning
1	\<newline> Backslash and newline ignored
2	\\ Backslash (\)
3	\' Single quote ('')
4	\" Double quote (")
5	\a ASCII Bell (BEL)
6	\b ASCII Backspace (BS)
7	\f ASCII Formfeed (FF)
8	\n ASCII Linefeed (LF)
9	\r ASCII Carriage Return (CR)
10	\t ASCII Horizontal Tab (TAB)
11	\v ASCII Vertical Tab (VT)
12	\ooo Character with octal value ooo
13	\xhh Character with hex value hh

Escape Characters Example

The following code shows the usage of escape sequences listed in the above table –

```
# ignore \
s = 'This string will not include \
backslashes or newline characters.'
print (s)

# escape backslash
```

```
s=s = 'The \\character is called backslash'
print (s)

# escape single quote
s='Hello \'Python\''
print (s)

# escape double quote
s="Hello \"Python\""
print (s)

# escape \b to generate ASCII backspace
s='Hel\blo'
print (s)

# ASCII Bell character
s='Hello\a'
print (s)

# newline
s='Hello\nPython'
print (s)

# Horizontal tab
s='Hello\tPython'
print (s)

# form feed
s= "hello\fworld"
print (s)

# Octal notation
s="\101"
print(s)

# Hexadecimal notation
```

```
s="\x41"  
print (s)
```

It will produce the following output –

```
This string will not include backslashes or newline characters.  
The \character is called backslash  
Hello 'Python'  
Hello "Python"  
Hello  
Hello  
Hello  
Python  
Hello Python  
hello  
world  
A  
A
```

60. Python - String Methods

Python's built-in str class defines different methods. They help in manipulating strings. Since string is an immutable object, these methods return a copy of the original string, performing the respective processing on it. The string methods can be classified in following categories –

Case Conversion Methods

This category of built-in methods of Python's str class deal with the conversion of alphabet characters in the string object. Following methods fall in this category –

Sr.No.	Method & Description
1	capitalize() Capitalizes first letter of string
2	casefold() Converts all uppercase letters in string to lowercase. Similar to lower(), but works on UNICODE characters also
3	lower() Converts all uppercase letters in string to lowercase.
4	swapcase() Inverts case for all letters in string.
5	title() Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
6	upper() Converts lowercase letters in string to uppercase.

Alignment Methods

Following methods in the str class control the alignment of characters within the string object.

Sr.No.	Methods & Description
1	center(width, fillchar) Returns a string padded with fillchar with the original string centered to a total of width columns.
2	ljust(width[, fillchar]) Returns a space-padded string with the original string left-justified to a total of width columns.

	<code>rjust(width[, fillchar])</code>
3	Returns a space-padded string with the original string right-justified to a total of width columns.
4	<code>expandtabs(tabsize = 8)</code> Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
5	<code>zfill (width)</code> Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

Split and Join Methods

Python has the following methods to perform split and join operations –

Sr.No.	Method & Description
1	<code>lstrip()</code> Removes all leading whitespace in string.
2	<code>rstrip()</code> Removes all trailing whitespace of string.
3	<code>strip()</code> Performs both lstrip() and rstrip() on string
4	<code>rsplit()</code> Splits the string from the end and returns a list of substrings
5	<code>split()</code> Splits string according to delimiter (space if not provided) and returns list of substrings.
6	<code>splitlines()</code> Splits string at NEWLINEs and returns a list of each line with NEWLINEs removed.
7	<code>partition()</code> Splits the string in three string tuple at the first occurrence of separator
8	<code>rpartition()</code> Splits the string in three string tuple at the last occurrence of separator

	<u>join()</u>
9	Concatenates the string representations of elements in sequence into a string, with separator string.
10	<u>removeprefix()</u> Returns a string after removing the prefix string
11	<u>removesuffix()</u> Returns a string after removing the suffix string

Boolean String Methods

Following methods in str class return True or False.

Sr.No.	Methods & Description
1	<u>isalnum()</u> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
2	<u>isalpha()</u> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
3	<u>isdigit()</u> Returns true if the string contains only digits and false otherwise.
4	<u>islower()</u> Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
5	<u>isnumeric()</u> Returns true if a unicode string contains only numeric characters and false otherwise.
6	<u>isspace()</u> Returns true if string contains only whitespace characters and false otherwise.
7	<u>istitle()</u> Returns true if string is properly "titlecased" and false otherwise.
8	<u>isupper()</u> Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
9	<u>isascii()</u> Returns True if all the characters in the string are from the ASCII character set
10	<u>isdecimal()</u>

	Checks if all the characters are decimal characters
11	isidentifier() Checks whether the string is a valid Python identifier
12	isprintable() Checks whether all the characters in the string are printable

Find and Replace Methods

Following are the Find and Replace methods in Python –

Sr.No.	Method & Description
1	count(sub, beg ,end) Counts how many times sub occurs in string or in a substring of string if starting index beg and ending index end are given.
2	find(sub, beg, end) Determine if sub occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
3	index(sub, beg, end) Same as find(), but raises an exception if str not found.
4	replace(old, new [, max]) Replaces all occurrences of old in string with new or at most max occurrences if max given.
5	rfind(sub, beg, end) Same as find(), but search backwards in string.
6	rindex(sub, beg, end) Same as index(), but search backwards in string.
7	startswith(sub, beg, end) Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring sub; returns true if so and false otherwise.
8	endswith(suffix, beg, end)

Translation Methods

Following are the Translation methods of the string –

Sr.No.	Method & Description
--------	----------------------

	<u>maketrans()</u>
1	Returns a translation table to be used in translate function.
	<u>translate(table, deletechars="")</u>
2	Translates string according to translation table str(256 chars), removing those in the del string.

61. Python - String Exercises

Example 1

Python program to find number of vowels in a given string.

```
mystr = "All animals are equal. Some are more equal"
vowels = "aeiou"
count=0
for x in mystr:
    if x.lower() in vowels: count+=1
print ("Number of Vowels:", count)
```

It will produce the following output –

```
Number of Vowels: 18
```

Example 2

Python program to convert a string with binary digits to integer.

```
mystr = '10101'

def strtoint(mystr):
    for x in mystr:
        if x not in '01': return "Error. String with non-binary characters"
    num = int(mystr, 2)
    return num
print ("binary:{} integer: {}".format(mystr,strtoint(mystr)))
```

It will produce the following output –

```
binary:10101 integer: 21
Change mystr to '10, 101'

binary:10,101 integer: Error. String with non-binary characters
```

Example 3

Python program to drop all digits from a string.

```
digits = [str(x) for x in range(10)]
mystr = 'He12llo, Py00th55on!'
```

```
chars = []
for x in mystr:
    if x not in digits:
        chars.append(x)
newstr = ''.join(chars)
print (newstr)
```

It will produce the following output –

```
Hello, Python!
```

Exercise Programs

- Python program to sort the characters in a string
- Python program to remove duplicate characters from a string
- Python program to list unique characters with their count in a string
- Python program to find number of words in a string
- Python program to remove all non-alphabetic characters from a string

Python Lists

62. Python - Lists

Python Lists

List is one of the built-in data types in Python. A Python list is a sequence of comma separated items, enclosed in square brackets []. The items in a Python list need not be of the same data type.

Following are some examples of Python lists –

```
list1 = ["Rohan", "Physics", 21, 69.75]
list2 = [1, 2, 3, 4, 5]
list3 = ["a", "b", "c", "d"]
list4 = [25.50, True, -55, 1+2j]
```

List is an ordered collection of items. Each item in a list has a unique position index, starting from 0.

A list in Python is similar to an array in C, C++ or Java. However, the major difference is that in C/C++/Java, the array elements must be of same type. On the other hand, Python lists may have objects of different data types.

A Python List is mutable. Any item from the list can be accessed using its index, and can be modified. One or more objects from the list can be removed or added. A list may have same item at more than one index positions.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7];
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result –

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example –

```
list = ['physics', 'chemistry', 1997, 2000];
print ("Value available at index 2 : ")
print (list[2])
list[2] = 2001;
print ("New value available at index 2 : ")
print (list[2])
```

When the above code is executed, it produces the following result –

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the `del` statement if you know exactly which element(s) you are deleting or the `remove()` method if you do not know. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
print (list1)
del list1[2];
print ("After deleting value at index 2 : ")
print (list1)
```

When the above code is executed, it produces following result –

```
['physics', 'chemistry', 1997, 2000]
```

After deleting value at index 2 :

```
['physics', 'chemistry', 2000]
```

Note – `remove()` method is discussed in subsequent section.

Python List Operations

In Python, List is a sequence. Hence, we can concatenate two lists with "+" operator and concatenate multiple copies of a list with "*" operator. The membership operators "in" and "not in" work with list object.

Python Expression	Results	Description
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	<code>TRUE</code>	Membership

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Python List Methods

Python includes following list methods –

Sr.No.	Methods with Description
1	<u>list.append(obj)</u> Appends object obj to list.
2	<u>list.clear()</u> Clears the contents of list.
3	<u>list.copy()</u> Returns a copy of the list object.
4	<u>list.count(obj)</u> Returns count of how many times obj occurs in list
5	<u>list.extend(seq)</u> Appends the contents of seq to list
6	<u>list.index(obj)</u> Returns the lowest index in list that obj appears
7	<u>list.insert(index, obj)</u> Inserts object obj into list at offset index
8	<u>list.pop(obj=list[-1])</u> Removes and returns last object or obj from list
9	<u>list.remove(obj)</u> Removes object obj from list
10	<u>list.reverse()</u> Reverses objects of list in place
11	<u>list.sort([func])</u> Sorts objects of list, use compare func if given

Built-in Functions with Lists

Following are the built-in functions we can use with lists –

Sr.No.	Function with Description
1	<u>cmp(list1, list2)</u> Compares elements of both lists.
2	<u>len(list)</u> Gives the total length of the list.
3	<u>max(list)</u> Returns item from the list with max value.
4	<u>min(list)</u> Returns item from the list with min value.
5	<u>list(seq)</u> Converts a tuple into list.

63. Python - Access List Items

Access List Items

In Python, a list is a sequence of elements or objects, i.e. an ordered collection of objects. Similar to arrays, each element in a list corresponds to an index.

To access the values within a list, we need to use the square brackets "[]" notation and, specify the index of the elements we want to retrieve.

The index starts from 0 for the first element and increments by one for each subsequent element. Index of the last item in the list is always "Length-1", where "Length" represents the total number of items in the list.

In addition to this, Python provides various other ways to access list items such as slicing, negative indexing, extracting a sublist from a list etc. Let us go through this one-by-one –

Accessing List Items with Indexing

As discussed above to access the items in a list using indexing, just specify the index of the element with in the square brackets ("[]") as shown below –

```
mylist[4]
```

Example

Following is the basic example to access list items –

```
list1 = ["Rohan", "Physics", 21, 69.75]
list2 = [1, 2, 3, 4, 5]

print ("Item at 0th index in list1: ", list1[0])
print ("Item at index 2 in list2: ", list2[2])
```

It will produce the following output –

```
Item at 0th index in list1: Rohan
Item at index 2 in list2: 3
```

Access List Items with Negative Indexing

Negative indexing in Python is used to access elements from the end of a list, with -1 referring to the last element, -2 to the second last, and so on.

We can also access list items with negative indexing by using negative integers to represent positions from the end of the list.

Example

In the following example, we are accessing list items with negative indexing –

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]

print ("Item at 0th index in list1: ", list1[-1])
print ("Item at index 2 in list2: ", list2[-3])
```

We get the output as shown below –

```
Item at 0th index in list1: d
Item at index 2 in list2: True
```

Access List Items with Slice Operator

The slice operator in Python is used to fetch one or more items from the list. We can access list items with the slice operator by specifying the range of indices we want to extract. It uses the following syntax –

```
[start:stop]
```

Where,

- start is the starting index (inclusive).
- stop is the ending index (exclusive).

If we do not provide any indices, the slice operator defaults to starting from index 0 and stopping at the last item in the list.

Example

In the following example, we are retrieving sublist from index 1 to last in "list1" and index 0 to 1 in "list2", and retrieving all elements in "list3" –

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]
list3 = ["Rohan", "Physics", 21, 69.75]

print ("Items from index 1 to last in list1: ", list1[1:])
print ("Items from index 0 to 1 in list2: ", list2[:2])
print ("Items from index 0 to index last in list3", list3[:])
```

Following is the output of the above code –

```
Items from index 1 to last in list1:  ['b', 'c', 'd']
Items from index 0 to 1 in list2:  [25.5, True]
Items from index 0 to index last in list3 ['Rohan', 'Physics', 21, 69.75]
```

Access Sub List from a List

A sublist is a part of a list that consists of a consecutive sequence of elements from the original list. We can access a sublist from a list by using the slice operator with appropriate start and stop indices.

Example

In this example, we are fetching sublist from index "1 to 2" in "list1" and index "0 to 1" in "list2" using slice operator –

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]

print ("Items from index 1 to 2 in list1: ", list1[1:3])
print ("Items from index 0 to 1 in list2: ", list2[0:2])
```

The output obtained is as follows –

```
Items from index 1 to 2 in list1: ['b', 'c']
Items from index 0 to 1 in list2: [25.5, True]
```

64. Python - Change List Items

Change List Items

List is a mutable data type in Python. It means, the contents of list can be modified in place, after the object is stored in the memory. You can assign a new value at a given index position in the list

Syntax

```
list1[i] = newvalue
```

Example

In the following code, we change the value at index 2 of the given list.

```
list3 = [1, 2, 3, 4, 5]
print ("Original list ", list3)
list3[2] = 10
print ("List after changing value at index 2: ", list3)
```

It will produce the following output –

```
Original list [1, 2, 3, 4, 5]
List after changing value at index 2: [1, 2, 10, 4, 5]
```

Change Consecutive List Items

You can replace more consecutive items in a list with another sublist.

Example

In the following code, items at index 1 and 2 are replaced by items in another sublist.

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['Y', 'Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following output –

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'Y', 'Z', 'd']
```

Change a Range of List Items

If the source sublist has more items than the slice to be replaced, the extra items in the source will be inserted. Take a look at the following code –

Example

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['X','Y', 'Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following output –

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'X', 'Y', 'Z', 'd']
```

Example

If the sublist with which a slice of original list is to be replaced, has lesser items, the items with match will be replaced and rest of the items in original list will be removed.

In the following code, we try to replace "b" and "c" with "Z" (one less item than items to be replaced). It results in Z replacing b and c removed.

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following output –

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'Z', 'd']
```

65. Python - Add List Items

Add List Items

Adding list items in Python implies inserting new elements into an existing list. Lists are mutable, meaning they can be modified after creation, allowing for the addition, removal, or modification of their elements.

Adding items in a list typically refers to appending new elements to the end of the list, inserting them at specific positions within the list, or extending the list with elements from another iterable object.

We can add list items in Python using various methods such as `append()`, `extend()` and `insert()`. Let us explore through all these methods in this tutorial.

Adding List Items Using `append()` Method

The `append()` method in Python is used to add a single element to the end of a list.

We can add list items using the `append()` method by specifying the element we want to add within the parentheses, like `my_list.append(new_item)`, which adds `new_item` to the end of `my_list`.

Example

In the following example, we are adding an element "e" to the end of the list "list1" using the `append()` method –

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list1.append('e')
print ("List after appending: ", list1)
```

Output

Following is the output of the above code –

```
Original list: ['a', 'b', 'c', 'd']
List after appending: ['a', 'b', 'c', 'd', 'e']
```

Adding List Items Using `insert()` Method

The `insert()` method in Python is used to add an element at a specified index (position) within a list, shifting existing elements to accommodate the new one.

We can add list items using the `insert()` method by specifying the index position where we want to insert the new item and the item itself within the parentheses, like `my_list.insert(index, new_item)`.

Example

In this example, we have an original list containing various items. We use the `insert()` method to add new elements to the list at specific positions –

```
list1 = ["Rohan", "Physics", 21, 69.75]

list1.insert(2, 'Chemistry')
print ("List after appending: ", list1)

list1.insert(-1, 'Pass')
print ("List after appending: ", list1)
```

After appending 'Chemistry' to the list, we get the following output –

```
List after appending: ['Rohan', 'Physics', 'Chemistry', 21, 69.75]
```

Then, by inserting 'Pass' at the index "-1", which originally referred to 69.75, we get –

```
List after appending: ['Rohan', 'Physics', 'Chemistry', 21, 'Pass', 69.75]
```

We can see that "Pass" is not inserted at the updated index "-1", but the previous index "-1". This behavior is because when appending or inserting items into a list, Python does not dynamically update negative index positions.

Adding List Items Using extend() Method

The extend() method in Python is used to add multiple elements from an iterable (such as another list) to the end of a list.

We can add list items using the extend() method by passing another iterable containing the elements we want to add, like my_list.extend(iterable), which appends each element from the iterable to the end of my_list.

Example

In the below example, we are using the extend() method to add the elements from "another_list" to the end of "list1" –

```
# Original list
list1 = [1, 2, 3]
# Another list to extend with
another_list = [4, 5, 6]

list1.extend(another_list)
print("Extended list:", list1)
```

Output

Output of the above code is as follows –

```
Extended list: [1, 2, 3, 4, 5, 6]
```

66. Python - Remove List Items

Removing List Items

Removing list items in Python implies deleting elements from an existing list. Lists are ordered collections of items, and sometimes you need to remove certain elements from them based on specific criteria or indices. When we remove list items, we are reducing the size of the list or eliminating specific elements.

We can remove list items in Python using various methods such as `remove()`, `pop()` and `clear()`. Additionally, we can use the `del` statement to remove items at a specific index. Let us explore through all these methods in this tutorial.

Remove List Item Using `remove()` Method

The `remove()` method in Python is used to remove the first occurrence of a specified item from a list.

We can remove list items using the `remove()` method by specifying the value we want to remove within the parentheses, like `my_list.remove(value)`, which deletes the first occurrence of `value` from `my_list`.

Example

In the following example, we are deleting the element "Physics" from the list "list1" using the `remove()` method –

```
list1 = ["Rohan", "Physics", 21, 69.75]
print ("Original list: ", list1)

list1.remove("Physics")
print ("List after removing: ", list1)
```

It will produce the following output –

```
Original list: ['Rohan', 'Physics', 21, 69.75]
List after removing: ['Rohan', 21, 69.75]
```

Remove List Item Using `pop()` Method

The `pop()` method in Python is used to remove and return the last element from a list if no index is specified. It can also remove and return the element at a specified index, altering the original list.

We can remove list items using the `pop()` method by calling it without any arguments `my_list.pop()`, which removes and returns the last item from `my_list`, or by providing the index of the item we want to remove `my_list.pop(index)`, which removes and returns the item at that index.

Example

The following example shows how you can use the `pop()` method to remove list items –

```
list2 = [25.50, True, -55, 1+2j]
print ("Original list: ", list2)
list2.pop(2)
print ("List after popping: ", list2)
```

We get the output as shown below –

```
Original list: [25.5, True, -55, (1+2j)]
List after popping: [25.5, True, (1+2j)]
```

Remove List Item Using `clear()` Method

The `clear()` method in Python is used to remove all elements from a list, leaving it empty.

We can remove all list items using the `clear()` method by calling it on the list object like `my_list.clear()`, which empties `my_list`, leaving it with no elements.

Example

In this example, we are using the `clear()` method to remove all elements from the list "my_list" –

```
my_list = [1, 2, 3, 4, 5]

# Clearing the list
my_list.clear()

print("Cleared list:", my_list)
```

Output of the above code is as follows –

```
Cleared list: []
```

Remove List Item Using `del` Keyword

The `del` keyword in Python is used to delete element either at a specific index or a slice of indices from memory.

We can remove list items using the `del` keyword by specifying the index or slice of the items we want to delete, like `del my_list[index]` to delete a single item or `del my_list[start:stop]` to delete a range of items.

Example

In the below example, we are using the "`del`" keyword to delete an element at the index "2" from the list "list1" –

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
del list1[2]
```

```
print ("List after deleting: ", list1)
```

The result produced is as follows –

```
Original list: ['a', 'b', 'c', 'd']
List after deleting: ['a', 'b', 'd']
```

Example

In here, we are deleting a series of consecutive items from a list with the slicing operator –

```
list2 = [25.50, True, -55, 1+2j]
print ("List before deleting: ", list2)
del list2[0:2]
print ("List after deleting: ", list2)
```

It will produce the following output –

```
List before deleting: [25.5, True, -55, (1+2j)]
List after deleting: [-55, (1+2j)]
```

67. Python - Loop Lists

Loop Through List Items

Looping through list items in Python refers to iterating over each element within a list. We do so to perform the desired operations on each item. These operations include list modification, conditional operations, string manipulation, data analysis, etc.

Python provides various methods for looping through list items, with the most common being the for loop. We can also use the while loop to iterate through list items, although it requires additional handling of the loop control variable explicitly i.e. an index.

Loop Through List Items with For Loop

A for loop in Python is used to iterate over a sequence (like a list, tuple, dictionary, string, or range) or any other iterable object. It allows you to execute a block of code repeatedly for each item in the sequence.

In a for loop, you can access each item in a sequence using a variable, allowing you to perform operations or logic based on that item's value. We can loop through list items using for loop by iterating over each item in the list.

Syntax

Following is the basic syntax to loop through items in a list using a for loop in Python –

```
for item in list:  
    # Code block to execute
```

Example

In the following example, we are using a for loop to iterate through each element in the list "lst" and retrieving each element followed by a space on the same line –

```
lst = [25, 12, 10, -21, 10, 100]  
for num in lst:  
    print (num, end = ' ')
```

Output

Following is the output of the above code –

```
25 12 10 -21 10 100
```

Loop Through List Items with While Loop

A while loop in Python is used to repeatedly execute a block of code as long as a specified condition evaluates to "True".

We can loop through list items using the while loop by initializing an index variable, then iterating through the list using the index variable and incrementing it until reaching the end of the list.

An index variable is used within a Loop to keep track of the current position or index in a sequence, such as a list or array. It is generally initialized before the Loop and updated within the Loop to iterate over the sequence.

Syntax

Following is the basic syntax for looping through items in a list using a while loop in Python –

```
while condition:  
    # Code block to execute
```

Example

In the below example, we iterate through each item in the list "my_list" using a while loop. We use an index variable "index" to access each item sequentially, incrementing it after each iteration to move to the next item –

```
my_list = [1, 2, 3, 4, 5]  
index = 0  
  
while index < len(my_list):  
    print(my_list[index])  
    index += 1
```

Output

Output of the above code is as follows –

```
1  
2  
3  
4  
5
```

Loop Through List Items with Index

An index is a numeric value representing the position of an element within a sequence, such as a list, starting from 0 for the first element.

We can loop through list items using index by iterating over a range of indices corresponding to the length of the list and accessing each element using the index within the loop.

Example

This example initializes a list "lst" with integers and creates a range of indices corresponding to the length of the list. Then, it iterates over each index in the range and prints the value at that index in the list "lst" –

```
lst = [25, 12, 10, -21, 10, 100]
indices = range(len(lst))
for i in indices:
    print ("lst[{}]: {}".format(i), lst[i])
```

Output

We get the output as shown below –

```
lst[0]: 25
lst[1]: 12
lst[2]: 10
lst[3]: -21
lst[4]: 10
lst[5]: 100
```

Iterate using List Comprehension

A list comprehension in Python is a concise way to create lists by applying an expression to each element of an iterable. These expressions can be arithmetic operations, function calls, conditional expressions etc.

We can iterate using list comprehension by specifying the expression and the iterable (like a list, tuple, dictionary, string, or range). Following is the syntax –

```
[expression for item in iterable]
```

This applies the expression to each item in the iterable and creates a list of results.

Example

In this example, we use list comprehension to iterate through each number in a list of numbers, square each one, and store the squared result in the new list "squared_numbers" –

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [num ** 2 for num in numbers]
print (squared_numbers)
```

Output

We get the output as shown below –

```
[1, 4, 9, 16, 25]
```

Iterate using the enumerate() Function

The enumerate() function in Python is used to iterate over an iterable object while also providing the index of each element.

We can iterate using the enumerate() function by applying it to the iterable. Following is the syntax –

```
for index, item in enumerate(iterable):
```

This provides both the index and item of each element in the iterable during iteration

Example

In the following example, we are using the enumerate() function to iterate through a list "fruits" and retrieve each fruit along with its corresponding index –

```
fruits = ["apple", "banana", "cherry"]  
for index, fruit in enumerate(fruits):  
    print(index, fruit)
```

Output

We get the output as shown below –

```
0 apple  
1 banana  
2 cherry
```

68. Python - List Comprehension

List Comprehension in Python

A list comprehension is a concise way to create lists. It is similar to set builder notation in mathematics. It is used to define a list based on an existing iterable object, such as a list, tuple, or string, and apply an expression to each element in the iterable.

Syntax of Python List Comprehension

The basic syntax of list comprehension is –

```
new_list = [expression for item in iterable if condition]
```

Where,

- **expression** is the operation or transformation to apply to each item in the iterable.
- **item** is the variable representing each element in the iterable.
- **iterable** is the sequence of elements to iterate over.
- **condition (optional)** is an expression that filters elements based on a specified condition.

Example of Python List Comprehension

Suppose we want to convert all the letters in the string "hello world" to their upper-case form. Using list comprehension, we iterate through each character, check if it is a letter, and if so, convert it to uppercase, resulting in a list of uppercase letters –

```
string = "hello world"
uppercase_letters = [char.upper() for char in string if char.isalpha()]
print(uppercase_letters)
```

The result obtained is displayed as follows –

```
['H', 'E', 'L', 'L', 'O', 'W', 'O', 'R', 'L', 'D']
```

List Comprehensions and Lambda

In Python, lambda is a keyword used to create anonymous functions. An anonymous function is a function defined without a name. These functions are created using the lambda keyword followed by a comma-separated list of arguments, followed by a colon :, and then the expression to be evaluated.

We can use list comprehension with lambda by applying the lambda function to each element of an iterable within the comprehension, generating a new list.

Example

In the following example, we are using list comprehension with a lambda function to double each element in a given list "original_list". We iterate over each element in the "original_list" and apply the lambda function to double it –

```
original_list = [1, 2, 3, 4, 5]
doubled_list = [(lambda x: x * 2)(x) for x in original_list]
print(doubled_list)
```

Following is the output of the above code –

```
[2, 4, 6, 8, 10]
```

Nested Loops in Python List Comprehension

A nested loop in Python is a loop inside another loop, where the inner loop is executed multiple times for each iteration of the outer loop.

We can use nested loops in list comprehension by placing one loop inside another, allowing for concise creation of lists from multiple iterations.

Example

In this example, all combinations of items from two lists in the form of a tuple are added in a third list object –

```
list1=[1,2,3]
list2=[4,5,6]
CombLst=[(x,y) for x in list1 for y in list2]
print (CombLst)
```

It will produce the following output –

```
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]
```

Conditionals in Python List Comprehension

Conditionals in Python refer to the use of statements like "if", "elif", and "else" to control the flow of a code based on certain conditions. They allow you to execute different blocks of code depending on whether a condition evaluates to "True" or "False".

We can use conditionals in list comprehension by including them after the iterable and before the loop, which filters elements from the iterable based on the specified condition while generating the list.

Example

The following example uses conditionals within a list comprehension to generate a list of even numbers from 1 to 20 –

```
list1=[x for x in range(1,21) if x%2==0]
print (list1)
```

We get the output as follows –

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

List Comprehensions vs For Loop

List comprehensions and for loops are both used for iteration, but they differ in terms of syntax and usage.

List comprehensions are like shortcuts for creating lists in Python. They let you generate a new list by applying an operation to each item in an existing list.

For loop, on the other hand, is a control flow statement used to iterate over elements of an iterable one by one, executing a block of code for each element.

List comprehensions are often preferred for simpler operations, while for Loops offer more flexibility for complex tasks.

Example Using For Loop

Suppose we want to separate each letter in a string and put all non-vowel letters in a list object. We can do it by a for loop as shown below –

```
chars=[]
for ch in 'TutorialsPoint':
    if ch not in 'aeiou':
        chars.append(ch)
print (chars)
```

The chars list object is displayed as follows –

```
['T', 't', 'r', 'l', 's', 'P', 'n', 't']
```

Example Using List Comprehension

We can easily get the same result by a list comprehension technique. A general usage of list comprehension is as follows –

```
listObj = [x for x in iterable]
```

Applying this, chars list can be constructed by the following statement –

```
chars = [ char for char in 'TutorialsPoint' if char not in 'aeiou']
print (chars)
```

The chars list will be displayed as before –

```
['T', 't', 'r', 'l', 's', 'P', 'n', 't']
```

Example

The following example uses list comprehension to build a list of squares of numbers between 1 to 10 –

```
squares = [x*x for x in range(1,11)]
print (squares)
```

The squares list object is –

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Advantages of List Comprehension

Following are the advantages of using list comprehension –

- **Conciseness** – List comprehensions are more concise and readable compared to traditional for loops, allowing you to create lists with less code.
- **Efficiency** – List comprehensions are generally faster and more efficient than for loops because they are optimized internally by Python's interpreter.
- **Clarity** – List comprehensions result in clearer and more expressive code, making it easier to understand the purpose and logic of the operation being performed.
- **Reduced Chance of Errors** – Since list comprehensions are more compact, there is less chance of errors compared to traditional for loops, reducing the likelihood of bugs in your code.

69. Python - Sort Lists

Sorting Lists in Python

Sorting a list in Python is a way to arrange the elements of the list in either ascending or descending order based on a defined criterion, such as numerical or lexicographical order.

This can be achieved using the built-in `sorted()` function or by calling the `sort()` method on the list itself, both of which modify the original list or return a new sorted list depending on the method used.

Sorting Lists Using `sort()` Method

The python `sort()` method is used to sort the elements of a list in place. This means that it modifies the original list and does not return a new list.

Syntax

The syntax for using the `sort()` method is as follows –

```
list_name.sort(key=None, reverse=False)
```

Where,

- **list_name** is the name of the list to be sorted.
- **key (optional)** is a function that defines the sorting criterion. If provided, it is applied to each element of the list for sorting. Default is None.
- **reverse (optional)** is a boolean value. If True, the list will be sorted in descending order. If False (default), the list will be sorted in ascending order.

Example of Sorting List in Lexicographical Order

In the following example, we are using the `sort()` function to sort the items of the list alphanumerically –

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
print ("list before sort:", list1)
list1.sort()
print ("list after sort : ", list1)
```

It will produce the following output –

```
list before sort: ['physics', 'Biology', 'chemistry', 'maths']
list after sort : ['Biology', 'chemistry', 'maths', 'physics']
```

Example of Sorting List in Numerical Order

Here, we are using the `sort()` function to sort the given list in numerical order –

```
list2 = [10,16, 9, 24, 5]
print ("list before sort", list2)
```

```
list2.sort()
print ("list after sort : ", list2)
```

The output produced is as shown below –

```
list before sort [10, 16, 9, 24, 5]
list after sort : [5, 9, 10, 16, 24]
```

Sorting Lists Using sorted() Method

The `sorted()` function in Python is a built-in function used to sort the elements of an iterable (such as a list, tuple, or string) and returns a new sorted list, leaving the original iterable unchanged.

Syntax

The syntax for using the `sorted()` method is as follows –

```
sorted(iterable, key=None, reverse=False)
```

Where,

- **iterable** is the iterable (e.g., list, tuple, string) whose elements are to be sorted.
- **key (optional)** is a function that defines the sorting criterion. If provided, it is applied to each element of the iterable for sorting. Default is None.
- **reverse (optional)** is a boolean value. If True, the iterable will be sorted in descending order. If False (default), the iterable will be sorted in ascending order.

Example

In the following example, we are using the `sorted()` function to sort a list of numbers and retrieve a new sorted list –

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
# Sorting in descending order
sorted_numbers_desc = sorted(numbers, reverse=True)
print(sorted_numbers_desc)
```

Following is the output of the above code –

```
[9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
```

Sorting List Items with Callback Function

In Python, a callback function refers to a function that is passed as an argument to another function and is invoked or called within that function

We can sort list items with a callback function by using the `sorted()` function or `sort()` function in Python. Both of these functions allows us to specify a custom sorting criterion using the "key" parameter, which accepts a callback function. This callback function defines how the elements should be compared and sorted.

Example Using `str.lower()` as key Parameter

The `str.lower()` method in Python is used to convert all the characters in a string to lowercase. It returns a new string with all alphabetic characters converted to lowercase while leaving non-alphabetic characters unchanged.

In this example, we are passing the `str.lower()` method as an argument to the "key" parameter within the `sort()` function –

```
list1 = ['Physics', 'biology', 'Biomechanics', 'psychology']
print ("list before sort", list1)
list1.sort(key=str.lower)
print ("list after sort : ", list1)
```

It will produce the following output –

```
list before sort ['Physics', 'biology', 'Biomechanics', 'psychology']
list after sort : ['biology', 'Biomechanics', 'Physics', 'psychology']
```

Example using user-defined Function as key Parameter

We can also use a user-defined function as the key parameter in `sort()` method.

In this example, the `myfunction()` uses `%` operator to return the remainder, based on which the sorting is performed –

```
def myfunction(x):
    return x%10
list1 = [17, 23, 46, 51, 90]
print ("list before sort", list1)
list1.sort(key=myfunction)
print ("list after sort : ", list1)
```

It will produce the following output –

```
list before sort [17, 23, 46, 51, 90]
list after sort: [90, 51, 23, 46, 17]
```

70. Python - Copy Lists

Copying a List in Python

Copying a list in Python refers to creating a new list that contains the same elements as the original list. There are different methods for copying a list, including, using slice notation, the `list()` function, and using the `copy()` method.

Each method behaves differently in terms of whether it creates a shallow copy or a deep copy. Let us discuss about all of these deeply in this tutorial.

Shallow Copy on a Python List

A shallow copy in Python creates a new object, but instead of copying the elements recursively, it copies only the references to the original elements. This means that the new object is a separate entity from the original one, but if the elements themselves are mutable, changes made to those elements in the new object will affect the original object as well.

Example of Shallow Copy

Let us illustrate this with the following example –

```
import copy

# Original list
original_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Creating a shallow copy
shallow_copied_list = copy.copy(original_list)

# Modifying an element in the shallow copied list
shallow_copied_list[0][0] = 100

# Printing both lists
print("Original List:", original_list)
print("Shallow Copied List:", shallow_copied_list)
```

As you can see, even though we only modified the first element of the first sublist in the shallow copied list, the same change is reflected in the original list as well.

This is because a shallow copy only creates new references to the original objects, rather than creating copies of the objects themselves –

```
Original List: [[100, 2, 3], [4, 5, 6], [7, 8, 9]]
Shallow Copied List: [[100, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Deep Copy on a Python List

A deep copy in Python creates a completely new object and recursively copies all the objects referenced by the original object. This means that even nested objects within the original object are duplicated, resulting in a fully independent copy where changes made to the copied object do not affect the original object, and vice versa.

Example of Deep Copy

Let us illustrate this with the following example –

```
import copy

# Original list
original_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Creating a deep copy
deep_copied_list = copy.deepcopy(original_list)

# Modifying an element in the deep copied list
deep_copied_list[0][0] = 100

# Printing both lists
print("Original List:", original_list)
print("Deep Copied List:", deep_copied_list)
```

As you can see, when we modify the first element of the first sublist in the deep copied list, it does not affect the original list.

This is because a deep copy creates a new object and recursively copies all the nested objects, ensuring that the copied object is fully independent from the original one –

```
Original List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Deep Copied List: [[100, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Copying List Using Slice Notation

Slice notation in Python allows you to create a subsequence of elements from a sequence (like a list, tuple, or string) by specifying a start index, an end index, and an optional step size. The syntax for slice notation is as follows –

```
[start:end:step]
```

Where, start is the index where the slice starts, end is the index where the slice ends (exclusive), and step is the step size between elements.

We can copy a list using slice notation by specifying the entire range of indices of the original list. This effectively creates a new list with the same elements as the original list.

Any modifications made to the copied list will not affect the original list, and vice versa, because they are separate objects in memory.

Example

In this example, we are creating a slice of the "original_list", effectively copying all its elements into a new list "copied_list" –

```
# Original list
original_list = [1, 2, 3, 4, 5]
# Copying the list using slice notation
copied_list = original_list[1:4]
# Modifying the copied list
copied_list[0] = 100
# Printing both lists
print("Original List:", original_list)
print("Copied List:", copied_list)
```

We get the result as shown below –

```
Original List: [1, 2, 3, 4, 5]
Copied List: [100, 3, 4]
```

Copying List Using the list() Function

The `list()` function in Python is a built-in function used to create a new list object. It can accept an iterable (like another list, tuple, set, etc.) as an argument and create a new list containing the elements of that iterable. If no argument is provided, an empty list is created.

We can copy a list using the `list()` function by passing the original list as an argument. This will create a new list object containing the same elements as the original list.

Example

In the example below, we are creating a new list object "copied_list" containing the same elements as "original_list" using the `list()` function –

```
# Original list
original_list = [1, 2, 3, 4, 5]
# Copying the list using the list() constructor
copied_list = list(original_list)
# Printing both lists
print("Original List:", original_list)
print("Copied List:", copied_list)
```

Following is the output of the above code –

```
Original List: [1, 2, 3, 4, 5]
Copied List: [1, 2, 3, 4, 5]
```

Copying List using the copy() Function

In Python, the `copy()` function is used to create a shallow copy of a list or other mutable objects. This function is part of the `copy` module in Python's standard library.

We can copy a list using the `copy()` function by invoking it on the original list. This creates a new list object that contains the same elements as the original list.

Example

In the following example, we are using the `copy()` function to creates a new list object "copied_list" containing the same elements as "original_list" –

```
import copy
original_list = [1, 2, 3, 4, 5]
# Copying the list using the copy() function
copied_list = copy.copy(original_list)
print("Copied List:", copied_list)
```

Output of the above code is as shown below –

```
Copied List: [1, 2, 3, 4, 5]
```

71. Python - Join Lists

Join Lists in Python

Joining lists in Python refers to combining the elements of multiple lists into a single list. This can be achieved using various methods, such as concatenation, list comprehension, or using built-in functions like `extend()` or `+` operator.

Joining lists does not modify the original lists but creates a new list containing the combined elements.

Join Lists Using Concatenation Operator

The concatenation operator in Python, denoted by `+`, is used to join two sequences, such as strings, lists, or tuples, into a single sequence. When applied to lists, the concatenation operator joins the elements of the two (or more) lists to create a new list containing all the elements from both lists.

We can join a list using the concatenation operator by simply using the `+` symbol to concatenate the lists.

Example

In the following example, we are concatenating the elements of two lists "L1" and "L2", creating a new list "joined_list" containing all the elements from both lists –

```
# Two lists to be joined
L1 = [10, 20, 30, 40]
L2 = ['one', 'two', 'three', 'four']
# Joining the lists
joined_list = L1 + L2

# Printing the joined list
print("Joined List:", joined_list)
```

Following is the output of the above code –

```
Joined List: [10, 20, 30, 40, 'one', 'two', 'three', 'four']
```

Join Lists Using List Comprehension

List comprehension is a concise way to create lists in Python. It is used to generate new lists by applying an expression to each item in an existing iterable, such as a list, tuple, or range. The syntax for list comprehension is –

```
new_list = [expression for item in iterable]
```

This creates a new list where expression is evaluated for each item in the iterable.

We can join a list using list comprehension by iterating over multiple lists and appending their elements to a new list.

Example

In this example, we are joining two lists, L1 and L2, into a single list using list comprehension. The resulting list, joined_list, contains all elements from both L1 and L2 –

```
# Two lists to be joined
L1 = [36, 24, 3]
L2 = [84, 5, 81]

# Joining the lists using list comprehension
joined_list = [item for sublist in [L1, L2] for item in sublist]

# Printing the joined list
print("Joined List:", joined_list)
```

Output of the above code is as follows –

```
Joined List: [36, 24, 3, 84, 5, 81]
```

Join Lists Using append() Function

The append() function in Python is used to add a single element to the end of a list. This function modifies the original list by adding the element to the end of the list.

We can join a list using the append() function by iterating over the elements of one list and appending each element to another list.

Example

In the example below, we are appending elements from "list2" to "list1" using the append() function. We achieve this by iterating over "list2" and adding each element to "list1" –

```
# List to which elements will be appended
list1 = ['Fruit', 'Number', 'Animal']

# List from which elements will be appended
list2 = ['Apple', 5, 'Dog']

# Joining the lists using the append() function
for element in list2:
    list1.append(element)

# Printing the joined list
print("Joined List:", list1)
```

We get the output as shown below –

```
Joined List: ['Fruit', 'Number', 'Animal', 'Apple', 5, 'Dog']
```

Join Lists Using extend() Function

The Python `extend()` function is used to append elements from an iterable (such as another list) to the end of the list. This function modifies the original list in place, adding the elements of the iterable to the end of the list.

We can join a list using the `extend()` function by calling it on one list and passing another list (or any iterable) as an argument. This will append all the elements from the second list to the end of the first list.

Example

In the following example, we are extending "list1" by appending the elements of "list2" using the `extend()` function –

```
# List to be extended
list1 = [10, 15, 20]
# List to be added
list2 = [25, 30, 35]
# Joining the lists using the extend() function
list1.extend(list2)
# Printing the extended list
print("Extended List:", list1)
```

The output obtained is as shown below –

```
Extended List: [10, 15, 20, 25, 30, 35]
```

72. Python - List Methods

List is one of the fundamental data structures in Python. It provides a flexible way to store and manage a collection of items. It has several built-in methods that allow you to add, update, and delete items efficiently.

Lists in Python can contain items of different data types, including other lists, making them highly flexible to different scenarios. The list object includes several built-in methods that allow you to add, update, and delete items efficiently, as well as to perform various operations on the list's elements.

Python List Methods

The list methods enable you to manipulate lists easily and effectively, whether you are appending new items, removing existing ones, or even sorting and reversing the list. By using these built-in methods, you can work with lists in Python more effectively, allowing you to write more efficient and readable code.

Printing All the List Methods

To view all the available methods for lists, you can use the Python `dir()` function, which returns all the properties and functions related to an object. Additionally, you can use the Python `help()` function to get more detailed information about each method. For example:

```
print(dir([]))
print(help([].append))
```

The above code snippet provides a complete list of properties and functions related to the list class. It also demonstrates how to access detailed documentation for a specific method in your Python environment. Here is the output –

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Help on built-in function append:

append(object, /) method of builtins.list instance

    Append object to the end of the list.

(END)
```

Below, the built-in methods for lists in Python, which are categorized based on their functionality. Let's explore and understand the basic functionality of each method.

Methods to Add Elements to a List

The following are the methods specifically designed for adding new item/items into a list –

Sr.No.	Methods with Description
1	<code>list.append(obj)</code> Appends object obj to list.
2	<code>list.extend(seq)</code> Appends the contents of seq to list
3	<code>list.insert(index, obj)</code> Inserts object obj into list at offset index

Methods to Remove Elements from a List

The following are the methods specifically designed for removing items from a list –

Sr.No.	Methods with Description
1	<code>list.clear()</code> Clears all the contents of the list.
2	<code>list.pop(obj=list[-1])</code> Removes and returns the last object or the object at the specified index from the list.
3	<code>list.remove(obj)</code> Removes the first occurrence of object obj from the list.

Methods to Access Elements in a List

These are the methods used for finding or counting items in a list –

Sr.No.	Methods with Description
1	<code>list.index(obj)</code> Returns the lowest index in list that obj appears
2	<code>list.count(obj)</code> Returns count of how many times obj occurs in the list.

Copying and Ordering Methods

These are the methods used for creating copies and arranging items in a list –

Sr.No.	Methods with Description
1	<code>list.copy()</code> Returns a copy of the list object.
2	<code>list.sort([func])</code>

	Sorts the objects in the list in place, using a comparison function if provided.
3	Reverses the order of objects in the list in place. <u>list.reverse()</u>

73. Python - List Exercises

Python List Exercise 1

Python program to find unique numbers in a given list.

```
L1 = [1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2]
L2 = []
for x in L1:
    if x not in L2:
        L2.append(x)
print (L2)
```

It will produce the following output –

```
[1, 9, 6, 3, 4, 5, 2, 7, 8]
```

Python List Exercise 2

Python program to find sum of all numbers in a list.

```
L1 = [1, 9, 1, 6, 3, 4]
ttl = 0
for x in L1:
    ttl+=x
print ("Sum of all numbers Using loop:", ttl)
ttl = sum(L1)
print ("Sum of all numbers sum() function:", ttl)
```

It will produce the following output –

```
Sum of all numbers Using loop: 24
Sum of all numbers sum() function: 24
```

Python List Exercise 3

Python program to create a list of 5 random integers.

```
import random
L1 = []
for i in range(5):
    x = random.randint(0, 100)
```

```
L1.append(x)  
print (L1)
```

It will produce the following output –

```
[77, 3, 20, 91, 85]
```

Python Tuples

74. Python - Tuples

Tuple is one of the built-in data types in Python. It is a sequence of comma separated items, enclosed in parentheses (). The items in a Python tuple need not be of same data type.

Following are some examples of Python tuples –

```
tup1 = ("Rohan", "Physics", 21, 69.75)
tup2 = (1, 2, 3, 4, 5)
tup3 = ("a", "b", "c", "d")
tup4 = (25.50, True, -55, 1+2j)
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
tup1 = (50,);
```

Following are the points to be noted –

- In Python, tuple is a sequence data type. It is an ordered collection of items. Each item in the tuple has a unique position index, starting from 0.
- In C/C++/Java array, the array elements must be of same type. On the other hand, Python tuple may have objects of different data types.
- Python tuple and list both are sequences. One major difference between the two is, Python list is mutable, whereas tuple is immutable. Although any item from the tuple can be accessed using its index, and cannot be modified, removed or added.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print ("tup1[0]: ", tup1[0]);
print ("tup2[1:5]: ", tup2[1:5]);
```

When the above code is executed, it produces the following result –

```
tup1[0]: physics
tup2[1:5]: [2, 3, 4, 5]
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print (tup3);
```

When the above code is executed, it produces the following result –

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the `del` statement. For example –

```
tup = ('physics', 'chemistry', 1997, 2000);
print (tup);
del tup;
print ("After deleting tup : ");
print (tup);
```

This produces the following result. Note an exception raised, this is because after `del` tup tuple does not exist any more –

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print (tup);
NameError: name 'tup' is not defined
```

Python Tuple Operations

In Python, Tuple is a sequence. Hence, we can concatenate two tuples with + operator and concatenate multiple copies of a tuple with "*" operator. The membership operators "in" and "not in" work with tuple object.

Python Expression	Results	Description
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	TRUE	Membership

Even if there is only one object in a tuple, you must give a comma after it. Otherwise, it is treated as a string.

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples –

```
print ('abc', -4.24e93, 18+6.6j, 'xyz');
x, y = 1, 2;
print ("Value of x , y : ", x,y);
```

When the above code is executed, it produces the following result –

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Functions with Tuples

Following are the built-in functions we can use with tuples –

Sr.No.	Function with Description
1	cmp(tuple1, tuple2)

	Compares elements of both tuples.
2	<u>len(tuple)</u>
	Gives the total length of the tuple.
3	<u>max(tuple)</u>
	Returns item from the tuple with max value.
4	<u>min(tuple)</u>
	Returns item from the tuple with min value.
5	<u>tuple(seq)</u>
	Converts a list into tuple.

75. Python - Access Tuple Items

Access Tuple Items

The most common way to access values within a Python tuple is using indexing. We just need to specify the index of the elements we want to retrieve to the square bracket [] notation.

In Python, a tuple is an immutable ordered collection of elements. "Immutable" means that once a tuple is created, we cannot modify or change its contents. We can use tuples to group together related data elements, similar to lists, but with the key difference that tuples are immutable, while lists are mutable.

In addition to indexing, Python provides various other ways to access tuple items such as slicing, negative indexing, extracting a subtuple from a tuple etc. Let us go through this one-by-one –

Accessing Tuple Items with Indexing

Each element in a tuple corresponds to an index. The index starts from 0 for the first element and increments by one for each subsequent element. Index of the last item in the tuple is always "length-1", where "length" represents the total number of items in the tuple. To access the elements of a tuple we just need to specify the index of the item we need to access/retrieve, as shown below –

```
tuple[3]
```

Example

Following is the basic example to access tuple items with slicing index –

```
tuple1 = ("Rohan", "Physics", 21, 69.75)
tuple2 = (1, 2, 3, 4, 5)

print ("Item at 0th index in tuple1: ", tuple1[0])
print ("Item at index 2 in tuple2: ", tuple2[2])
```

It will produce the following output –

```
Item at 0th index in tuple1:  Rohan
Item at index 2 in tuple2:  3
```

Accessing Tuple Items with Negative Indexing

Negative indexing in Python is used to access elements from the end of a tuple, with -1 referring to the last element, -2 to the second last, and so on.

We can also access tuple items with negative indexing by using negative integers to represent positions from the end of the tuple.

Example

In the following example, we are accessing tuple items with negative indexing –

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)

print ("Item at 0th index in tup1: ", tup1[-1])
print ("Item at index 2 in tup2: ", tup2[-3])
```

We get the output as shown below –

```
Item at 0th index in tup1:  d
Item at index 2 in tup2:  True
```

Accessing Range of Tuple Items with Negative Indexing

By range of tuple items, we mean accessing a subset of elements from a tuple using slicing. Therefore, we can access a range of tuple items with negative indexing by using the slicing operation in Python.

Example

In the example below, we are accessing a range of tuple items by using negative indexing –

```
tup1 = ("a", "b", "c", "d")
tup2 = (1, 2, 3, 4, 5)

print ("Items from index 1 to last in tup1: ", tup1[1:])
print ("Items from index 2 to last in tup2", tup2[2:-1])
```

It will produce the following output –

```
Items from index 1 to last in tup1: ('b', 'c', 'd')
Items from index 2 to last in tup2: (3, 4)
```

Access Tuple Items with Slice Operator

The slice operator in Python is used to fetch one or more items from the tuple. We can access tuple items with the slice operator by specifying the range of indices we want to extract. It uses the following syntax –

```
[start:stop]
```

Where,

- **start** is the starting index (inclusive).
- **stop** is the ending index (exclusive).

Example

In the following example, we are retrieving subtuple from index 1 to last in "tuple1" and index 0 to 1 in "tuple2", and retrieving all elements in "tuple3" –

```
tuple1 = ("a", "b", "c", "d")
tuple2 = (25.50, True, -55, 1+2j)
tuple3 = (1, 2, 3, 4, 5)
tuple4 = ("Rohan", "Physics", 21, 69.75)

print ("Items from index 1 to last in tuple1: ", tuple1[1:])
print ("Items from index 0 to 1 in tuple2: ", tuple2[:2])
print ("Items from index 0 to index last in tuple3", tuple3[:])
```

Following is the output of the above code –

```
Items from index 1 to last in tuple1: ('b', 'c', 'd')
Items from index 0 to 1 in tuple2: (25.5, True)
Items from index 0 to index last in tuple3 ('Rohan', 'Physics', 21, 69.75)
```

Accessing Sub Tuple from a Tuple

A subtuple is a part of a tuple that consists of a consecutive sequence of elements from the original tuple.

We can access a subtuple from a tuple by using the slice operator with appropriate start and stop indices. It uses the following syntax –

```
my_tuple[start:stop]
```

Where,

- **start** is the starting index (inclusive).
- **stop** is the ending index (exclusive) of the subtuple.

If we do not provide any indices, the slice operator defaults to starting from index 0 and stopping at the last item in the tuple.

Example

In this example, we are fetching subtuple from index "1 to 2" in "tuple1" and index "0 to 1" in "tuple2" using slice operator –

```
tuple1 = ("a", "b", "c", "d")
tuple2 = (25.50, True, -55, 1+2j)

print ("Items from index 1 to 2 in tuple1: ", tuple1[1:3])
print ("Items from index 0 to 1 in tuple2: ", tuple2[0:2])
```

The output obtained is as follows –

Items from index 1 to 2 in tuple1: ('b', 'c')

Items from index 0 to 1 in tuple2: (25.5, True)

76. Python - Update Tuples

Updating Tuples in Python

In Python, tuple is an immutable sequence, meaning once a tuple is created, its elements cannot be changed, added, or removed.

To update a tuple in Python, you can combine various operations to create a new tuple. For instance, you can concatenate tuples, slice them, or use tuple unpacking to achieve the desired result. This often involves converting the tuple to a list, making the necessary modifications, and then converting it back to a tuple.

Updating Tuples Using Concatenation Operator

The concatenation operator in Python, denoted by `+`, is used to join two sequences, such as strings, lists, or tuples, into a single sequence. When applied to tuples, the concatenation operator joins the elements of the two (or more) tuples to create a new tuple containing all the elements from both tuples.

We can update a tuple using the concatenation operator by creating a new tuple that combines the original tuple with additional elements.

Since tuples are immutable, updating tuples using concatenation operator does not modify the original tuple but instead creates a new one with the desired elements.

Example

In the following example, we create a new tuple by concatenating "T1" with "T2" using the `+` operator –

```
# Original tuple
T1 = (10, 20, 30, 40)
# Tuple to be concatenated
T2 = ('one', 'two', 'three', 'four')
# Updating the tuple using the concatenation operator
T1 = T1 + T2
print(T1)
```

It will produce the following output –

```
(10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

Updating Tuples Using Slicing

Slicing in Python is used to extract a portion of a sequence (such as a list, tuple, or string) by specifying a range of indices. The syntax for slicing is as follows –

```
sequence[start:stop:step]
```

Where,

- **start** is the index at which the slice begins (inclusive).
- **stop** is the index at which the slice ends (exclusive).
- **step** is the interval between elements in the slice (optional).

We can update a tuple using slicing by creating a new tuple that includes slices of the original tuple combined with new elements.

Example

In this example, we are updating a tuple by slicing it into two parts and inserting new elements between the slices –

```
# Original tuple
T1 = (37, 14, 95, 40)

# Elements to be added
new_elements = ('green', 'blue', 'red', 'pink')

# Extracting slices of the original tuple
# Elements before index 2
part1 = T1[:2]

# Elements from index 2 onward
part2 = T1[2:]

# Create a new tuple
updated_tuple = part1 + new_elements + part2

# Printing the updated tuple
print("Original Tuple:", T1)
print("Updated Tuple:", updated_tuple)
```

Following is the output of the above code –

```
Original Tuple: (37, 14, 95, 40)
Updated Tuple: (37, 14, 'green', 'blue', 'red', 'pink', 95, 40)
```

Updating Tuples using List Comprehension

List comprehension in Python is a concise way to create lists. It allows you to generate new lists by applying an expression to each item in an existing iterable, such as a list, tuple, or string, optionally including a condition to filter elements.

Since tuples are immutable, updating a tuple involves converting it to a list, making the desired changes using list comprehension, and then converting it back to a tuple.

Example

In the example below, we are updating a tuple by first converting it to a list and using list comprehension to add 100 to each element. We then convert the list back to a tuple to get the updated tuple –

```
# Original tuple
T1 = (10, 20, 30, 40)

# Converting the tuple to a list
list_T1 = list(T1)

# Using list comprehension
updated_list = [item + 100 for item in list_T1]

# Converting the updated list back to a tuple
updated_tuple = tuple(updated_list)

# Printing the updated tuple
print("Original Tuple:", T1)
print("Updated Tuple:", updated_tuple)
```

Output of the above code is as follows –

```
Original Tuple: (10, 20, 30, 40)
Updated Tuple: (110, 120, 130, 140)
```

Updating Tuples using append() function

The append() function is used to add a single element to the end of a list. However, since tuples are immutable, the append() function cannot be directly used to update a tuple.

To update a tuple using the append() function, we need to first convert the tuple to a list, then use append() to add elements, and finally convert the list back to a tuple.

Example

In the following example, we first convert the original tuple "T1" to a list "list_T1". We then use a loop to iterate over the new elements and append each one to the list using the append() function. Finally, we convert the updated list back to a tuple to get the updated tuple –

```
# Original tuple
T1 = (10, 20, 30, 40)

# Convert tuple to list
list_T1 = list(T1)

# Elements to be added
new_elements = [50, 60, 70]

# Updating the list using append()
for element in new_elements:
    list_T1.append(element)

# Converting list back to tuple
updated_tuple = tuple(list_T1)

# Printing the updated tuple
```

```
print("Original Tuple:", T1)
print("Updated Tuple:", updated_tuple)
```

We get the output as shown below –

```
Original Tuple: (10, 20, 30, 40)
Updated Tuple: (10, 20, 30, 40, 50, 60, 70)
```

77. Python - Unpack Tuple Items

Unpack Tuple Items

The term "unpacking" refers to the process of parsing tuple items in individual variables. In Python, the parentheses are the default delimiters for a literal representation of sequence object.

Following statements to declare a tuple are identical.

```
>>> t1 = (x,y)
>>> t1 = x,y
>>> type (t1)
<class 'tuple'>
```

Example

To store tuple items in individual variables, use multiple variables on the left of assignment operator, as shown in the following example –

```
tup1 = (10,20,30)
x, y, z = tup1
print ("x: ", x, "y: ", "z: ",z)
```

It will produce the following output –

```
x: 10 y: 20 z: 30
```

That's how the tuple is unpacked in individual variables.

In the above example, the number of variables on the left of assignment operator is equal to the items in the tuple. What if the number is not equal to the items?

ValueError While Unpacking a Tuple

If the number of variables is more or less than the length of tuple, Python raises a ValueError.

Example

```
tup1 = (10,20,30)
x, y = tup1
x, y, p, q = tup1
```

It will produce the following output –

```
x, y = tup1
^^^^^
ValueError: too many values to unpack (expected 2)
```

```
x, y, p, q = tup1
^^^^^^^^^^^
ValueError: not enough values to unpack (expected 4, got 3)
```

Unpack Tuple Items Using Asterisk (*)

In such a case, the "*" symbol is used for unpacking. Prefix "*" to "y", as shown below –

Example 1

```
tup1 = (10,20,30)
x, *y = tup1
print ("x: ", "y: ", y)
```

It will produce the following output –

```
x: y: [20, 30]
```

The first value in tuple is assigned to "x", and rest of items to "y" which becomes a list.

Example 2

In this example, the tuple contains 6 values and variables to be unpacked are 3. We prefix "*" to the second variable.

```
tup1 = (10,20,30, 40, 50, 60)
x, *y, z = tup1
print ("x: ",x, "y: ", y, "z: ", z)
```

It will produce the following output –

```
x: 10 y: [20, 30, 40, 50] z: 60
```

Here, values are unpacked in "x" and "z" first, and then the rest of values are assigned to "y" as a list.

Example 3

What if we add "*" to the first variable?

```
tup1 = (10,20,30, 40, 50, 60)
*x, y, z = tup1
print ("x: ",x, "y: ", y, "z: ", z)
```

It will produce the following output –

```
x: [10, 20, 30, 40] y: 50 z: 60
```

Here again, the tuple is unpacked in such a way that individual variables take up the value first, leaving the remaining values to the list "x".

78. Python - Loop Tuples

Loop Through Tuple Items

Looping through tuple items in Python refers to iterating over each element in a tuple sequentially.

In Python, we can loop through the items of a tuple in various ways, with the most common being the for loop. We can also use the while loop to iterate through tuple items, although it requires additional handling of the loop control variable explicitly i.e. an index.

Loop Through Tuple Items with For Loop

A for loop in Python is used to iterate over a sequence (like a list, tuple, dictionary, string, or range) or any other iterable object. It allows you to execute a block of code repeatedly for each item in the sequence.

In a for loop, you can access each item in a sequence using a variable, allowing you to perform operations or logic based on that item's value. We can loop through tuple items using for loop by iterating over each item in the tuple.

Syntax

Following is the basic syntax to loop through items in a tuple using a for loop in Python –

```
for item in tuple:  
    # Code block to execute
```

Example

In the following example, we are using a for loop to iterate through each element in the tuple "tup" and retrieving each element followed by a space on the same line –

```
tup = (25, 12, 10, -21, 10, 100)  
for num in tup:  
    print (num, end = ' ')
```

Output

Following is the output of the above code –

```
25 12 10 -21 10 100
```

Loop Through Tuple Items with While Loop

A while loop in Python is used to repeatedly execute a block of code as long as a specified condition evaluates to "True".

We can loop through tuple items using while loop by initializing an index variable, then iterating through the tuple using the index variable and incrementing it until reaching the end of the tuple.

An index variable is used within a Loop to keep track of the current position or index in a sequence, such as a tuple or array. It is generally initialized before the Loop and updated within the Loop to iterate over the sequence.

Syntax

Following is the basic syntax for looping through items in a tuple using a while loop in Python –

```
while condition:  
    # Code block to execute
```

Example

In the below example, we iterate through each item in the tuple "my_tup" using a while loop. We use an index variable "index" to access each item sequentially, incrementing it after each iteration to move to the next item –

```
my_tup = (1, 2, 3, 4, 5)  
  
index = 0  
  
  
while index < len(my_tup):  
    print(my_tup[index])  
    index += 1
```

Output

Output of the above code is as follows –

```
1  
2  
3  
4  
5
```

Loop Through Tuple Items with Index

An index is a numeric value representing the position of an element within a sequence, such as a tuple, starting from 0 for the first element.

We can loop through tuple items using index by iterating over a range of indices corresponding to the length of the tuple and accessing each element using the index within the loop.

Example

This example initializes a tuple "tup" with integers and creates a range of indices corresponding to the length of the tuple. Then, it iterates over each index in the range and prints the value at that index in the tuple "tup" –

```
tup = (25, 12, 10, -21, 10, 100)  
  
indices = range(len(tup))
```

```
for i in indices:  
    print ("tup[{}]: {}".format(i), tup[i])
```

Output

We get the output as shown below –

```
tup[0]: 25  
tup[1]: 12  
tup[2]: 10  
tup[3]: -21  
tup[4]: 10  
tup[5]: 100
```

79. Python - Join Tuples

Joining Tuples in Python

Joining tuples in Python refers to combining the elements of multiple tuples into a single tuple. This can be achieved using various methods, such as concatenation, list comprehension, or using built-in functions like `extend()` or `sum()`.

Joining tuples does not modify the original tuples but creates a new tuple containing the combined elements.

Joining Tuples Using Concatenation ("+") Operator

The concatenation operator in Python, denoted by `+`, is used to join two sequences, such as strings, lists, or tuples, into a single sequence. When applied to tuples, the concatenation operator joins the elements of the two (or more) tuples to create a new tuple containing all the elements from both tuples.

We can join tuples using the concatenation operator by simply using the `+` symbol to concatenate them.

Example

In the following example, we are concatenating the elements of two tuples "T1" and "T2", creating a new tuple "joined_tuple" containing all the elements from both tuples –

```
# Two tuples to be joined
T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
# Joining the tuples
joined_tuple = T1 + T2

# Printing the joined tuple
print("Joined Tuple:", joined_tuple)
```

Following is the output of the above code –

```
Joined Tuple: (10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

Joining Tuples Using List Comprehension

List comprehension is a concise way to create lists in Python. It is used to generate new lists by applying an expression to each item in an existing iterable, such as a list, tuple, or range. The syntax for list comprehension is –

```
new_list = [expression for item in iterable]
```

This creates a new list where expression is evaluated for each item in the iterable.

We can join a tuple using list comprehension by iterating over multiple tuples and appending their elements to a new tuple.

Example

In this example, we are joining two tuples, T1 and T2, into a single tuple using list comprehension. The resulting tuple, joined_tuple, contains all elements from both T1 and T2 –

```
# Two tuples to be joined
T1 = (36, 24, 3)
T2 = (84, 5, 81)

# Joining the tuples using list comprehension
joined_tuple = [item for subtuple in [T1, T2] for item in subtuple]

# Printing the joined tuple
print("Joined Tuple:", joined_tuple)
```

Output of the above code is as follows –

```
Joined Tuple: [36, 24, 3, 84, 5, 81]
```

Joining Tuples Using extend() Function

The Python extend() function is used to append elements from an iterable (such as another list) to the end of the list. This function modifies the original list in place, adding the elements of the iterable to the end of the list.

The extend() function is not used for joining tuples in Python. It is used to extend a list by appending elements from another iterable (such as another list), effectively merging the two lists together.

We can join tuples using the extend() function by temporarily converting the tuples into lists, performing the joining operation as if they were lists, and then converting the resulting list back into a tuple.

Example

In the following example, we are extending the first tuple "T1" by converting it into a list "L1", then adding elements from the second tuple "T2" by first converting it into a list "L2", and finally converting the merged list back into a tuple, effectively joining the two tuples –

```
T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
L1 = list(T1)
L2 = list(T2)
L1.extend(L2)
T1 = tuple(L1)
print ("Joined Tuple:", T1)
```

The output obtained is as shown below –

```
Joined Tuple: (10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

Join Tuples using sum() Function

In Python, the `sum()` function is used to add up all the elements in an iterable, such as a list, tuple, or set. It takes an iterable as its argument and returns the sum of all the elements in that iterable.

We can join a tuple using the `sum()` function by providing the tuple as an argument to the `sum()` function. However, since the `sum()` function is specifically designed for numeric data types, this method only works for tuples containing numeric elements. It will add up all the numeric elements in the tuple and return their sum.

Syntax

Following is the syntax for using the `sum()` function to join tuples in Python –

```
result_tuple = sum((tuple1, tuple2), ())
```

Here, the first argument is a tuple containing the tuples to be joined. The second argument is the starting value for the sum. Since we are joining tuples, we use an empty tuple `()` as the starting value.

Example

In this example, the elements of the first tuple are first appended to an empty tuple. Then elements from the second tuple are appended, resulting in a new tuple that is a concatenation of the two –

```
T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
T3 = sum((T1, T2), ())
print ("Joined Tuple:", T3)
```

After executing the above code, we get the following output –

```
Joined Tuple: (10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

Joining Tuples using for Loop

A for loop in Python is used for iterating over a sequence (such as a list, tuple, string, or range) and executing a block of code for each element in the sequence. The loop continues until all elements have been processed.

We can join a tuple using a for loop by iterating over the elements of one tuple and appending each element to another tuple with the `+=` operator.

Example

In the following example, we are iterating over each element in tuple `T2`, and for each element, we are appending it to tuple `T1`, effectively joining the two tuples –

```
T1 = (10, 20, 30, 40)
T2 = ('one', 'two', 'three', 'four')
```

```
for t in T2:  
    T1+=(t,)  
print (T1)
```

We get the output as shown below –

```
(10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

80. Python - Tuple Methods

Tuple is one of the fundamental data structures in Python, and it is an immutable sequence. Unlike lists, tuples cannot be modified after creation, making them ideal for representing fixed collections of data. This immutability plays a crucial role in various scenarios where data stability and security are important. It can contain elements of different data types, such as integers, floats, strings, or even other tuples.

Python Tuple Methods

The tuple class provides few methods to analyze the data or elements. These methods allow users to retrieve information about the occurrences of specific items within a tuple and their respective indices. Since it is immutable, this class doesn't define methods for adding or removing items. It defines only two methods and these methods provide a convenient way to analyze tuple data.

Listing All the Tuple Methods

To explore all available methods for tuples, you can utilize the Python `dir()` function, which lists all properties and functions related to a class. Additionally, the `help()` function provides detailed documentation for each method. Here's an example:

```
print(dir((1, 2)))
print(help((1, 2).index))
```

The above code snippet provides a complete list of properties and functions related to the tuple class. It also demonstrates how to access detailed documentation for a specific method in your Python environment. Here is the output –

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']

Help on built-in function index:

index(value, start=0, stop=9223372036854775807, /) method of builtins.tuple instance

    Return first index of value.

    Raises ValueError if the value is not present.

(END)
```

Below are the built-in methods for tuples. Let's explore each method's basic functionality

–

Sr.No	Methods & Description
1	tuple.count(obj) Returns count of how many times obj occurs in tuple
2	tuple.index(obj) Returns the lowest index in tuple that obj appears

Finding the Index of a Tuple Item

The index() method of tuple class returns the index of first occurrence of the given item.

Syntax

```
tuple.index(obj)
```

Return value

The index() method returns an integer, representing the index of the first occurrence of "obj".

Example

Take a look at the following example –

```
tup1 = (25, 12, 10, -21, 10, 100)
print ("Tup1:", tup1)
x = tup1.index(10)
print ("First index of 10:", x)
```

It will produce the following output –

```
Tup1: (25, 12, 10, -21, 10, 100)
First index of 10: 2
```

Counting Tuple Items

The count() method in tuple class returns the number of times a given object occurs in the tuple.

Syntax

```
tuple.count(obj)
```

Return Value

Number of occurrence of the object. The count() method returns an integer.

Example

```
tup1 = (10, 20, 45, 10, 30, 10, 55)
print ("Tup1:", tup1)
c = tup1.count(10)
print ("count of 10:", c)
```

It will produce the following output –

```
Tup1: (10, 20, 45, 10, 30, 10, 55)
count of 10: 3
```

Example

Even if the items in the tuple contain expressions, they will be evaluated to obtain the count.

```
tup1 = (10, 20/80, 0.25, 10/40, 30, 10, 55)
print ("Tup1:", tup1)
c = tup1.count(0.25)
print ("count of 10:", c)
```

It will produce the following output –

```
Tup1: (10, 0.25, 0.25, 0.25, 30, 10, 55)
count of 10: 3
```

81. Python Tuple Exercises

Python Tuple Exercise 1

Python program to find unique numbers in a given tuple –

```
T1 = (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
T2 = ()
for x in T1:
    if x not in T2:
        T2+=(x,)
print ("original tuple:", T1)
print ("Unique numbers:", T2)
```

It will produce the following output –

```
original tuple: (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
Unique numbers: (1, 9, 6, 3, 4, 5, 2, 7, 8)
```

Python Tuple Exercise 2

Python program to find sum of all numbers in a tuple –

```
T1 = (1, 9, 1, 6, 3, 4)
ttl = 0
for x in T1:
    ttl+=x

print ("Sum of all numbers Using loop:", ttl)

ttl = sum(T1)
print ("Sum of all numbers sum() function:", ttl)
```

It will produce the following output –

```
Sum of all numbers Using loop: 24
Sum of all numbers sum() function: 24
```

Python Tuple Exercise 3

Python program to create a tuple of 5 random integers –

```
import random
t1 = ()
for i in range(5):
    x = random.randint(0, 100)
    t1+=(x,)
print (t1)
```

It will produce the following output –

```
(64, 21, 68, 6, 12)
```

Python Sets

82. Python - Sets

Sets in Python

In Python, a set is an unordered collection of unique elements. Unlike lists or tuples, sets do not allow duplicate values i.e. each element in a set must be unique. Sets are mutable, meaning you can add or remove items after a set has been created.

Sets are defined using curly braces {} or the built-in set() function. They are particularly useful for membership testing, removing duplicates from a sequence, and performing common mathematical set operations like union, intersection, and difference.

A set refers to a collection of distinct objects. It is used to group objects together and to study their properties and relationships. The objects in a set are called elements or members of the set.

Creating a Set in Python

Creating a set in Python refers to defining and initializing a collection of unique elements. This includes specifying the elements that will be a part of the set, ensuring that each element is unique within the set.

You can create a set in Python using curly braces {} or the set() function –

Using Curly Braces

You can directly define a set by listing its elements within curly braces, separating each element by a comma as shown below –

```
my_set = {1, 2, 3, 4, 5}  
print (my_set)
```

It will produce the following result –

```
{1, 2, 3, 4, 5}
```

Using the set() Function

Alternatively, you can create a set using the set() function by passing an iterable (like a list or a tuple) containing the elements you want to include in the set –

```
my_set = set([1, 2, 3, 4, 5])  
print (my_set)
```

We get the output as shown below –

```
{1, 2, 3, 4, 5}
```

Duplicate Elements in Set

Sets in Python are unordered collections of unique elements. If you try to create a set with duplicate elements, duplicates will be automatically removed –

```
my_set = {1, 2, 2, 3, 3, 4, 5, 5}
print (my_set)
```

The result obtained is as shown below –

```
{1, 2, 3, 4, 5}
```

Sets can contain elements of different data types, including numbers, strings, and even other sets (as long as they are immutable) –

```
mixed_set = {1, 'hello', (1, 2, 3)}
print (mixed_set)
```

The result produced is as follows –

```
{1, 'hello', (1, 2, 3)}
```

In Python, sets support various basic operations that is used to manipulate their elements. These operations include adding and removing elements, checking membership, and performing set-specific operations like union, intersection, difference, and symmetric difference.

Adding Elements in a Set

To add an element to a set, you can use the `add()` function. This is useful when you want to include new elements into an existing set. If the element is already present in the set, the set remains unchanged –

```
my_set = {1, 2, 3, 3}
# Adding an element 4 to the set
my_set.add(4)
print (my_set)
```

Following is the output obtained –

```
{1, 2, 3, 4}
```

Removing Elements from a Set

You can remove an element from a set using the `remove()` function. This is useful when you want to eliminate specific elements from the set. If the element is not present, a `KeyError` is raised –

```
my_set = {1, 2, 3, 4}
# Removes the element 3 from the set
my_set.remove(3)
print (my_set)
```

The output displayed is as shown below –

```
{1, 2, 4}
```

Alternatively, you can use the `discard()` function to remove an element from the set if it is present. Unlike `remove()`, `discard()` does not raise an error if the element is not found in the set –

```
my_set = {1, 2, 3, 4}
# No error even if 5 is not in the set
my_set.discard(5)
print (my_set)
```

We get the output as shown below –

```
{1, 2, 3, 4}
```

Membership Testing in a Set

Sets provide an efficient way to check if an element is present in the set. You can use the `in` keyword to perform this check, which returns `True` if the element is present and `False` otherwise –

```
my_set = {1, 2, 3, 4}
if 2 in my_set:
    print("2 is present in the set")
else:
    print("2 is not present in the set")
```

Following is the output of the above code –

```
2 is present in the set
```

Set Operations

In Python, sets support various set operations, which is used to manipulate and compare sets. These operations include union, intersection, difference, symmetric difference, and subset testing. Sets are particularly useful when dealing with collections of unique elements and performing operations based on set theory.

- **Union** – It combines elements from both sets using the `union()` function or the `|` operator.
- **Intersection** – It is used to get common elements using the `intersection()` function or the `&` operator.
- **Difference** – It is used to get elements that are in one set but not the other using the `difference()` function or the `-` operator.
- **Symmetric Difference** – It is used to get elements that are in either of the sets but not in both using the `symmetric_difference()` method or the `^` operator.

Python Set Comprehensions

Set comprehensions in Python is a concise way to create sets based on iterable objects, similar to list comprehensions. It is used to generate sets by applying an expression to each item in an iterable.

Set comprehensions are useful when you need to create a set from the result of applying some operation or filtering elements from another iterable.

Syntax

The syntax for set comprehensions is similar to list comprehensions, but instead of square brackets [], you use curly braces { } to denote a set –

```
set_variable = {expression for item in iterable if condition}
```

Example

In the following example, we are creating a set containing the squares of numbers from 1 to 5 using a set comprehension –

```
squared_set = {x**2 for x in range(1, 6)}
print(squared_set)
```

The output obtained is as follows –

```
{1, 4, 9, 16, 25}
```

Filtering Elements Using Set Comprehensions

You can include conditional statements in set comprehensions to filter elements based on certain criteria. For instance, to create a set of even numbers from 1 to 10, you can use a set comprehension with an if condition as shown below –

```
even_set = {x for x in range(1, 11) if x % 2 == 0}
print(even_set)
```

This will produce the following output –

```
{2, 4, 6, 8, 10}
```

Nested Set Comprehensions

Set comprehensions also support nested loops, allowing you to create sets from nested iterables. This can be useful for generating combinations or permutations of elements.

Example

```
nested_set = {(x, y) for x in range(1, 3) for y in range(1, 3)}
print(nested_set)
```

Output of the above code is as shown below –

```
{(1, 1), (1, 2), (2, 1), (2, 2)}
```

Frozen Sets

In Python, a frozen set is an immutable collection of unique elements, similar to a regular set but with the distinction that it cannot be modified after creation. Once created, the elements within a frozen set cannot be added, removed, or modified, making it a suitable choice when you need an immutable set.

You can create a frozen set in Python using the `frozenset()` function by passing an iterable (such as a list, tuple, or another set) containing the elements you want to include in the frozen set.

Example

In the following example, we are creating a frozen set of integers and then adding an element to it –

```
my_frozen_set = frozenset([1, 2, 3])
print(my_frozen_set)
my_frozen_set.add(4)
```

Following is the output of the above code –

```
frozenset({1, 2, 3})
Traceback (most recent call last):
  File "/home/cg/root/664b2732e125d/main.py", line 3, in <module>
    my_frozen_set.add(4)
AttributeError: 'frozenset' object has no attribute 'add'
```

83. Python - Access Set Items

Access Set Items

The primary way to access set items is by traversing the set using a loop, such as a for loop. By iterating over the set, you can access each element one by one and perform operations on them as needed.

In Python, sets are unordered collections of unique elements, and unlike sequences (such as lists or tuples), sets do not have a positional index for their elements. This means that you cannot access individual elements of a set directly by specifying an index.

Additionally, sets do not have keys associated with their elements, as dictionaries do. In a dictionary, each element is paired with a key, allowing you to access the value associated with a specific key. However, sets do not have this key-value pairing.

Therefore, to access the elements of a set, we need to use for loop (or List Comprehension)

Access Set Items Using For Loop

A for loop in Python is a control flow statement used for iterating over a sequence and executing a block of code for each element in the sequence. The loop continues until all elements have been processed.

We can access set items using a for loop by iterating over each element in the set sequentially. Since sets are unordered collections of unique elements, a for loop provides a convenient way to traverse the set and access each element one by one.

Example

In the following example, the for loop iterates over the set "langs", and in each iteration, the variable "lang" is assigned the value of the current element –

```
# Defining a set
langs = {"C", "C++", "Java", "Python"}
# Accessing set items using a for loop
for lang in langs:
    print (lang)
```

It will produce the following output –

```
Python
C
C++
Java
```

Access Set Items Using List Comprehension

List comprehension is an efficient way to create lists in Python. It allows you to generate a new list by applying an expression to each item in an iterable, optionally including a condition to filter elements.

We can access set items using list comprehension by converting the set into a list within the comprehension. This allows you to iterate over the set elements and perform operations on them, similar to using a for loop.

Example

In this example, we are using list comprehension to access set items by iterating over each element of "my_set" –

```
my_set = {1, 2, 3, 4, 5}
# Accessing set items using list comprehension
accessed_items = [item for item in my_set]
print(accessed_items)
```

We get the output as shown below –

```
[1, 2, 3, 4, 5]
```

Access Subset from a Set

Mathematically, a subset is a set that contains only elements that are also contained in another set. In other words, every element of the subset is also an element of the original set. If every element of set A is also an element of set B, then A is considered a subset of B, denoted as $A \subseteq B$.

In Python, you can access subsets from a set using set operations or by iterating over the power set (the set of all subsets) and filtering based on specific criteria.

- **Using Set Operations** – You can use built-in set operations such as `issubset()` function to check if one set is a subset of another.
- **Iterating Over Power Set** – Iterate over all possible subsets of the set and filter based on certain criteria to access specific subsets.

Example

Following is the basic example demonstrating how to access subsets from a set –

```
import itertools

# Defining a set
original_set = {1, 2, 3, 4}

# Checking if {1, 2} is a subset of the original set
is_subset = {1, 2}.issubset(original_set)

print("{1, 2} is a subset of the original set:", is_subset)

# Generating all subsets with two elements
subsets_with_two_elements = [set(subset) for subset in
itertools.combinations(original_set, 2)]
```

```
print("Subsets with two elements:", subsets_with_two_elements)
```

Following is the output of the above code –

```
{1, 2} is a subset of the original set: True
Subsets with two elements: [{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

Checking if Set Item Exists

You can check whether a certain item available in the set is using the [Python's membership operators](#), **in** and **not in**.

The **in** operator returns True if the specified element is found within the collection, and False otherwise. Conversely, the **not in** operator returns True if the element is not present in the collection, and False if it is.

Example

In the below example, the "**in**" operator verifies whether the item "Java" exists in the set "langs", and the "**not in**" operator checks whether the item "SQL" does not exist in the set –

```
# Defining a set
langs = {"C", "C++", "Java", "Python"}

# Checking if an item exists in the set
if "Java" in langs:
    print("Java is present in the set.")
else:
    print("Java is not present in the set.")

# Checking if an item does not exist in the set
if "SQL" not in langs:
    print("SQL is not present in the set.")
else:
    print("SQL is present in the set.")
```

It will produce the following output –

```
Java is present in the set.
SQL is not present in the set.
```

84. Python - Add Set Items

Add Set Items

Adding set items means adding new elements into an existing set. In Python, sets are mutable, which means you can modify them after they have been created. While the elements within a set are immutable (such as integers, strings, or tuples), the set itself can be modified.

You can add items to a set using various methods, such as `add()`, `update()`, or set operations like `union ()` and set comprehension.

One of the defining features of sets is their ability to hold only immutable (hashable) objects. This is because sets internally use a hash table for fast membership testing. Immutable objects are hashable, meaning they have a hash value that never changes during their lifetime.

Add Set Items using the `add()` Method

The `add()` method in Python is used to add a single element to the set. It modifies the set by inserting the specified element if it is not already present. If the element is already in the set, the `add()` method does not make any changes in the set.

Syntax

Following is the syntax to add an element to a set –

```
set.add(obj)
```

Where, `obj` is an object of any immutable type.

Example

In the following example, we are initializing an empty set called "language" and adding elements to it using the `add()` method –

```
# Defining an empty set
language = set()

# Adding elements to the set using add() method
language.add("C")
language.add("C++")
language.add("Java")
language.add("Python")

# Printing the updated set
print("Updated Set:", language)
```

It will produce the following output –

```
Updated Set: {'Python', 'Java', 'C++', 'C'}
```

Add Set Items using the update() Method

In Python, the update() method of set class is used to add multiple elements to the set. It modifies the set by adding elements from an iterable (such as another set, list, tuple, or string) to the current set. The elements in the iterable are inserted into the set if they are not already present.

Syntax

Following is the syntax to update an element to a set –

```
set.update(obj)
```

Where, obj is a set or a sequence object (list, tuple, string).

Example: Adding a Single Set Item

In the example below, we initialize a set called "my_set". Then, we use the update() method to add the element "4" to the set –

```
# Define a set
my_set = {1, 2, 3}

# Adding element to the set
my_set.update([4])

# Print the updated set
print("Updated Set:", my_set)
```

Following is the output of the above code –

```
Updated Set: {1, 2, 3, 4}
```

Example: Adding any Sequence Object as Set Items

The update() method also accepts any sequence object as argument.

In this example, we first define a set and a tuple, lang1 and lang2, containing different programming languages. Then, we add all elements from "lang2" to "lang1" using the update() method –

```
# Defining a set
lang1 = {"C", "C++", "Java", "Python"}

# Defining a tuple
lang2 = {"PHP", "C#", "Perl"}

lang1.update(lang2)

print (lang1)
```

The result obtained is as shown below –

```
{'Python', 'C++', 'C#', 'C', 'Java', 'PHP', 'Perl'}
```

Example

In this example, a set is constructed from a string, and another string is used as argument for update() method –

```
set1 = set("Hello")
set1.update("World")
print (set1)
```

It will produce the following output –

```
{'H', 'r', 'o', 'd', 'W', 'l', 'e'}
```

Add Set Items using Union Operator

In Python, the union operator (`|`) is used to perform a union operation between two sets. The union of two sets contains all the distinct elements present in either of the sets, without any duplicates.

We can add set items using the union operator by performing the union operation between two sets using the `|` operator or `union()` function, which combines the elements from both sets into a new set, containing all unique elements present in either of the original sets.

Example

The following example combine sets using the `union()` method and the `|` operator to create new sets containing unique elements from the original sets –

```
# Defining three sets
lang1 = {"C", "C++", "Java", "Python"}
lang2 = {"PHP", "C#", "Perl"}
lang3 = {"SQL", "C#"}

# Performing union operation
combined_set1 = lang1.union(lang2)
combined_set2 = lang2 | lang3

# Print the combined set
print ("Combined Set1:", combined_set1)
print("Combined Set2:", combined_set2)
```

Output of the above code is as shown below –

```
Combined Set1: {'C#', 'Perl', 'C++', 'Java', 'PHP', 'Python', 'C'}
Combined Set2: {'C#', 'Perl', 'PHP', 'SQL'}
```

Example

If a sequence object is given as argument to union() method, Python automatically converts it to a set first and then performs union –

```
lang1 = {"C", "C++", "Java", "Python"}
lang2 = ["PHP", "C#", "Perl"]
lang3 = lang1.union(lang2)
print (lang3)
```

The output produced is as follows –

```
{'PHP', 'C#', 'Python', 'C', 'Java', 'C++', 'Perl'}
```

Add Set Items Using Set Comprehension

In Python, set comprehension is a way to create sets using a single line of code. It allows you to generate a new set by applying an expression to each item in an iterable (such as a list, tuple, or range), optionally including a condition to filter elements.

We can add set items using set comprehension by iterating over an iterable, applying an expression to each element, and enclosing the comprehension expression within curly braces {} to generate a new set containing the results of the expression applied to each element.

Example

In the following example, we are defining a list of integers and then using set comprehension to generate a set containing the squares of those integers –

```
# Defining a list containing integers
numbers = [1, 2, 3, 4, 5]
# Creating a set containing squares of numbers using set comprehension
squares_set = {num ** 2 for num in numbers}
# Printing the set containing squares of numbers
print("Squares Set:", squares_set)
```

Following is the output of the above code –

```
Squares Set: {1, 4, 9, 16, 25}
```

85. Python - Remove Set Items

Remove Set Items

Removing set items implies deleting elements from a set. In Python, sets are mutable, unordered collections of unique elements, and there are several methods available to remove items from a set based on different criteria.

We can remove set items in Python using various methods such as `remove()`, `discard()`, `pop()`, `clear()`, and set comprehension. Each method provide different ways to eliminate elements from a set based on specific criteria or conditions.

Remove Set Item Using `remove()` Method

The `remove()` method in Python is used to remove the first occurrence of a specified item from a set.

We can remove set items using the `remove()` method by specifying the element we want to remove from the set. If the element is present in the set, it will be removed. However, if the element is not found, the `remove()` method will raise a `KeyError` exception.

Example

In the following example, we are deleting the element "Physics" from the set "my_set" using the `remove()` method –

```
my_set = {"Rohan", "Physics", 21, 69.75}
print ("Original set: ", my_set)

my_set.remove("Physics")
print ("Set after removing: ", my_set)
```

It will produce the following output –

```
Original set: {21, 69.75, 'Rohan', 'Physics'}
Set after removing: {21, 69.75, 'Rohan'}
```

Example

If the element to delete is not found in the set, the `remove()` method will raise a `KeyError` exception –

```
my_set = {"Rohan", "Physics", 21, 69.75}
print ("Original set: ", my_set)

my_set.remove("PHP")
print ("Set after removing: ", my_set)
```

We get the error as shown below –

```
Original set: {'Physics', 21, 69.75, 'Rohan'}
Traceback (most recent call last):
  File "/home/cg/root/664c365ac1c3c/main.py", line 4, in <module>
    my_set.remove("PHP")
KeyError: 'PHP'
```

Remove Set Item Using discard() Method

The `discard()` method in `set` class is similar to `remove()` method. The only difference is, it doesn't raise error even if the object to be removed is not already present in the set collection.

Example

In this example, we are using the `discard()` method to delete an element from a set regardless of whether it is present or not –

```
my_set = {"Rohan", "Physics", 21, 69.75}
print ("Original set: ", my_set)

# removing an existing element
my_set.discard("Physics")
print ("Set after removing Physics: ", my_set)

# removing non-existing element
my_set.discard("PHP")
print ("Set after removing non-existent element PHP: ", my_set)
```

Following is the output of the above code –

```
Original set: {21, 'Rohan', 69.75, 'Physics'}
Set after removing Physics: {21, 'Rohan', 69.75}
Set after removing non-existent element PHP: {21, 'Rohan', 69.75}
```

Remove Set Item Using pop() Method

We can also remove set items using the `pop()` method by removing and returning an arbitrary element from the set. If the set is empty, the `pop()` method will raise a `KeyError` exception.

Example

In the example below, we are defining a set with elements "1" through "5" and removing an arbitrary element from it using the `pop()` method –

```
# Defining a set
```

```
my_set = {1, 2, 3, 4, 5}

# removing and returning an arbitrary element from the set
removed_element = my_set.pop()

# Printing the removed element and the updated set
print("Removed Element:", removed_element)
print("Updated Set:", my_set)
```

We get the output as shown below –

```
Removed Element: 1
Updated Set: {2, 3, 4, 5}
```

Example

If we try to remove element from an empty set, the pop() method will raise a KeyError exception –

```
# Defining an empty set
empty_set = set()

# Removing an arbitrary element from the empty set
removed_element = empty_set.pop()
```

The error produced is as shown below –

```
Traceback (most recent call last):
  File "/home/cg/root/664c69620cd40/main.py", line 5, in <module>
    removed_element = empty_set.pop()
KeyError: 'pop from an empty set'
```

Remove Set Item Using clear() Method

The clear() method in set class removes all the items in a set object, leaving an empty set.

We can remove set items using the clear() method by removing all elements from the set, effectively making it empty.

Example

In the following example, we are defining a set with elements "1" through "5" and then using the clear() method to remove all elements from the set –

```
# Defining a set with multiple elements
my_set = {1, 2, 3, 4, 5}
```

```
# Removing all elements from the set
my_set.clear()
# Printing the updated set
print("Updated Set:", my_set)
```

It will produce the following output –

```
Updated Set: set()
```

Remove Items Existing in Both Sets

You can remove items that exist in both sets (i.e., the intersection of two sets) using the `difference_update()` method or the subtraction operator (`-=`). This removes all elements that are present in both sets from the original set.

Example

In this example, we are defining two sets "s1" and "s2", and then using the `difference_update()` method to remove elements from "s1" that are also in "s2" –

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1 before running difference_update: ", s1)
s1.difference_update(s2)
print ("s1 after running difference_update: ", s1)
```

After executing the above code, we get the following output –

```
s1 before running difference_update: {1, 2, 3, 4, 5}
s1 after running difference_update: {1, 2, 3}
set()
```

Remove Items Existing in Either of the Sets

To remove items that exist in either of two sets, you can use the symmetric difference operation. The symmetric difference between two sets results in a set containing elements that are in either of the sets but not in their intersection.

In Python, the symmetric difference operation can be performed using the `^` operator or `symmetric_difference()` method.

Example

In the following example, we are defining two sets "set1" and "set2". We are then using the symmetric difference operator (`^`) to create a new set "result_set" containing elements that are in either "set1" or "set2" but not in both –

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Removing items that exist in either set
```

```
result_set = set1 ^ set2
print("Resulting Set:", result_set)
```

Following is the output of the above code –

```
Resulting Set: {1, 2, 5, 6}
```

Remove Uncommon Set Items

You can remove uncommon items between two sets using the `intersection_update()` method. The intersection of two sets results in a set containing only the elements that are present in both sets.

To keep only the common elements in one of the original sets and remove the uncommon ones, you can update the set with its intersection.

Example

In this example, we are defining two sets "set1" and "set2". We are then using the `intersection_update()` method to modify "set1" so that it only contains elements that are also in "set2" –

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Keeping only common items in set1
set1.intersection_update(set2)
print("Set 1 after keeping only common items:", set1)
```

Output of the above code is as shown below –

```
Set 1 after keeping only common items: {3, 4}
```

The `intersection()` Method

The `intersection()` method in set class is similar to its `intersection_update()` method, except that it returns a new set object that consists of items common to existing sets.

Syntax

Following is the basic syntax of the `intersection()` method –

```
set.intersection(obj)
```

Where, `obj` is a set object.

Return value

The `intersection()` method returns a set object, retaining only those items common in itself and `obj`.

Example

In the following example, we are defining two sets "s1" and "s2", then using the `intersection()` method to create a new set "s3" containing elements that are common to both "s1" and "s2" –

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1: ", s1, "s2: ", s2)
s3 = s1.intersection(s2)
print ("s3 = s1 & s2: ", s3)
```

It will produce the following output –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}
s3 = s1 & s2: {4, 5}
```

Symmetric Difference Update of Set Items

The symmetric difference between two sets is the collection of all the uncommon items, rejecting the common elements. The `symmetric_difference_update()` method updates a set with symmetric difference between itself and the set given as argument.

Example

In the example below, we are defining two sets "s1" and "s2", then using the `symmetric_difference_update()` method to modify "s1" so that it contains elements that are in either "s1" or "s2", but not in both –

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1: ", s1, "s2: ", s2)
s1.symmetric_difference_update(s2)
print ("s1 after running symmetric difference ", s1)
```

The result obtained is as shown below –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}
s1 after running symmetric difference {1, 2, 3, 6, 7, 8}
```

Symmetric Difference of Set Items

The `symmetric_difference()` method in set class is similar to `symmetric_difference_update()` method, except that it returns a new set object that holds all the items from two sets minus the common items.

Example

In the following example, we are defining two sets "s1" and "s2". We are then using the `symmetric_difference()` method to create a new set "s3" containing elements that are in either "s1" or "s2", but not in both –

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
print ("s1: ", s1, "s2: ", s2)
```

```
s3 = s1.symmetric_difference(s2)
print ("s1 = s1^s2 ", s3)
```

It will produce the following output –

```
s1: {1, 2, 3, 4, 5} s2: {4, 5, 6, 7, 8}
s1 = s1^s2 {1, 2, 3, 6, 7, 8}
```

86. Python - Loop Sets

Loop Through Set Items

Looping through set items in Python refers to iterating over each element in a set. We can later perform required operations on each item. These operation includes list printing elements, conditional operations, filtering elements etc.

Unlike lists and tuples, sets are unordered collections, so the elements will be accessed in an arbitrary order. You can use a for loop to iterate through the items in a set.

Loop Through Set Items with For Loop

A for loop in Python is used to iterate over a sequence (like a list, tuple, dictionary, string, or range) or any other iterable object. It allows you to execute a block of code repeatedly for each item in the sequence.

In a for loop, you can access each item in a sequence using a variable, allowing you to perform operations or logic based on that item's value. We can loop through set items using for loop by iterating over each item in the set.

Syntax

Following is the basic syntax to loop through items in a set using a for loop in Python –

```
for item in set:  
    # Code block to execute
```

Example

In the following example, we are using a for loop to iterate through each element in the set "my_set" and retrieving each element –

```
# Defining a set with multiple elements  
my_set = {25, 12, 10, -21, 10, 100}  
  
# Loop through each item in the set  
for item in my_set:  
    # Performing operations on each element  
    print("Item:", item)
```

Output

Following is the output of the above code –

```
Item: 100  
Item: 25  
Item: 10  
Item: -21
```

```
Item: 12
```

Loop Through Set Items with While Loop

A while loop in Python is used to repeatedly execute a block of code as long as a specified condition evaluates to "True".

We can loop through set items using a while loop by converting the set to an iterator and then iterating over each element until the iterator reaches end of the set.

An iterator is an object that allows you to traverse through all the elements of a collection (such as a list, tuple, set, or dictionary) one element at a time.

Example

In the below example, we are iterating through a set using an iterator and a while loop. The "try" block retrieves and prints each item, while the "except StopIteration" block breaks the loop when there are no more items to fetch –

```
# Defining a set with multiple elements
my_set = {1, 2, 3, 4, 5}

# Converting the set to an iterator
set_iterator = iter(my_set)

# Looping through each item in the set using a while loop
while True:
    try:
        # Getting the next item from the iterator
        item = next(set_iterator)
        # Performing operations on each element
        print("Item:", item)
    except StopIteration:
        # If StopIteration is raised, break from the loop
        break
```

Output

Output of the above code is as follows –

```
Item: 1
Item: 2
Item: 3
Item: 4
Item: 5
```

Iterate using Set Comprehension

A set comprehension in Python is a concise way to create sets by iterating over an iterable and optionally applying a condition. It is used to generate sets using a syntax similar to list comprehensions but results in a set, ensuring all elements are unique and unordered.

We can iterate using set comprehension by defining a set comprehension expression within curly braces {} and specifying the iteration and condition logic within the expression. Following is the syntax –

```
result_set = {expression for item in iterable if condition}
```

Where,

- **expression** – It is an expression to evaluate for each item in the iterable.
- **item** – It is a variable representing each element in the iterable.
- **iterable** – It is a collection to iterate over (e.g., list, tuple, set).
- **condition** – It is optional condition to filter elements included in the resulting set.

Example

In this example, we are using a set comprehension to generate a set containing squares of even numbers from the original list "numbers" –

```
# Original list
numbers = [1, 2, 3, 4, 5]

# Set comprehension to create a set of squares of even numbers
squares_of_evens = {x**2 for x in numbers if x % 2 == 0}

# Print the resulting set
print(squares_of_evens)
```

Output

We get the output as shown below –

```
{16, 4}
```

Iterate through a Set Using the enumerate() Function

The enumerate() function in Python is used to iterate over an iterable object while also providing the index of each element.

We can iterate through a set using the enumerate() function by converting the set into a list and then applying enumerate() to iterate over the elements along with their index positions. Following is the syntax –

```
for index, item in enumerate(list(my_set)):
    # Your code here
```

Example

In the following example, we are first converting a set into a list. Then, we iterate through the list using a for loop with enumerate() function, retrieving each item along with its index –

```
# Converting the set into a list
my_set = {1, 2, 3, 4, 5}
set_list = list(my_set)

# Iterating through the list
for index, item in enumerate(set_list):
    print("Index:", index, "Item:", item)
```

Output

The output produced is as shown below –

```
Index: 0 Item: 1
Index: 1 Item: 2
Index: 2 Item: 3
Index: 3 Item: 4
Index: 4 Item: 5
```

Loop Through Set Items with add() Method

The `add()` method in Python is used to add a single element to a set. If the element is already present in the set, the set remains unchanged.

We cannot directly loop through set items using the `add()` method because `add()` is used specifically to add individual elements to a set, not to iterate over existing elements.

To loop through set items, we use methods like a for loop or set comprehension.

Example

In this example, we loop through a sequence of numbers and add each number to the set using the `add()` method. The loop iterates over existing elements, while `add()` method adds new elements to the set –

```
# Creating an empty set
my_set = set()

# Looping through a sequence and adding elements to the set
for i in range(5):
    my_set.add(i)
print(my_set)
```

It will produce the following output –

```
{0, 1, 2, 3, 4}
```

87. Python - Join Sets

In Python, a set is an ordered collection of items. The items may be of different types. However, an item in the set must be an immutable object. It means, we can only include numbers, string and tuples in a set and not lists. Python's set class has different provisions to join set objects.

Join Sets in Python

Joining sets in Python refers to merging two or more sets as a single set. When you join sets, you merge the elements of multiple sets while ensuring that duplicate elements are removed, as sets do not allow duplicate elements.

This can be achieved using various methods, such as union, update, set comprehension, set concatenation, copying, and iterative addition.

Join Python Sets Using "|" Operator

The "|" symbol (pipe) is defined as the union operator. It performs the $A \cup B$ operation and returns a set of items in A, B or both. Set doesn't allow duplicate items.

Example

In the following example, we are performing a union operation on sets "s1" and "s2" using the "|" operator, creating a new set "s3" containing elements from both sets without duplicates –

```
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
s3 = s1|s2  
print (s3)
```

It will produce the following output –

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Join Python Sets Using union() Method

The set class has union() method that performs the same operation as | operator. It returns a set object that holds all items in both sets, discarding duplicates.

Example

In this example, we are invoking the union() method on set "s1", passing set "s2" as an argument, which returns a new set "s3" containing elements from both sets without duplicates –

```
s1={1,2,3,4,5}  
s2={4,5,6,7,8}  
s3 = s1.union(s2)
```

```
print (s3)
```

Following is the output obtained –

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Join Python Sets Using update() Method

The update() method also joins the two sets, as the union() method. However, it doesn't return a new set object. Instead, the elements of second set are added in first, duplicates not allowed.

Example

In the example below, we are updating set "s1" with the elements of set "s2" using the update() method, modifying "s1" to contain elements from both sets without duplicates –

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
s1.update(s2)
print (s1)
```

The result obtained is as shown below –

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Join Python Sets Using Unpacking Operator

In Python, the "*" symbol is used as unpacking operator. The unpacking operator internally assigns each element in a collection to a separate variable.

We can join Python sets using the unpacking operator (*) by unpacking the elements of multiple sets into a new set.

Example

In the following example, we are creating a new set "s3" by unpacking the elements of sets "s1" and "s2" using the * operator within a set literal –

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
s3 = {*s1, *s2}
print (s3)
```

The output produced is as follows –

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Join Python Sets Using Set Comprehension

Set comprehension in Python is a concise way to create sets using an iterable, similar to list comprehension but resulting in a set instead of a list. It allows you to generate sets by applying an expression to each item in an iterable while optionally filtering the items based on a condition.

We can join python sets using set comprehension by iterating over multiple sets and adding their elements to a new set.

Example

In this example, we are creating a new set "joined_set" using a set comprehension. By iterating over a list containing "set1" and "set2", and then iterating over each element "x" within each set "s", we merge all elements from both sets into "joined_set" –

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

joined_set = {x for s in [set1, set2] for x in s}
print(joined_set)
```

Output of the above code is as shown below –

```
{1, 2, 3, 4, 5}
```

Join Python Sets Using Iterative Addition

Iterative addition in the context of sets refers to iteratively adding elements from one set to another set using a loop or iteration construct. This allows you to merge the elements of multiple sets into a single set, ensuring that duplicate elements are not included.

We can join python sets using iterative addition by iterating over the elements of each set and adding them to a new set.

Example

In the example below, we first initialize an empty set. Then, we iterate over each element in "set1" and "set2" separately, adding each element into a new set named "joined_set" using the add() method –

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Initializing an empty set to hold the merged elements
joined_set = set()

# Iterating over set1 and adding its elements to the joined set
for element in set1:
    joined_set.add(element)

# Iterating over set2 and adding its elements to the joined set
for element in set2:
    joined_set.add(element)

print(joined_set)
```

After executing the above code, we get the following output –

```
{1, 2, 3, 4, 5}
```

88. Python - Copy Sets

Python Copy Sets

Copying sets in Python refers to creating a new set that contains the same elements as an existing set. Unlike simple variable assignment, which creates a reference to the original set, copying ensures that changes made to the copied set do not affect the original set, and vice versa.

There are different methods for copying a set in Python, including using the `copy()` method, the `set()` function or set comprehension.

Copy Sets Using the `copy()` Method

The `copy()` method in `set` class is used to create a shallow copy of a `set` object.

A shallow copy means that the method creates a new collection object, but does not create copies of the objects contained within the original collection. Instead, it copies the references to these objects.

Therefore, if the original collection contains mutable objects (like lists, dictionaries, or other sets), modifications to these objects will be reflected in both the original and the copied collections.

Syntax

Following is the syntax of the `copy()` method –

```
set.copy()
```

Return Value

The `copy()` method returns a new set which is a shallow copy of existing set.

Example

In the following example, we are creating a copy of the set "lang1" and storing it in "lang2", then retrieving both sets and their memory addresses using `id()`.

After adding an element to "lang1", we retrieve both sets and their memory addresses again to show that "lang1" and "lang2" are independent copies –

```
lang1 = {"C", "C++", "Java", "Python"}  
print ("lang1: ", lang1, "id(lang1): ", id(lang1))  
lang2 = lang1.copy()  
print ("lang2: ", lang2, "id(lang2): ", id(lang2))  
lang1.add("PHP")  
print ("After updating lang1")  
print ("lang1: ", lang1, "id(lang1): ", id(lang1))  
print ("lang2: ", lang2, "id(lang2): ", id(lang2))
```

Output

This will produce the following output –

```
lang1: {'Python', 'Java', 'C', 'C++'} id(lang1): 2451578196864
lang2: {'Python', 'Java', 'C', 'C++'} id(lang2): 2451578197312
```

After updating lang1

```
lang1: {'Python', 'C', 'C++', 'PHP', 'Java'} id(lang1): 2451578196864
lang2: {'Python', 'Java', 'C', 'C++'} id(lang2): 2451578197312
```

Copy Sets using the set() Function

The Python `set()` function is used to create a new set object. It takes an iterable as an argument and convert it into a set, removing any duplicate elements in the process. If no argument is provided, it creates an empty set.

We can copy set using the `set()` function by passing the original set as an argument to the `set()` constructor. This creates a new set that contains all the elements of the original set, ensuring that any modifications to the new set do not affect the original set.

Example

In this example, we are creating a copy of "original_set" using the `set()` function and storing it in "copied_set" –

```
# Original set
original_set = {1, 2, 3, 4}

# Copying the set using the set() function
copied_set = set(original_set)

print("copied set:", copied_set)

# Demonstrating that the sets are independent
copied_set.add(5)
print("copied set:", copied_set)
print("original set:", original_set)
```

Output

Following is the output of the above code –

```
copied set: {1, 2, 3, 4}
copied set: {1, 2, 3, 4, 5}
original set: {1, 2, 3, 4}
```

Copy Sets Using Set Comprehension

Set comprehension is a concise way to create sets in Python. It is used to generate a new set by iterating over an iterable and optionally applying conditions to filter elements. The

syntax is similar to list comprehension but with curly braces {} instead of square brackets [] –

```
{expression for item in iterable if condition}
```

We can copy sets using set comprehension by iterating over the elements of the original set and directly creating a new set with those elements.

Example

In the example below, we create an original set named "original_set", then copy it using set comprehension into "copied_set" –

```
# Original set  
original_set = {1, 2, 3, 4, 5}  
  
# Copying the set using set comprehension  
copied_set = {x for x in original_set}  
  
print("Copied set:", copied_set)
```

Output

Output of the above code is as shown below –

```
Copied set: {1, 2, 3, 4, 5}
```

89. Python - Set Operators

Set Operators in Python

The set operators in Python are special symbols and functions that allow you to perform various operations on sets, such as union, intersection, difference, and symmetric difference. These operators provide a way to combine, compare, and modify sets.

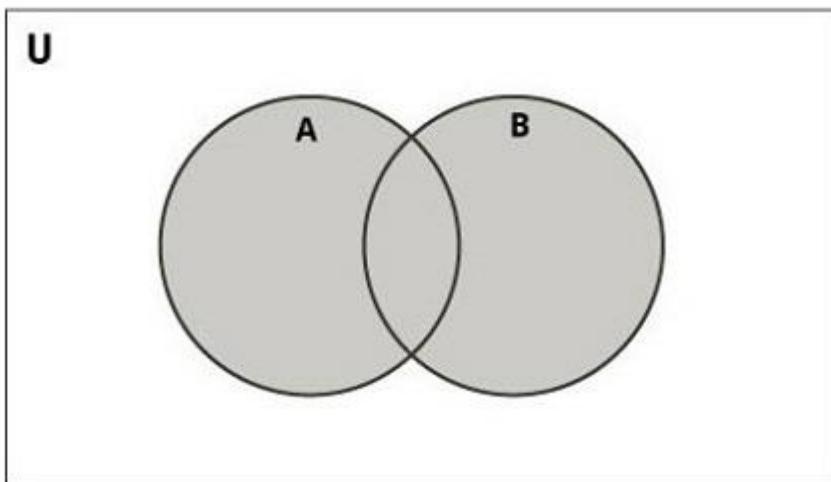
Python implements them with following set operators –

Python Set Union Operator (|)

The union of two sets is a set containing all distinct elements that are in A or in B or both. For example,

```
{1,2}U{2,3}={1,2,3}
```

The following diagram illustrates the union of two sets.



In Python, you can perform the union operation using the `union()` function or the `|` operator. This operation combines the elements of two sets while eliminating duplicates, resulting in a new set containing all unique elements from both sets –

Example

The following example uses the "`|`" operator and `union()` function, and returns the union of two sets –

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = {6, 8, 9}
set4 = {9, 45, 73}
union_set1 = set1.union(set2)
union_set2 = set3 | set4
```

```
print ('The union of set1 and set2 is', union_set1)
print ('The union of set3 and set4 is', union_set2)
```

After executing the above code, we get the following output –

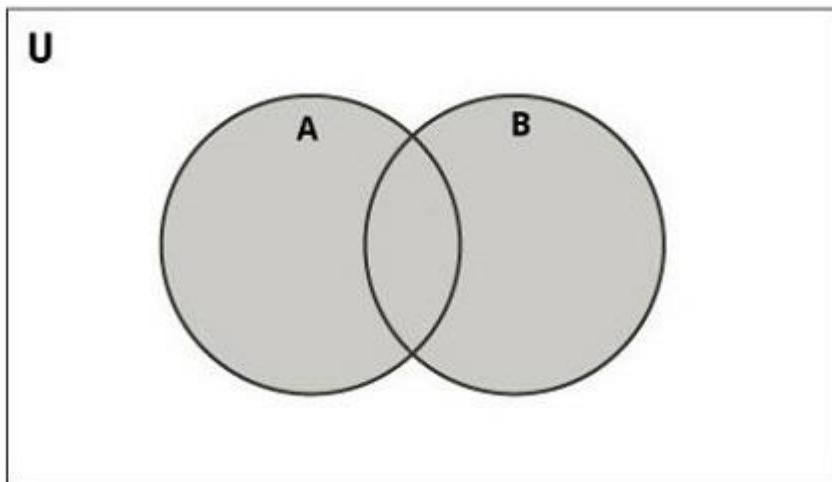
```
The union of set1 and set2 is {1, 2, 3, 4, 5}
The union of set3 and set4 is {73, 6, 8, 9, 45}
```

Python Set Intersection Operator (&)

The intersection of two sets AA and BB, denoted by $A \cap B$, consists of all elements that are common to both in A and B. For example,

```
{1,2} ∩ {2,3} = {2}
```

The following diagram illustrates intersection of two sets.



Python provides the `intersection()` function or the `&` operator to perform this operation. The resulting set contains only the elements present in both sets –

Example

Following example uses `&` operator and `intersection()` function, and returns intersection of two sets –

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = {6, 8, 9}
set4 = {9, 8, 73}

intersection_set1 = set1.intersection(set2)
intersection_set2 = set3 & set4

print ('The intersection of set1 and set2 is', intersection_set1)
print ('The intersection of set3 and set4 is', intersection_set2)
```

It will produce the following output –

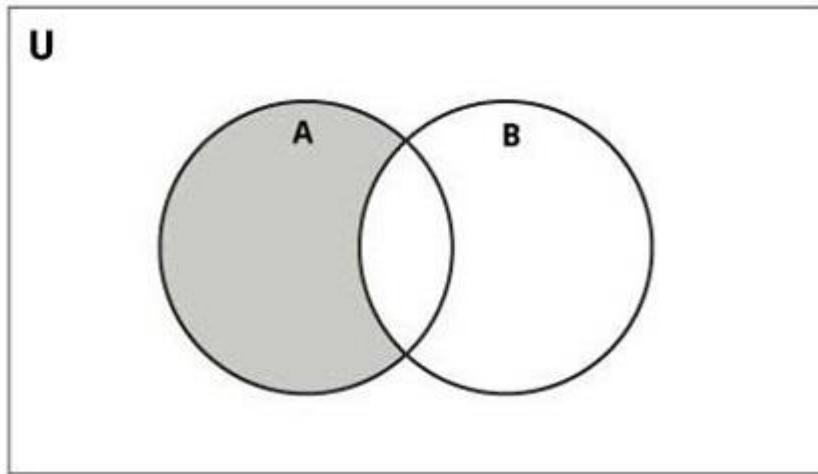
```
The intersection of set1 and set2 is {3}
The intersection of set3 and set4 is {8, 9}
```

Python Set Difference Operator (-)

The difference (subtraction) between two sets consists of elements present in the first set but not in the second set. It is defined as follows. The set $A-B$ consists of elements that are in A but not in B. For example,

```
If A={1,2,3} and B={3,5}, then A-B={1,2}
```

The following diagram illustrates difference of two sets –



Python provides the `difference()` function or the `-` operator to perform this operation. The resulting set contains elements unique to the first set –

Example

The following example uses the `"-"` operator and the `difference()` function, and returns difference of two sets –

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = {6, 8, 9}
set4 = {9, 8, 73}
difference_set1 = set1.difference(set2)
difference_set2 = set3 - set4
print ('The difference between set1 and set2 is', difference_set1)
print ('The difference between set3 and set4 is', difference_set2)
```

We get the output as shown below –

```
The difference between set1 and set2 is {1, 2}
The difference between set3 and set4 is {6}
```

Note that `"s1-s2"` is not the same as `"s2-s1"`.

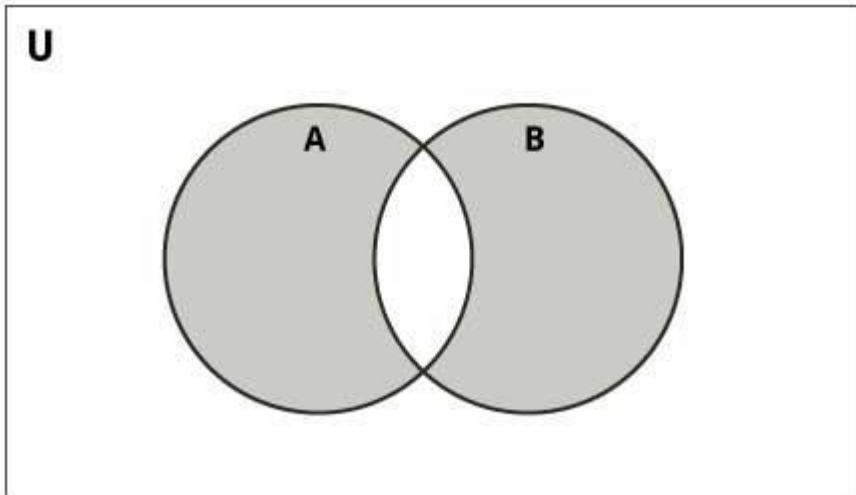
Python Set Symmetric Difference Operator

The symmetric difference of two sets consists of elements that are present in either set but not in both sets. The symmetric difference of A and B is denoted by " $A \Delta B$ " and is defined by –

$$A \Delta B = (A - B) \cup (B - A)$$

If $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $B = \{1, 3, 5, 6, 7, 8, 9\}$, then $A \Delta B = \{2, 4, 9\}$.

The following diagram illustrates the symmetric difference between two sets –



Python provides the `symmetric_difference()` function or the `^` operator to perform this operation. The resulting set contains elements that are unique to each set.

Example

The following example uses the "`^`" operator and the `symmetric_difference()` function, and returns symbolic difference of two sets –

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = {6, 8, 9}
set4 = {9, 8, 73}
symmetric_difference_set1 = set1.symmetric_difference(set2)
symmetric_difference_set2 = set3 ^ set4
print ('The symmetric difference of set1 and set2 is',
      symmetric_difference_set1)
print ('The symmetric difference of set3 and set4 is',
      symmetric_difference_set2)
```

The result produced is as follows –

```
The symmetric difference of set1 and set2 is {1, 2, 4, 5}
The symmetric difference of set3 and set4 is {73, 6}
```

Python Subset Testing Operation

You can check whether one set is a subset of another using the `issubset()` function or the `<=` operator. A set A is considered a subset of another set B if all elements of A are also present in B –

Example

The following example uses the "`<=`" operator and the `issubset()` function, and returns subset testing of two sets –

```
set1 = {1, 2}
set2 = {1, 2, 3, 4}
set3 = {64, 47, 245, 48}
set4 = {64, 47, 3}
is_subset1 = set1.issubset(set2)
is_subset2 = set3 <= set4
print ('set1 is a subset of set2:', is_subset1)
print ('set3 is a subset of set4:', is_subset2)
```

The result produced is as follows –

```
set1 is a subset of set2: True
set3 is a subset of set4: False
```

90. Python - Set Methods

Sets in Python are unordered collections of unique elements, often used for membership testing and eliminating duplicates. Set objects support various mathematical operations like union, intersection, difference, and symmetric difference.

The set class includes several built-in methods that allow you to add, update, and delete elements efficiently, as well as to perform various set operations such as union, intersection, difference, and symmetric difference on elements.

Understanding Set Methods

The set methods provide convenient ways to manipulate sets, allowing users to add or remove elements, perform set operations, and check for membership and relationships between sets.

You can view all available methods for sets, using the Python `dir()` function to list all properties and functions related to the set class. Additionally, the `help()` function provides detailed documentation for each method.

Python Set Methods

Below are the built-in methods for sets in Python, categorized based on their functionality. Let's explore and understand the basic functionality of each method.

Adding and Removing Elements

The following are the methods specifically designed for adding and removing item/items into a set –

Sr.No.	Methods with Description
1	<u>set.add()</u> Add an element to a set.
2	<u>set.clear()</u> Remove all elements from a set.
3	<u>set.copy()</u> Return a shallow copy of a set.
4	<u>set.discard()</u> Remove an element from a set if it is a member.
5	<u>set.pop()</u> Remove and return an arbitrary set element.
6	<u>set.remove()</u> Remove an element from a set; it must be a member.

Set Operations

These methods perform set operations such as union, intersection, difference, and symmetric difference –

Sr.No.	Methods with Description
1	<u>set.update()</u> Update a set with the union of itself and others.
2	<u>set.difference_update()</u> Remove all elements of another set from this set.
3	<u>set.intersection()</u> Returns the intersection of two sets as a new set.
4	<u>set.intersection_update()</u> Updates a set with the intersection of itself and another.
5	<u>set.isdisjoint()</u> Returns True if two sets have a null intersection.
6	<u>set.issubset()</u> Returns True if another set contains this set.
7	<u>set.issuperset()</u> Returns True if this set contains another set.
8	<u>set.symmetric_difference()</u> Returns the symmetric difference of two sets as a new set.
9	<u>set.symmetric_difference_update()</u> Update a set with the symmetric difference of itself and another.
10	<u>set.union()</u> Returns the union of sets as a new set.
11	<u>set.difference()</u> Returns the difference of two or more sets as a new set.

91. Python - Set Exercises

Python Set Exercise 1

Python program to find common elements in two lists with the help of set operations –

```
l1=[1,2,3,4,5]
l2=[4,5,6,7,8]
s1=set(l1)
s2=set(l2)
commons = s1&s2 # or s1.intersection(s2)
commonlist = list(commons)
print (commonlist)
```

It will produce the following output –

```
[4, 5]
```

Python Set Exercise 2

Python program to check if a set is a subset of another –

```
s1={1,2,3,4,5}
s2={4,5}
if s2.issubset(s1):
    print ("s2 is a subset of s1")
else:
    print ("s2 is not a subset of s1")
```

It will produce the following output –

```
s2 is a subset of s1
```

Python Set Exercise 3

Python program to obtain a list of unique elements in a list –

```
T1 = (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
s1 = set(T1)
print (s1)
```

It will produce the following output –

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Python Dictionaries

92. Python - Dictionaries

Dictionaries in Python

In Python, a dictionary is a built-in data type that stores data in key-value pairs. It is an unordered, mutable, and indexed collection. Each key in a dictionary is unique and maps to a value. Dictionaries are often used to store data that is related, such as information associated with a specific entity or object, where you can quickly retrieve a value based on its key.

Python's dictionary is an example of a mapping type. A mapping object 'maps' the value of one object to another. To establish mapping between a key and a value, the colon (:) symbol is put between the two.

Each key-value pair is separated by a comma and enclosed within curly braces {}. The key and value within each pair are separated by a colon (:), forming the structure key:value.

Given below are some examples of Python dictionary objects –

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar",
"Telangana": "Hyderabad", "Karnataka": "Bengaluru"}  
numbers = {10: "Ten", 20: "Twenty", 30: "Thirty", 40: "Forty"}  
marks = {"Savita": 67, "Imtiaz": 88, "Laxman": 91, "David": 49}
```

Key Features of Dictionaries

Following are the key features of dictionaries –

- **Unordered** – The elements in a dictionary do not have a specific order. Python dictionaries before version 3.7 did not maintain insertion order. Starting from Python 3.7, dictionaries maintain insertion order as a language feature.
- **Mutable** – You can change, add, or remove items after the dictionary has been created.
- **Indexed** – Although dictionaries do not have numeric indexes, they use keys as indexes to access the associated values.
- **Unique Keys** – Each key in a dictionary must be unique. If you try to assign a value to an existing key, the old value will be replaced by the new value.
- **Heterogeneous** – Keys and values in a dictionary can be of any data type.

Example 1

Only a number, string or tuple can be used as key. All of them are immutable. You can use an object of any type as the value. Hence following definitions of dictionary are also valid –

```
d1 = {"Fruit": ["Mango", "Banana"], "Flower": ["Rose", "Lotus"]}  
d2 = {('India, USA'): 'Countries', ('New Delhi', 'New York'): 'Capitals'}  
print (d1)  
print (d2)
```

It will produce the following output –

```
{'Fruit': ['Mango', 'Banana'], 'Flower': ['Rose', 'Lotus']}
{'India, USA': 'Countries', ('New Delhi', 'New York'): 'Capitals'}
```

Example 2

Python doesn't accept mutable objects such as list as key, and raises `TypeError`.

```
d1 = {[{"Mango", "Banana"}]: "Fruit", "Flower": ["Rose", "Lotus"]}
print (d1)
```

It will raise a `TypeError` –

```
Traceback (most recent call last):
  File "C:\Users\Sairam\PycharmProjects\pythonProject\main.py", line 8, in <module>
    d1 = {[{"Mango", "Banana"}]: "Fruit", "Flower": ["Rose", "Lotus"]}
                                         ^
TypeError: unhashable type: 'list'
```

Example 3

You can assign a value to more than one keys in a dictionary, but a key cannot appear more than once in a dictionary.

```
d1 = {"Banana": "Fruit", "Rose": "Flower", "Lotus": "Flower", "Mango": "Fruit"}
d2 = {"Fruit": "Banana", "Flower": "Rose", "Fruit": "Mango", "Flower": "Lotus"}
print (d1)
print (d2)
```

It will produce the following output –

```
{'Banana': 'Fruit', 'Rose': 'Flower', 'Lotus': 'Flower', 'Mango': 'Fruit'}
{'Fruit': 'Mango', 'Flower': 'Lotus'}
```

Creating a Dictionary

You can create a dictionary in Python by placing a comma-separated sequence of key-value pairs within curly braces {}, with a colon : separating each key and its associated value. Alternatively, you can use the `dict()` function.

Example

The following example demonstrates how to create a dictionary called "student_info" using both curly braces and the `dict()` function –

```
# Creating a dictionary using curly braces
sports_player = {
    "Name": "Sachin Tendulkar",
    "Age": 48,
```

```

    "Sport": "Cricket"
}

print ("Dictionary using curly braces:", sports_player)
# Creating a dictionary using the dict() function
student_info = dict(name="Alice", age=21, major="Computer Science")
print("Dictionary using dict():",student_info)

```

The result produced is as shown below –

```

Dictionary using curly braces: {'Name': 'Sachin Tendulkar', 'Age': 48, 'Sport': 'Cricket'}
Dictionary using dict(): {'name': 'Alice', 'age': 21, 'major': 'Computer Science'}

```

Accessing Dictionary Items

You can access the value associated with a specific key using square brackets [] or the get() method –

```

student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}

# Accessing values using square brackets
name = student_info["name"]
print("Name:",name)

# Accessing values using the get() method
age = student_info.get("age")
print("Age:",age)

```

The result obtained is as follows –

```

Name: Alice
Age: 21

```

Modifying Dictionary Items

You can modify the value associated with a specific key or add a new key-value pair –

```

student_info = {
    "name": "Alice",
    "age": 21,
}

```

```

    "major": "Computer Science"
}

# Modifying an existing key-value pair
student_info["age"] = 22

# Adding a new key-value pair
student_info["graduation_year"] = 2023
print("The modified dictionary is:", student_info)

```

Output of the above code is as follows –

```
The modified dictionary is: {'name': 'Alice', 'age': 22, 'major': 'Computer Science', 'graduation_year': 2023}
```

Removing Dictionary Items

You can remove items using the `del` statement, the `pop()` method, or the `popitem()` method –

```

student_info = {
    "name": "Alice",
    "age": 22,
    "major": "Computer Science",
    "graduation_year": 2023
}

# Removing an item using the del statement
del student_info["major"]

# Removing an item using the pop() method
graduation_year = student_info.pop("graduation_year")

print(student_info)

```

Following is the output of the above code –

```
{'name': 'Alice', 'age': 22}
```

Iterating Through a Dictionary

You can iterate through the keys, values, or key-value pairs in a dictionary using loops –

```

student_info = {
    "name": "Alice",
    "age": 22,
}

```

```

    "major": "Computer Science",
    "graduation_year": 2023
}

# Iterating through keys
for key in student_info:
    print("Keys:",key, student_info[key])

# Iterating through values
for value in student_info.values():
    print("Values:",value)

# Iterating through key-value pairs
for key, value in student_info.items():
    print("Key:Value:",key, value)

```

After executing the above code, we get the following output –

```

Keys: name Alice
Keys: age 22
Keys: major Computer Science
Keys: graduation_year 2023
Values: Alice
Values: 22
Values: Computer Science
Values: 2023
Key:Value: name Alice
Key:Value: age 22
Key:Value: major Computer Science
Key:Value: graduation_year 2023

```

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys –

- More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
```

```
print ("dict['Name']: ", dict['Name'])
```

When the above code is executed, it produces the following result –

```
dict['Name']: Manni
```

- Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example –

```
dict = {[ 'Name']: 'Zara', 'Age': 7}
print ("dict['Name']: ", dict['Name'])
```

When the above code is executed, it produces the following result –

```
Traceback (most recent call last):
File "test.py", line 3, in <module>
dict = {[ 'Name']: 'Zara', 'Age': 7};
TypeError: unhashable type: 'list'
```

Python Dictionary Operators

In Python, following operators are defined to be used with dictionary operands. In the example, the following dictionary objects are used.

```
d1 = {'a': 2, 'b': 4, 'c': 30}
d2 = {'a1': 20, 'b1': 40, 'c1': 60}
```

Operator	Description	Example
dict[key]	Extract/assign the value mapped with key	print (d1['b']) retrieves 4
		d1['b'] = 'Z' assigns new value to key 'b'
dict1 dict2	Union of two dictionary objects, returning new object	d3=d1 d2 ; print (d3)
		{'a': 2, 'b': 4, 'c': 30, 'a1': 20, 'b1': 40, 'c1': 60}
dict1 =dict2	Augmented dictionary union operator	d1 =d2; print (d1)
		{'a': 2, 'b': 4, 'c': 30, 'a1': 20, 'b1': 40, 'c1': 60}

Python Dictionary Methods

Python includes following dictionary methods –

Sr.No.	Methods with Description
1	dict.clear()
	Removes all elements of dictionary <i>dict</i>
2	dict.copy()

	Returns a shallow copy of dictionary <i>dict</i>
3	<code>dict.fromkeys()</code> Create a new dictionary with keys from seq and values <i>set</i> to <i>value</i> .
4	<code>dict.get(key, default=None)</code> For key <i>key</i> , returns value or default if key not in dictionary
5	<code>dict.has_key(key)</code> Returns <i>true</i> if key in dictionary <i>dict</i> , <i>false</i> otherwise
6	<code>dict.items()</code> Returns a list of <i>dict</i> 's (key, value) tuple pairs
7	<code>dict.keys()</code> Returns list of dictionary <i>dict</i> 's keys
8	<code>dict.setdefault(key, default=None)</code> Similar to <code>get()</code> , but will set <code>dict[key]=default</code> if <i>key</i> is not already in <i>dict</i>
9	<code>dict.update(dict2)</code> Adds dictionary <i>dict2</i> 's key-values pairs to <i>dict</i>
10	<code>dict.values()</code> Returns list of dictionary <i>dict</i> 's values

Built-in Functions with Dictionaries

Following are the built-in functions we can use with Dictionaries –

Sr.No.	Function with Description
1	<code>cmp(dict1, dict2)</code> Compares elements of both dict.
2	<code>len(dict)</code> Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	<code>str(dict)</code> Produces a printable string representation of a dictionary
4	<code>type(variable)</code>

Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

93. Python - Access Dictionary Items

Access Dictionary Items

Accessing dictionary items in Python involves retrieving the values associated with specific keys within a dictionary data structure. Dictionaries are composed of key-value pairs, where each key is unique and maps to a corresponding value. Accessing dictionary items allows you to retrieve these values by providing the respective keys.

There are various ways to access dictionary items in Python. They include –

- Using square brackets []
- The get() method
- Iterating through the dictionary using loops
- Or specific methods like keys(), values(), and items()

We will discuss each method in detail to understand how to access and retrieve data from dictionaries.

Access Dictionary Items Using Square Brackets []

In Python, the square brackets [] are used for creating lists, accessing elements from a list or other iterable objects (like strings, tuples, or dictionaries), and for list comprehension.

We can access dictionary items using square brackets by providing the key inside the brackets. This retrieves the value associated with the specified key.

Example 1

In the following example, we are defining a dictionary named "capitals" where each key represents a state and its corresponding value represents the capital city.

Then, we access and retrieve the capital cities of Gujarat and Karnataka using their respective keys 'Gujarat' and 'Karnataka' from the dictionary –

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar",
"Telangana":"Hyderabad", "Karnataka":"Bengaluru"}

print ("Capital of Gujarat is : ", capitals['Gujarat'])

print ("Capital of Karnataka is : ", capitals['Karnataka'])
```

It will produce the following output –

```
Capital of Gujarat is: Gandhinagar
Capital of Karnataka is: Bengaluru
```

Example 2

Python raises a KeyError if the key given inside the square brackets is not present in the dictionary object –

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar",
"Telangana":"Hyderabad", "Karnataka":"Bengaluru"}
```

```
print ("Capital of Haryana is : ", capitals['Haryana'])
```

Following is the error obtained –

```
print ("Capital of Haryana is : ", capitals['Haryana'])
```

~~~~~ ^~~~~~ ^~~~~~ ^~~~~~ ^~~~~~

KeyError: 'Haryana'

## Access Dictionary Items Using get() Method

The `get()` method in Python's `dict` class is used to retrieve the value associated with a specified key. If the key is not found in the dictionary, it returns a default value (usually `None`) instead of raising a `KeyError`.

We can access dictionary items using the `get()` method by specifying the key as an argument. If the key exists in the dictionary, the method returns the associated value; otherwise, it returns a default value, which is often `None` unless specified otherwise.

## Syntax

Following is the syntax of the `get()` method in Python –

```
Val = dict.get("key")
```

where, key is an immutable object used as key in the dictionary object.

## Example 1

In the example below, we are defining a dictionary named "capitals" where each key-value pair maps a state to its capital city. Then, we use the get() method to retrieve the capital cities of "Gujarat" and "Karnataka" –

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar",
"Telangana": "Hyderabad", "Karnataka": "Bengaluru"}  
print ("Capital of Gujarat is: ", capitals.get('Gujarat'))  
print ("Capital of Karnataka is: ", capitals.get('Karnataka'))
```

We get the output as shown below –

Capital of Gujarat is: Gandhinagar  
Capital of Karnataka is: Bengaluru

## Example 2

Unlike the "[]" operator, the `get()` method doesn't raise error if the key is not found; it return `None` –

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar",  
"Telangana": "Hyderabad", "Karnataka": "Bengaluru"}  
print ("Capital of Haryana is : ", capitals.get('Haryana'))
```

It will produce the following output –

Capital of Haryana is : None

### Example 3

The `get()` method accepts an optional string argument. If it is given, and if the key is not found, this string becomes the return value –

```
capitals = {"Maharashtra": "Mumbai", "Gujarat": "Gandhinagar",
            "Telangana": "Hyderabad", "Karnataka": "Bengaluru"}
print ("Capital of Haryana is : ", capitals.get('Haryana', 'Not found'))
```

After executing the above code, we get the following output –

```
Capital of Haryana is: Not found
```

### Access Dictionary Keys

In a dictionary, keys are the unique identifiers associated with each value. They act as labels or indices that allow you to access and retrieve the corresponding value. Keys are immutable, meaning they cannot be changed once they are assigned. They must be of an immutable data type, such as strings, numbers, or tuples.

We can access dictionary keys in Python using the `keys()` method, which returns a view object containing all the keys in the dictionary.

#### Example

In this example, we are retrieving all the keys from the dictionary "student\_info" using the `keys()` method –

```
# Creating a dictionary with keys and values
student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}
# Accessing all keys using the keys() method
all_keys = student_info.keys()
print("Keys:", all_keys)
```

Following is the output of the above code –

```
Keys: dict_keys(['name', 'age', 'major'])
```

### Access Dictionary Values

In a dictionary, values are the data associated with each unique key. They represent the actual information stored in the dictionary and can be of any data type, such as strings, integers, lists, other dictionaries, and more. Each key in a dictionary maps to a specific value, forming a key-value pair.

We can access dictionary values in Python using –

- **Square Brackets ([])** – By providing the key inside the brackets.

- **The get() Method** – By calling the method with the key as an argument, optionally providing a default value.
- **The values() Method** – which returns a view object containing all the values in the dictionary

### Example 1

In this example, we are directly accessing associated with the key "name" and "age" using the square brackets –

```
# Creating a dictionary with student information
student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}
# Accessing dictionary values using square brackets
name = student_info["name"]
age = student_info["age"]
print("Name:", name)
print("Age:", age)
```

Output of the above code is as follows –

```
Name: Alice
Age: 21
```

### Example 2

Here, we use the get() method to retrieve the value associated with the key "major" and provide a default value of "2023" for the key "graduation\_year" –

```
# Creating a dictionary with student information
student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}
# Accessing dictionary values using the get() method
major = student_info.get("major")
# Default value provided if key is not found
grad_year = student_info.get("graduation_year", "2023")

print("Major:", major)
```

```
print("Graduation Year:", grad_year)
```

We get the result as follows –

```
Major: Computer Science
Graduation Year: 2023
```

### Example 3

Now, we are retrieving all the values from the dictionary "student\_info" using the values() method –

```
# Creating a dictionary with keys and values
student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}
# Accessing all values using the values() method
all_values = student_info.values()
print("Values:", all_values)
```

The result obtained is as shown below –

```
Values: dict_values(['Alice', 21, 'Computer Science'])
```

## Access Dictionary Items Using the items() Function

The items() function in Python is used to return a view object that displays a list of a dictionary's key-value tuple pairs.

This view object can be used to iterate over the dictionary's keys and values simultaneously, making it easy to access both the keys and the values in a single loop.

### Example

In the following example, we are using the items() function to retrieve all the key-value pairs from the dictionary "student\_info" –

```
# Creating a dictionary with student information
student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}

# Using the items() method to get key-value pairs
all_items = student_info.items()
```

```
print("Items:", all_items)
# Iterating through the key-value pairs
print("Iterating through key-value pairs:")
for key, value in all_items:
    print(f"{key}: {value}")
```

Following is the output of the above code –

```
Items: dict_items([('name', 'Alice'), ('age', 21), ('major', 'Computer
Science')])

Iterating through key-value pairs:
name: Alice
age: 21
major: Computer Science
```

# 94. Python - Change Dictionary Items

## Change Dictionary Items

Changing dictionary items in Python refers to modifying the values associated with specific keys within a dictionary. This can involve updating the value of an existing key, adding a new key-value pair, or removing a key-value pair from the dictionary.

Dictionaries are mutable, meaning their contents can be modified after they are created.

## Modifying Dictionary Values

Modifying values in a Python dictionary refers to changing the value associated with an existing key. To achieve this, you can directly assign a new value to that key.

### Example

In the following example, we are defining a dictionary named "person" with keys 'name', 'age', and 'city' and their corresponding values. Then, we modify the value associated with the key 'age' to 26 –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# Modifying the value associated with the key 'age'
person['age'] = 26
print(person)
```

It will produce the following output –

```
{'name': 'Alice', 'age': 26, 'city': 'New York'}
```

## Updating Multiple Dictionary Values

If you need to update multiple values in a dictionary at once, you can use the `update()` method. This method is used to update a dictionary with elements from another dictionary or an iterable of key-value pairs.

The `update()` method adds the key-value pairs from the provided dictionary or iterable to the original dictionary, overwriting any existing keys with the new values if they already exist in the original dictionary.

### Example

In the example below, we are using the `update()` method to modify the values associated with the keys 'age' and 'city' in the 'persons' dictionary –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# Updating multiple values
```

```
person.update({'age': 26, 'city': 'Los Angeles'})
print(person)
```

We get the output as shown below –

```
{'name': 'Alice', 'age': 26, 'city': 'Los Angeles'}
```

## Conditional Dictionary Modification

Conditional modification in a Python dictionary refers to changing the value associated with a key only if a certain condition is met.

You can use an if statement to check whether a certain condition is true before modifying the value associated with a key.

### Example

In this example, we conditionally modify the value associated with the key 'age' to '26' if the current value is '25' in the 'persons' dictionary –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# Conditionally modifying the value associated with 'age'
if person['age'] == 25:
    person['age'] = 26
print(person)
```

The output obtained is as shown below –

```
{'name': 'Alice', 'age': 26, 'city': 'New York'}
```

## Modify Dictionary by Adding New Key-Value Pairs

Adding new key-value pairs to a Python dictionary refers to inserting a new key along with its corresponding value into the dictionary.

This process allows you to dynamically expand the data stored in the dictionary by including additional information as needed.

### Example: Using Assignment Operator

You can add a new key-value pair to a dictionary by directly assigning a value to a new key as shown below. In the example below, the key 'city' with the value 'New York' is added to the 'person' dictionary –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25}
# Adding a new key-value pair 'city': 'New York'
person['city'] = 'New York'
print(person)
```

The result produced is as follows –

```
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

### Example: Using the setdefault() Method

You can use the `setdefault()` method to add a new key-value pair to a dictionary if the key does not already exist.

In this example, the `setdefault()` method adds the new key 'city' with the value 'New York' to the 'person' dictionary only if the key 'city' does not already exist –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25}
# Adding a new key-value pair 'city': 'New York'
person.setdefault('city', 'New York')
print(person)
```

Following is the output of the above code –

```
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

### Modify Dictionary by Removing Key-Value Pairs

Removing key-value pairs from a Python dictionary refers to deleting specific keys along with their corresponding values from the dictionary.

This process allows you to selectively remove data from the dictionary based on the keys you want to eliminate.

### Example: Using the del Statement

You can use the `del` statement to remove a specific key-value pair from a dictionary. In this example, the `del` statement removes the key 'age' along with its associated value from the 'person' dictionary –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# Removing the key-value pair associated with the key 'age'
del person['age']
print(person)
```

Output of the above code is as shown below –

```
{'name': 'Alice', 'city': 'New York'}
```

### Example: Using the pop() Method

You can also use the `pop()` method to remove a specific key-value pair from a dictionary and return the value associated with the removed key.

In here, the `pop()` method removes the key 'age' along with its associated value from the 'person' dictionary –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
# Removing the key-value pair associated with the key 'age'
removed_age = person.pop('age')

print(person)
print("Removed age:", removed_age)
```

It will produce the following output –

```
{'name': 'Alice', 'city': 'New York'}
Removed age: 25
```

### **Example: Using the popitem() Method**

You can use the `popitem()` method as well to remove the last key-value pair from a dictionary and return it as a tuple.

Now, the `popitem()` method removes the last key-value pair from the `'person'` dictionary and returns it as a tuple –

```
# Initial dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
# Removing the last key-value pair
removed_item = person.popitem()

print(person)
print("Removed item:", removed_item)
```

We get the output as shown below –

```
{'name': 'Alice', 'age': 25}
Removed item: ('city', 'New York')
```

# 95. Python - Add Dictionary Items

## Add Dictionary Items

Adding dictionary items in Python refers to inserting new key-value pairs into an existing dictionary. Dictionaries are mutable data structures that store collections of key-value pairs, where each key is associated with a corresponding value.

Adding items to a dictionary allows you to dynamically update and expand its contents as needed during program execution.

We can add dictionary items in Python using various ways such as –

- Using square brackets
- Using the update() method
- Using a comprehension
- Using unpacking
- Using the Union Operator
- Using the |= Operator
- Using setdefault() method
- Using collections.defaultdict() method

### Add Dictionary Item Using Square Brackets

The square brackets [] in Python is used to access elements in sequences like lists and strings through indexing and slicing operations. Additionally, when working with dictionaries, square brackets are used to specify keys for accessing or modifying associated values.

You can add items to a dictionary by specifying the key within square brackets and assigning a value to it. If the key is already present in the dictionary object, its value will be updated to val. If the key is not present in the dictionary, a new key-value pair will be added.

#### Example

In this example, we are creating a dictionary named "marks" with keys representing names and their corresponding integer values. Then, we add a new key-value pair 'Kavta': 58 to the dictionary using square bracket notation –

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}  
print ("Initial dictionary: ", marks)  
marks['Kavya'] = 58  
print ("Dictionary after new addition: ", marks)
```

It will produce the following output –

```
Initial dictionary: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}  
Dictionary after new addition: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91,  
'David': 49, 'Kavya': 58}
```

## Add Dictionary Item Using the update() Method

The `update()` method in Python dictionaries is used to merge the contents of another dictionary or an iterable of key-value pairs into the current dictionary. It adds or updates key-value pairs, ensuring that existing keys are updated with new values and new keys are added to the dictionary.

You can add multiple items to a dictionary using the `update()` method by passing another dictionary or an iterable of key-value pairs.

### Example

In the following example, we use the `update()` method to add multiple new key-value pairs '`'Kavya': 58` and '`'Mohan': 98`' to the dictionary '`'marks'`' –

```
marks = {"Savita":67, "Imtiaz":88}
print ("Initial dictionary: ", marks)
marks.update({'Kavya': 58, 'Mohan': 98})
print ("Dictionary after new addition: ", marks)
```

We get the output as shown below –

```
Initial dictionary: {'Savita': 67, 'Imtiaz': 88}
Dictionary after new addition: {'Savita': 67, 'Imtiaz': 88, 'Kavya': 58,
'Mohan': 98}
```

## Add Dictionary Item Using Unpacking

Unpacking in Python refers to extracting individual elements from a collection, such as a list, tuple, or dictionary, and assigning them to variables in a single statement. This can be done using the `*` operator for iterables like lists and tuples, and the `**` operator for dictionaries.

We can add dictionary items using unpacking by combining two or more dictionaries with the `**` unpacking operator.

### Example

In the example below, we are initializing two dictionaries named "`"marks"`" and "`"marks1"`", both containing names and their corresponding integer values. Then, we create a new dictionary "`"newmarks"`" by merging "`"marks"`" and "`"marks1"`" using dictionary unpacking –

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
newmarks = {**marks, **marks1}
print ("marks dictionary after update: \n", newmarks)
```

Following is the output of the above code –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
```

```
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51,
'Mushtaq': 61}
```

## Add Dictionary Item Using the Union Operator (|)

The union operator in Python, represented by the | symbol, is used to combine the elements of two sets into a new set that contains all the unique elements from both sets. It can also be used with dictionaries in Python 3.9 and later to merge the contents of two dictionaries.

We can add dictionary items using the union operator by merging two dictionaries into a new dictionary, which includes all key-value pairs from both dictionaries.

### Example

In this example, we are using the | operator to combine the dictionaries "marks" and "marks1" with "marks1" values taking precedence in case of duplicate keys –

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
newmarks = marks | marks1
print ("marks dictionary after update: \n", newmarks)
```

Output of the above code is as shown below –

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51,
'Mushtaq': 61}
```

## Add Dictionary Item Using the "|=" Operator

The |= operator in Python is an in-place union operator for sets and dictionaries. It updates the set or dictionary on the left-hand side with elements from the set or dictionary on the right-hand side.

We can add dictionary items using the |= operator by updating an existing dictionary with key-value pairs from another dictionary. If there are overlapping keys, the values from the right-hand dictionary will overwrite those in the left-hand dictionary.

### Example

In the following example, we use the |= operator to update "marks" with the key-value pairs from "marks1", with values from "marks1" taking precedence in case of duplicate keys –

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
marks |= marks1
```

```
print ("marks dictionary after update: \n", marks)
```

The output produced is as shown below –

```
marks dictionary before update:  

{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}  

marks dictionary after update:  

{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51,
'Mushtaq': 61}
```

## Add Dictionary Item Using the `setdefault()` Method

The `setdefault()` method in Python is used to get the value of a specified key in a dictionary. If the key does not exist, it inserts the key with a specified default value.

We can add dictionary items using the `setdefault()` method by specifying a key and a default value.

### Example

In this example, we use the `setdefault()` to add the key-value pair "major": "Computer Science" to the "student" dictionary –

```
# Initial dictionary  

student = {"name": "Alice", "age": 21}  

# Adding a new key-value pair  

major = student.setdefault("major", "Computer Science")  

print(student)
```

Since the key "major" does not exist, it is added with the specified default value as shown in the output below –

```
{'name': 'Alice', 'age': 21, 'major': 'Computer Science'}
```

## Add Dictionary Item Using the `collections.defaultdict()` Method

The `collections.defaultdict()` method in Python is a subclass of the built-in "dict" class that creates dictionaries with default values for keys that have not been set yet. It is part of the `collections` module in Python's standard library.

We can add dictionary items using the `collections.defaultdict()` method by specifying a default factory, which determines the default value for keys that have not been set yet. When accessing a missing key for the first time, the default factory is called to create a default value, and this value is inserted into the dictionary.

### Example

In this example, we are initializing instances of `defaultdict` with different default factories: `int` to initialize missing keys with 0, `list` to initialize missing keys with an empty list, and a custom function `default_value` to initialize missing keys with the return value of the function –

```
from collections import defaultdict
```

```
# Using int as the default factory to initialize missing keys with 0
d = defaultdict(int)
# Incrementing the value for key 'a'
d["a"] += 1
print(d)

# Using list as the default factory to initialize missing keys with an empty
list
d = defaultdict(list)
# Appending to the list for key 'b'
d["b"].append(1)
print(d)

# Using a custom function as the default factory
def default_value():
    return "N/A"

d = defaultdict(default_value)
print(d["c"])
```

The output obtained is as follows –

```
defaultdict(<class 'int'>, {'a': 1})
defaultdict(<class 'list'>, {'b': [1]}
N/A
```

# 96. Python - Remove Dictionary Items

## Remove Dictionary Items

Removing dictionary items in Python refers to deleting key-value pairs from an existing dictionary. Dictionaries are mutable data structures that hold pairs of keys and their associated values. Each key acts as a unique identifier, mapping to a specific value within the dictionary.

Removing items from a dictionary allows you to eliminate unnecessary or unwanted data from the dictionary, thereby reducing its size and modifying its content.

We can remove dictionary items in Python using various ways such as –

- using the `del` keyword
- using the `pop()` method
- using the `popitem()` method
- using the `clear()` method
- using dictionary comprehension

### Remove Dictionary Items Using `del` Keyword

The `del` keyword in Python is used to delete objects. In the context of dictionaries, it is used to remove an item or a slice of items from the dictionary, based on the specified key(s).

We can remove dictionary items using the `del` keyword by specifying the key of the item we want to remove. This will delete the key-value pair associated with the specified key from the dictionary.

#### Example 1

In the following example, we are creating a dictionary named `numbers` with integer keys and their corresponding string values. Then, delete the item with the key '20' using the `del` keyword –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}  
print ("numbers dictionary before delete operation: \n", numbers)  
del numbers[20]  
print ("numbers dictionary before delete operation: \n", numbers)
```

It will produce the following output –

```
numbers dictionary before delete operation:  
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}  
numbers dictionary before delete operation:  
{10: 'Ten', 30: 'Thirty', 40: 'Forty'}
```

#### Example 2

The `del` keyword, when used with a dictionary object, removes the dictionary from memory –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before delete operation: \n", numbers)
del numbers
print ("numbers dictionary before delete operation: \n", numbers)
```

Following is the output obtained –

```
numbers dictionary before delete operation:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
Traceback (most recent call last):
  File "C:\Users\mlath\examples\main.py", line 5, in <module>
    print ("numbers dictionary before delete operation: \n", numbers)
                                         ^
NameError: name 'numbers' is not defined
```

## Remove Dictionary Items Using `pop()` Method

The `pop()` method in Python is used to remove a specified key from a dictionary and return the corresponding value. If the specified key is not found, it can optionally return a default value instead of raising a `KeyError`.

We can remove dictionary items using the `pop()` method by specifying the key of the item we want to remove. This method will return the value associated with the specified key and remove the key-value pair from the dictionary.

### Example

In this example, we are using the `pop()` method to remove the item with the key '20' (storing its value in `val`) from the 'numbers' dictionary. We then retrieve the updated dictionary and the popped value –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before pop operation: \n", numbers)
val = numbers.pop(20)
print ("numbers dictionary after pop operation: \n", numbers)
print ("Value popped: ", val)
```

Following is the output of the above code –

```
numbers dictionary before pop operation:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after pop operation:
{10: 'Ten', 30: 'Thirty', 40: 'Forty'}
Value popped: Twenty
```

## Remove Dictionary Items Using `popitem()` Method

The `popitem()` method in Python is used to remove and return the last key-value pair from a dictionary.

*Since Python 3.7, dictionaries maintain the insertion order, so `popitem()` removes the most recently added item. If the dictionary is empty, calling `popitem()` raises a `KeyError`.*

We can remove dictionary items using the `popitem()` method by calling the method on the dictionary, which removes and returns the last key-value pair added to the dictionary.

### Example

In the example below, we use the `popitem()` method to remove an arbitrary item from the dictionary 'numbers' (storing both its key-value pair in `val`), and retrieve the updated dictionary along with the popped key-value pair –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}
print ("numbers dictionary before pop operation: \n", numbers)
val = numbers.popitem()
print ("numbers dictionary after pop operation: \n", numbers)
print ("Value popped: ", val)
```

Output of the above code is as shown below –

```
numbers dictionary before pop operation:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after pop operation:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty'}
Value popped: (40, 'Forty')
```

## Remove Dictionary Items Using `clear()` Method

The `clear()` method in Python is used to remove all items from a dictionary. It effectively empties the dictionary, leaving it with a length of 0.

We can remove dictionary items using the `clear()` method by calling it on the dictionary object. This method removes all key-value pairs from the dictionary, effectively making it empty.

### Example

In the following example, we are using the `clear()` method to remove all items from the dictionary 'numbers' –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}
print ("numbers dictionary before clear method: \n", numbers)
numbers.clear()
print ("numbers dictionary after clear method: \n", numbers)
```

We get the output as shown below –

```

numbers dictionary before clear method:
{10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after clear method:
{}

```

## Remove Dictionary Items using Dictionary Comprehension

Dictionary comprehension is a concise way to create dictionaries in Python. It follows the same syntax as list comprehension but generates dictionaries instead of lists. With dictionary comprehension, you can iterate over iterable objects (such as lists, tuples, or other dictionaries), apply an expression to each item, and construct key-value pairs based on the result of that expression.

*We cannot directly remove dictionary items using dictionary comprehension. Dictionary comprehension is primarily used for creating new dictionaries based on some transformation or filtering of existing data, rather than for removing items from dictionaries.*

If you need to remove items from a dictionary based on certain conditions, you would typically use other methods like `del`, `pop()`, or `popitem()`. These methods allow you to explicitly specify which items to remove from the dictionary.

### Example

In this example, we remove items 'age' and 'major' from the 'student\_info' dictionary based on a predefined list of keys to remove –

```

# Creating a dictionary
student_info = {
    "name": "Alice",
    "age": 21,
    "major": "Computer Science"
}

# Removing items based on conditions
keys_to_remove = ["age", "major"]
for key in keys_to_remove:
    student_info.pop(key, None)

print(student_info)

```

The output obtained is as shown below –

```
{'name': 'Alice'}
```

# 97. Python - Dictionary View Objects

The items(), keys(), and values() methods of dict class return view objects. These views are refreshed dynamically whenever any change occurs in the contents of their source dictionary object.

## The items() Method

The items() method returns a dict\_items view object. It contains a list of tuples, each tuple made up of respective key, value pairs.

### Syntax

Following is the syntax of the items() method –

```
Obj = dict.items()
```

### Return value

The items() method returns dict\_items object which is a dynamic view of (key,value) tuples.

### Example

In the following example, we first obtain the dict\_items object with items() method and check how it is dynamically updated when the dictionary object is updated.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
obj = numbers.items()  
print ('type of obj: ', type(obj))  
print (obj)  
print ("update numbers dictionary")  
numbers.update({50:"Fifty"})  
print ("View automatically updated")  
print (obj)
```

It will produce the following output –

```
type of obj: <class 'dict_items'>  
dict_items([(10, 'Ten'), (20, 'Twenty'), (30, 'Thirty'), (40, 'Forty')])  
update numbers dictionary  
View automatically updated  
dict_items([(10, 'Ten'), (20, 'Twenty'), (30, 'Thirty'), (40, 'Forty'), (50,  
'Fifty')])
```

## The keys() Method

The `keys()` method of `dict` class returns `dict_keys` object which is a list of all keys defined in the dictionary. It is a view object, as it gets automatically updated whenever any update action is done on the dictionary object.

### Syntax

Following is the syntax of the `keys()` method –

```
Obj = dict.keys()
```

### Return value

The `keys()` method returns `dict_keys` object which is a view of keys in the dictionary.

### Example

In this example, we are creating a dictionary named "numbers" with integer keys and their corresponding string values. Then, we obtain a view object "obj" of the keys using the `keys()` method, and retrieve its type and content –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
obj = numbers.keys()  
print ('type of obj: ', type(obj))  
print (obj)  
print ("update numbers dictionary")  
numbers.update({50:"Fifty"})  
print ("View automatically updated")  
print (obj)
```

It will produce the following output –

```
type of obj: <class 'dict_keys'>  
dict_keys([10, 20, 30, 40])  
update numbers dictionary  
View automatically updated  
dict_keys([10, 20, 30, 40, 50])
```

## The values() Method

The `values()` method returns a view of all the values present in the dictionary. The object is of `dict_values` type, which gets automatically updated.

### Syntax

Following is the syntax of the `values()` method –

```
Obj = dict.values()
```

### Return value

The `values()` method returns a `dict_values` view of all the values present in the dictionary.

**Example**

In the example below, we obtain a view object "obj" of the values using the values() method from the "numbers" dictionary –

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty", 40:"Forty"}  
obj = numbers.values()  
print ('type of obj: ', type(obj))  
print (obj)  
print ("update numbers dictionary")  
numbers.update({50:"Fifty"})  
print ("View automatically updated")  
print (obj)
```

It will produce the following output –

```
type of obj: <class 'dict_values'>  
dict_values(['Ten', 'Twenty', 'Thirty', 'Forty'])  
update numbers dictionary  
View automatically updated  
dict_values(['Ten', 'Twenty', 'Thirty', 'Forty', 'Fifty'])
```

# 98. Python - Loop Dictionaries

## Loop Through Dictionaries

Looping through dictionaries in Python refers to iterating over key-value pairs within the dictionary and performing operations on each pair. This allows you to access both keys and their corresponding values. There are several ways/methods for looping through dictionaries –

- Using a for Loop
- Using dict.items() method
- Using dict.keys() method
- Using dict.values() method

### Loop Through Dictionary Using a For Loop

A for loop in Python is a control flow statement that iterates over a sequence of elements. It repeatedly executes a block of code for each item in the sequence. The sequence can be a range of numbers, a list, a tuple, a string, or any iterable object.

We can loop through dictionaries using a for loop in Python by iterating over the keys or key-value pairs within the dictionary. There are two common approaches –

#### Example: Iterating over Keys

In this approach, the loop iterates over the keys of the dictionary. Inside the loop, you can access the value corresponding to each key using dictionary indexing –

```
student = {"name": "Alice", "age": 21, "major": "Computer Science"}  
for key in student:  
    print(key, student[key])
```

It will produce the following output –

```
name Alice  
age 21  
major Computer Science
```

#### Example: Iterating over Key-Value Pairs

In this approach, the loop iterates over the key-value pairs using the items() method of the dictionary. Each iteration provides both the key and its corresponding value –

```
student = {"name": "Alice", "age": 21, "major": "Computer Science"}  
for key, value in student.items():  
    print(key, value)
```

We get the output as shown below –

```
name Alice
```

```
age 21
major Computer Science
```

## Loop through Dictionary Using dict.items() Method

The `dict.items()` method in Python is used to return a view object that displays a list of key-value pairs in the dictionary. This view object provides a dynamic view of the dictionary's items, allowing you to access both the keys and their corresponding values.

We can loop through dictionaries using the `dict.items()` method by iterating over the key-value pairs returned by this method.

### Example

In this example, the `items()` method is called on the "student" dictionary, returning a view object containing the key-value pairs. The for loop iterates over each pair, assigning the key to the variable "key" and the corresponding value to the variable "value" –

```
student = {"name": "Alice", "age": 21, "major": "Computer Science"}

# Looping through key-value pairs
for key, value in student.items():
    print(key, value)
```

The output produced is as shown below –

```
name Alice
age 21
major Computer Science
```

## Loop through Dictionary Using dict.keys() Method

The `dict.keys()` method in Python is used to return a view object that displays a list of keys in the dictionary. This view object provides a dynamic view of the dictionary's keys, allowing you to access and iterate over them.

We can loop through dictionaries using the `dict.keys()` method by iterating over the keys returned by this method. This allows us to access and iterate over the keys of the dictionary.

### Example

In the example below, the `keys()` method is called on the "student" dictionary, returning a view object containing the keys. The for loop iterates over each key in the view object, allowing you to perform operations based on the keys of the dictionary during each iteration –

```
student = {"name": "Alice", "age": 21, "major": "Computer Science"}

# Looping through keys
for key in student.keys():
```

```
print(key)
```

Following is the output of the above code –

```
name  
age  
major
```

### Loop through Dictionary Using dict.values() Method

The `dict.values()` method in Python is used to return a view object that displays a list of values in the dictionary. This view object provides a dynamic view of the dictionary's values, allowing you to access and iterate over them.

We can loop through dictionaries using the `dict.values()` method by iterating over the values returned by this method. This allows us to access and iterate over the values of the dictionary.

#### Example

In the following example, the `values()` method is called on the "student" dictionary, returning a view object containing the values –

```
student = {"name": "Alice", "age": 21, "major": "Computer Science"}  
  
# Looping through values  
for value in student.values():  
    print(value)
```

Output of the above code is as shown below –

```
Alice  
21  
Computer Science
```

# 99. Python - Copy Dictionaries

## Copy Dictionaries

Copying dictionaries in Python refers to creating a new dictionary that contains the same key-value pairs as the original dictionary.

We can copy dictionaries using various ways, depending on the requirements and the nature of the dictionary's values (whether they are mutable or immutable, nested or not).

### Shallow Copy

When you perform a shallow copy, a new dictionary object is created, but it contains references to the same objects as the original dictionary references.

This is useful when you want to duplicate the structure of a dictionary without duplicating the nested objects it contains.

This can be done using the `copy()` method or the `dict()` function as shown below –

#### Example: Using the `copy()` Method

In the following example, we can see that changing the "age" in the shallow copy does not affect the original.

However, modifying the list in the shallow copy also affects the original because the list is a mutable object and only a reference is copied.

```
original_dict = {"name": "Alice", "age": 25, "skills": ["Python", "Data Science"]}

shallow_copy = original_dict.copy()

# Modifying the shallow copy
shallow_copy["age"] = 26
shallow_copy["skills"].append("Machine Learning")

print("Original dictionary:", original_dict)
print("Shallow copy:", shallow_copy)
```

Following is the output of the above code –

```
Original dictionary: {'name': 'Alice', 'age': 25, 'skills': ['Python', 'Data Science', 'Machine Learning']}
Shallow copy: {'name': 'Alice', 'age': 26, 'skills': ['Python', 'Data Science', 'Machine Learning']}
```

#### Example: Using the `dict()` Method

Similar to the `copy()` method, the `dict()` method creates a shallow copy as shown in the example below –

```

original_dict = {"name": "Bob", "age": 30, "skills": ["Java", "C++"]}

shallow_copy = dict(original_dict)

# Modifying the shallow copy
shallow_copy["age"] = 31
shallow_copy["skills"].append("C#")

print("Original dictionary:", original_dict)
print("Shallow copy:", shallow_copy)

```

Output of the above code is as follows –

```

Original dictionary: {'name': 'Bob', 'age': 30, 'skills': ['Java', 'C++', 'C#']}
Shallow copy: {'name': 'Bob', 'age': 31, 'skills': ['Java', 'C++', 'C#']}

```

## Deep Copy

A deep copy creates a new dictionary and recursively copies all objects found in the original dictionary. This means that not only the dictionary itself but also all objects it contains (including nested dictionaries, lists, etc.) are copied. As a result, changes made to the deep copy do not affect the original dictionary and vice versa.

We can achieve this using the `deepcopy()` function in the `copy` module.

### Example

We can see in the example below that the "age" value in the deep copy is changed, the "skills" list in the deep copy is modified (an item is appended) and the "education" dictionary in the deep copy is modified, all without affecting the original –

```

import copy

original_dict = {
    "name": "Alice",
    "age": 25,
    "skills": ["Python", "Data Science"],
    "education": {
        "degree": "Bachelor's",
        "field": "Computer Science"
    }
}

# Creating a deep copy
deep_copy = copy.deepcopy(original_dict)

```

```
# Modifying the deep copy
deep_copy["age"] = 26
deep_copy["skills"].append("Machine Learning")
deep_copy["education"]["degree"] = "Master's"

# Retrieving both dictionaries
print("Original dictionary:", original_dict)
print("Deep copy:", deep_copy)
```

This will produce the following output –

```
Original dictionary: {'name': 'Alice', 'age': 25, 'skills': ['Python', 'Data Science'], 'education': {'degree': 'Bachelor's', 'field': 'Computer Science'}}
Deep copy: {'name': 'Alice', 'age': 26, 'skills': ['Python', 'Data Science', 'Machine Learning'], 'education': {'degree': "Master's", 'field': 'Computer Science'}}}
```

## Copy Dictionaries Using `copy()` Method

Dictionaries cannot be copied directly by using the assignment operator (=), you can use the `copy()` method to create a shallow copy of a dictionary.

### Syntax

Following is the basic syntax of the `copy()` method in Python –

```
new_dict = original_dict.copy()
```

Where, `original_dict` is the dictionary you want to copy.

### Example

The following example demonstrates the creation of a shallow copy of a dictionary using the `copy()` method –

```
# Creating a dictionary
dict1 = {"name": "Krishna", "age": "27", "doy": 1992}

# Copying the dictionary
dict2 = dict1.copy()

# Printing both of the dictionaries
print("dict1 :", dict1)
print("dict2 :", dict2)
```

**Output**

We will get the output as shown below –

```
dict1 : {'name': 'Krishna', 'age': '27', 'doy': 1992}  
dict2 : {'name': 'Krishna', 'age': '27', 'doy': 1992}
```

# 100. Python - Nested Dictionaries

## Nested Dictionaries

Nested dictionaries in Python refer to dictionaries that are stored as values within another dictionary. In other words, a dictionary can contain other dictionaries as its values, forming a hierarchical or nested structure.

Nested dictionaries can be modified, updated, or extended in the same way as regular dictionaries. You can add, remove, or update key-value pairs at any level of the nested structure.

### Creating a Nested Dictionary in Python

We can create a nested dictionary in Python by defining a dictionary where the values of certain keys are themselves dictionaries. This allows for the creation of a hierarchical structure where each key-value pair represents a level of nested information. This can be achieved in several ways –

#### Example: Direct Assignment

In this approach, we can directly assign dictionaries as values to outer keys within a single dictionary definition –

```
# Define the outer dictionary
nested_dict = {
    "outer_key1": {"inner_key1": "value1", "inner_key2": "value2"},
    "outer_key2": {"inner_key3": "value3", "inner_key4": "value4"}
}
print(nested_dict)
```

#### Example: Using a Loop

With this method, an empty outer dictionary is initialized, and then populated with dictionaries as values using a loop to define nested dictionaries –

```
# Define an empty outer dictionary
nested_dict = {}

# Add key-value pairs to the outer dictionary
outer_keys = ["outer_key1", "outer_key2"]
for key in outer_keys:
    nested_dict[key] = {"inner_key1": "value1", "inner_key2": "value2"}
print(nested_dict)
```

## Adding Items to a Nested Dictionary in Python

Once a nested dictionary is created, we can add items to it by accessing the specific nested dictionary using its key and then assigning a new key-value pair to it.

In the following example, we are defining a nested dictionary "students" where each key represents a student's name and its value is another dictionary containing details about the student.

Then, we add a new key-value pair to Alice's nested dictionary and add a new nested dictionary for a new student, Charlie –

```
# Initial nested dictionary
students = {
    "Alice": {"age": 21, "major": "Computer Science"},
    "Bob": {"age": 20, "major": "Engineering"}
}

# Adding a new key-value pair to Alice's nested dictionary
students["Alice"]["GPA"] = 3.8

# Adding a new nested dictionary for a new student
students["Charlie"] = {"age": 22, "major": "Mathematics"}

print(students)
```

It will produce the following output –

```
{'Alice': {'age': 21, 'major': 'Computer Science', 'GPA': 3.8}, 'Bob': {'age': 20, 'major': 'Engineering'}, 'Charlie': {'age': 22, 'major': 'Mathematics'}}
```

## Accessing Items of a Nested Dictionary in Python

Accessing items of a nested dictionary in Python refers to retrieving values stored within the nested structure by using a series of keys. Each key corresponds to a level in the hierarchy of the dictionary.

We can achieve this through direct indexing with square brackets or by using the `get()` method

### Example: Using Direct Indexing

In this approach, we access values in a nested dictionary by specifying each key in a sequence of square brackets. Each key in the sequence refers to a level in the nested dictionary, progressing one level deeper with each key –

```
# Define a nested dictionary
students = {
    "Alice": {"age": 21, "major": "Computer Science"},
```

```

    "Bob": {"age": 20, "major": "Engineering"},  

    "Charlie": {"age": 22, "major": "Mathematics"}  

}  
  

# Access Alice's major  

alice_major = students["Alice"]["major"]  

print("Alice's major:", alice_major)  
  

# Access Bob's age  

bob_age = students["Bob"]["age"]  

print("Bob's age:", bob_age)

```

Following is the output of the above code –

```

Alice's major: Computer Science  

Bob's age: 20

```

### **Example: Using the get() Method**

The `get()` method is used to fetch the value associated with the specified key. If the key does not exist, it returns a default value (which is `None` if not specified) –

```

# Define a nested dictionary  

students = {  

    "Alice": {"age": 21, "major": "Computer Science"},  

    "Bob": {"age": 20, "major": "Engineering"},  

    "Charlie": {"age": 22, "major": "Mathematics"}  

}  
  

# Access Alice's major using .get()  

alice_major = students.get("Alice", {}).get("major", "Not Found")  

print("Alice's major:", alice_major)  
  

# Safely access a non-existing key using .get()  

dave_major = students.get("Dave", {}).get("major", "Not Found")  

print("Dave's major:", dave_major)

```

Output of the above code is as follows –

```

Alice's major: Computer Science  

Dave's major: Not Found

```

## Deleting a Dictionary from a Nested Dictionary

We can delete dictionaries from a nested dictionary by using the `del` keyword. This keyword allows us to remove a specific key-value pair from the nested dictionary.

### Example

In the following example, we delete the nested dictionary for "Bob" from "students" dictionary using the `del` statement –

```
# Define a nested dictionary
students = {
    "Alice": {"age": 21, "major": "Computer Science"},
    "Bob": {"age": 20, "major": "Engineering"},
    "Charlie": {"age": 22, "major": "Mathematics"}
}

# Delete the dictionary for Bob
del students["Bob"]

# Print the updated nested dictionary
print(students)
```

We get the output as shown below –

```
{'Alice': {'age': 21, 'major': 'Computer Science'}, 'Charlie': {'age': 22,
'major': 'Mathematics'}}
```

## Iterating Through a Nested Dictionary in Python

Iterating through a nested dictionary refers to looping through the keys and values at each level of the dictionary. This allows you to access and manipulate items within the nested structure.

We can iterate through a nested dictionary by using nested loops. The outer loop iterates over the keys and values of the main dictionary, while the inner loop iterates over the keys and values of the nested dictionaries.

### Example

In this example, we are iterating through the "students" dictionary, retrieving each student's name and their corresponding details by iterating through the nested dictionaries –

```
# Defining a nested dictionary
students = {
    "Alice": {"age": 21, "major": "Computer Science"},
    "Bob": {"age": 20, "major": "Engineering"},
    "Charlie": {"age": 22, "major": "Mathematics"}
```

```
}
```

```
# Iterating through the Nested Dictionary:  
for student, details in students.items():  
    print(f"Student: {student}")  
    for key, value in details.items():  
        print(f"  {key}: {value}")
```

The output obtained is as shown below –

```
Student: Alice  
  age: 21  
  major: Computer Science  
  
Student: Bob  
  age: 20  
  major: Engineering  
  
Student: Charlie  
  age: 22  
  major: Mathematics
```

# 101. Python - Dictionary Methods

A Python dictionary is an object of the built-in dict class, which defines the following methods –

## Dictionary Methods

| Sr.No. | Method and Description                                                                                                                             |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><code>dict.clear()</code></a><br>Removes all elements of dictionary dict.                                                              |
| 2      | <a href="#"><code>dict.copy()</code></a><br>Returns a shallow copy of dictionary dict.                                                             |
| 3      | <a href="#"><code>dict.fromkeys()</code></a><br>Create a new dictionary with keys from seq and values set to value.                                |
| 4      | <a href="#"><code>dict.get(key, default=None)</code></a><br>For key key, returns value or default if key not in dictionary.                        |
| 5      | <a href="#"><code>dict.has_key(key)</code></a><br>Returns true if a given key is available in the dictionary, otherwise it returns a false.        |
| 6      | <a href="#"><code>dict.items()</code></a><br>Returns a list of dict's (key, value) tuple pairs.                                                    |
| 7      | <a href="#"><code>dict.keys()</code></a><br>Returns list of dictionary dict's keys.                                                                |
| 8      | <a href="#"><code>dict.pop()</code></a><br>Removes the element with specified key from the collection                                              |
| 9      | <a href="#"><code>dict.popitem()</code></a><br>Removes the last inserted key-value pair                                                            |
| 10     | <a href="#"><code>dict.setdefault(key, default=None)</code></a><br>Similar to get(), but will set dict[key]=default if key is not already in dict. |
| 11     | <a href="#"><code>dict.update(dict2)</code></a><br>Adds dictionary dict2's key-values pairs to dict.                                               |
| 12     | <a href="#"><code>dict.values()</code></a><br>Returns list of dictionary dict's values.                                                            |

# 102. Python - Dictionary Exercises

## Dictionary Exercise 1

Python program to create a new dictionary by extracting the keys from a given dictionary.

```
d1 = {"one":11, "two":22, "three":33, "four":44, "five":55}  
keys = ['two', 'five']  
d2={}  
for k in keys:  
    d2[k]=d1[k]  
print (d2)
```

It will produce the following output –

```
{'two': 22, 'five': 55}
```

## Dictionary Exercise 2

Python program to convert a dictionary to list of (k,v) tuples.

```
d1 = {"one":11, "two":22, "three":33, "four":44, "five":55}  
L1 = list(d1.items())  
print (L1)
```

It will produce the following output –

```
[('one', 11), ('two', 22), ('three', 33), ('four', 44), ('five', 55)]
```

## Dictionary Exercise 3

Python program to remove keys with same values in a dictionary.

```
d1 = {"one":"eleven", "2":2, "three":3, "11":"eleven", "four":44, "two":2}  
vals = list(d1.values())#all values  
uvals = [v for v in vals if vals.count(v)==1]#unique values  
d2 = {}  
for k,v in d1.items():  
    if v in uvals:  
        d = {k:v}  
        d2.update(d)  
print ("dict with unique value:",d2)
```

It will produce the following output –

```
dict with unique value: {'three': 3, 'four': 44}
```

## Dictionary Exercise Programs

- Python program to sort list of dictionaries by values
- Python program to extract dictionary with each key having non-numeric value from a given dictionary.
- Python program to build a dictionary from list of two item (k,v) tuples.
- Python program to merge two dictionary objects, using unpack operator.

# Python Arrays

# 103. Python - Arrays

## Arrays in Python

Unlike other programming languages like C++ or Java, Python does not have built-in support for arrays. However, Python has several data types like lists and tuples (especially lists) that are often used as arrays but, items stored in these types of sequences need not be of the same type.

In addition, we can create and manipulate arrays the using the array module. Before proceeding further, let's understand arrays in general.

## What are arrays?

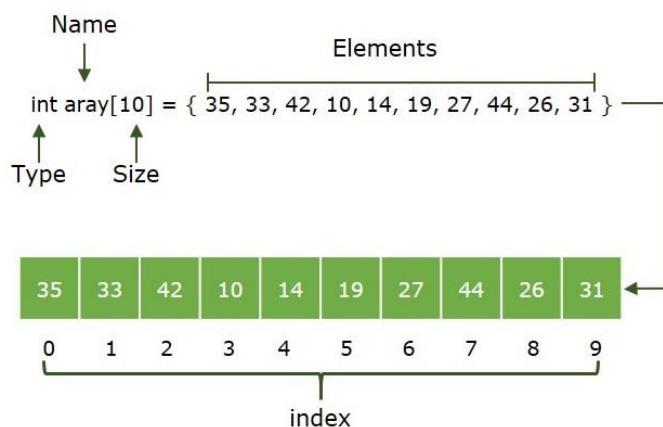
An array is a container which can hold a fix number of items and these items should be of the same type. Each item stored in an array is called an element and they can be of any type including integers, floats, strings, etc.

These elements are stored at contiguous memory location. Each location of an element in an array has a numerical index starting from 0. These indices are used to identify and access the elements.

## Array Representation

Arrays are represented as a collection of multiple containers where each container stores one element. These containers are indexed from '0' to 'n-1', where n is the size of that particular array.

Arrays can be declared in various ways in different languages. Below is an illustration –



As per the above illustration, following are the important points to be considered –

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

## Creating Array in Python

To create an array in Python, import the array module and use its array() function. We can create an array of three basic types namely integer, float and Unicode characters using this function.

The array() function accepts typecode and initializer as a parameter value and returns an object of array class.

### Syntax

The syntax for creating an array in Python is –

```
# importing
import array as array_name

# creating array
obj = array_name.array(typecode[, initializer])
```

Where,

- **typecode** – The typecode character used to specify the type of elements in the array.
- **initializer** – It is an optional value from which array is initialized. It must be a list, a bytes-like object, or iterable elements of the appropriate type.

### Example

The following example shows how to create an array in Python using the array module.

```
import array as arr

# creating an array with integer type
a = arr.array('i', [1, 2, 3])
print (type(a), a)

# creating an array with char type
a = arr.array('u', 'BAT')
print (type(a), a)

# creating an array with float type
a = arr.array('d', [1.1, 2.2, 3.3])
print (type(a), a)
```

It will produce the following output –

```
<class 'array.array'> array('i', [1, 2, 3])
<class 'array.array'> array('u', 'BAT')
<class 'array.array'> array('d', [1.1, 2.2, 3.3])
```

Python array type is decided by a single character Typecode argument. The type codes and the intended data type of array is listed below –

| typecode | Python data type  | Byte size |
|----------|-------------------|-----------|
| 'b'      | signed integer    | 1         |
| 'B'      | unsigned integer  | 1         |
| 'u'      | Unicode character | 2         |
| 'h'      | signed integer    | 2         |
| 'H'      | unsigned integer  | 2         |
| 'i'      | signed integer    | 2         |
| 'I'      | unsigned integer  | 2         |
| 'l'      | signed integer    | 4         |
| 'L'      | unsigned integer  | 4         |
| 'q'      | signed integer    | 8         |
| 'Q'      | unsigned integer  | 8         |
| 'f'      | floating point    | 4         |
| 'd'      | floating point    | 8         |

## Basic Operations on Python Arrays

Following are the basic operations supported by an array –

- **Traverse** – Print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

## Accessing Array Element

We can access each element of an array using the index of the element.

### Example

The below code shows how to access elements of an array.

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1[0])
print (array1[2])
```

When we compile and execute the above program, it produces the following result –

10

30

## Insertion Operation

In insertion operation, we insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

### Example

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.insert(1,60)
for x in array1:
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position 1.

```
10
60
20
30
40
50
```

## Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.remove(40)
for x in array1:
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is removed from the array.

```
10
20
30
50
```

## Search Operation

You can perform a search operation on an array to find an array element based on its value or its index.

### Example

Here, we search a data element using the python in-built index() method –

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1.index(40))
```

When we compile and execute the above program, it will display the index of the searched element. If the value is not present in the array, it will return an error.

```
3
```

## Update Operation

Update operation refers to updating an existing element from the array at a given index. Here, we simply reassign a new value to the desired index we want to update.

### Example

In this example, we are updating the value of array element at index 2.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1[2] = 80
for x in array1:
    print(x)
```

On executing the above program, it produces the following result which shows the new value at the index position 2.

```
10
20
80
40
50
```

# 104. Python - Access Array Items

Accessing an array item in Python refers to the process of retrieving the value stored at a specific index in the given array. Here, index is a numerical value that indicates the location of array items. Thus, you can use this index to access elements of an array in Python.

*An array is a container that holds a fix number of items of the same type.  
Python uses array module to achieve the functionality like an array.*

## Accessing array items in Python

You can use the following ways to access array items in Python –

- Using indexing
- Using iteration
- Using enumerate() function

### Using indexing

The process of accessing elements of an array through the index is known as Indexing. In this process, we simply need to pass the index number inside the index operator []. The index of an array in Python starts with 0 which means you can find its first element at index 0 and the last at one less than the length of given array.

#### Example

The following example shows how to access elements of an array using indexing.

```
import array as arr

# creating array
numericArray = arr.array('i', [111, 211, 311, 411, 511])

#indexing
print (numericArray[0])
print (numericArray[1])
print (numericArray[2])
```

When you run the above code, it will show the following output –

```
111
211
311
```

## Using iteration

In this approach, a block of code is executed repeatedly using loops such as for and while. It is used when you want to access array elements one by one.

### Example

In the below code, we use the for loop to access all the elements of the specified array.

```
import array as arr

# creating array
numericArray = arr.array('i', [111, 211, 311, 411, 511])

# iteration through for loop
for item in numericArray:
    print(item)
```

On executing the above code, it will display the following result –

```
111
211
311
411
511
```

### Using enumerate() function

The enumerate() function can be used to access elements of an array. It accepts an array and an optional starting index as parameter values and returns the array items by iterating.

### Example

In the below example, we will see how to use the enumerate() function to access array items.

```
import array as arr

# creating array
numericArray = arr.array('i', [111, 211, 311, 411, 511])

# use of enumerate() function
for loc, val in enumerate(numericArray):
    print(f"Index: {loc}, value: {val}")
```

It will produce the following output –

```
Index: 0, value: 111
Index: 1, value: 211
Index: 2, value: 311
Index: 3, value: 411
Index: 4, value: 511
```

## Accessing a range of array items in Python

In Python, to access a range of array items, you can use the slicing operation which is performed using index operator [] and colon (:).

This operation is implemented using multiple formats, which are listed below –

- Use the [:index] format to access elements from beginning to desired range.
- To access array items from end, use [:-index] format.
- Use the [index:] format to access array items from specific index number till the end.
- Use the [start index : end index] to slice the array elements within a range. You can also pass an optional argument after end index to determine the increment between each index.

### Example

The following example demonstrates the slicing operation in Python.

```
import array as arr

# creating array
numericArray = arr.array('i', [111, 211, 311, 411, 511])

# slicing operation
print (numericArray[2:])
print (numericArray[0:3])
```

On executing the above code, it will display the following result –

```
array('i', [311, 411, 511])
array('i', [111, 211, 311])
```

# 105. Python - Add Array Items

Python array is a mutable sequence which means they can be changed or modified whenever required. However, items of same data type can be added to an array. In the similar way, you can only join two arrays of the same data type.

*Python does not have built-in support for arrays, it uses **array module** to achieve the functionality like an array.*

## Adding Elements to Python Array

There are multiple ways to add elements to an array in Python –

- Using append() method
- Using insert() method
- Using extend() method

### Using append() method

To add a new element to an array, use the append() method. It accepts a single item as an argument and append it at the end of given array.

#### Syntax

Syntax of the append() method is as follows –

```
append(v)
```

Where,

- **v** – new value is added at the end of the array. The new value must be of the same type as datatype argument used while declaring array object.

#### Example

Here, we are adding element at the end of specified array using append() method.

```
import array as arr
a = arr.array('i', [1, 2, 3])
a.append(10)
print (a)
```

It will produce the following output –

```
array('i', [1, 2, 3, 10])
```

### Using insert() method

It is possible to add a new element at the specified index using the insert() method. The array module in Python defines this method. It accepts two parameters which are index and value and returns a new array after adding the specified value.

## Syntax

Syntax of this method is shown below –

```
insert(i, v)
```

Where,

- **i** – The index at which new value is to be inserted.
- **v** – The value to be inserted. Must be of the arraytype.

## Example

The following example shows how to add array elements at specific index with the help of `insert()` method.

```
import array as arr
a = arr.array('i', [1, 2, 3])
a.insert(1,20)
print (a)
```

It will produce the following output –

```
array('i', [1, 20, 2, 3])
```

## Using `extend()` method

The `extend()` method belongs to Python array module. It is used to add all elements from an iterable or array of same data type.

## Syntax

This method has the following syntax –

```
extend(x)
```

Where,

- **x** – This parameter specifies an array or iterable.

## Example

In this example, we are adding items from another array to the specified array.

```
import array as arr
a = arr.array('i', [1, 2, 3, 4, 5])
b = arr.array('i', [6,7,8,9,10])
a.extend(b)
print (a)
```

On executing the above code, it will produce the following output –

```
array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

# 106. Python - Remove Array Items

## Removing array items in Python

Python arrays are a mutable sequence which means operation like adding new elements and removing existing elements can be performed with ease. We can remove an element from an array by specifying its value or position within the given array.

The array module defines two methods namely `remove()` and `pop()`. The `remove()` method removes the element by value whereas the `pop()` method removes array item by its position.

*Python does not provide built-in support for arrays, however, we can use the array module to achieve the functionality like an array.*

### Remove First Occurrence

To remove the first occurrence of a given value from the array, use `remove()` method. This method accepts an element and removes it if the element is available in the array.

#### Syntax

```
array.remove(v)
```

Where, `v` is the value to be removed from the array.

#### Example

The below example shows the usage of `remove()` method. Here, we are removing an element from the specified array.

```
import array as arr

# creating array
numericArray = arr.array('i', [111, 211, 311, 411, 511])

# before removing array
print ("Before removing:", numericArray)

# removing array
numericArray.remove(311)

# after removing array
print ("After removing:", numericArray)
```

It will produce the following output –

```
Before removing: array('i', [111, 211, 311, 411, 511])
After removing: array('i', [111, 211, 411, 511])
```

## Remove Items from Specific Indices

To remove an array element from specific index, use the `pop()` method. This method removes an element at the specified index from the array and returns the element at its position after removal.

### Syntax

```
array.pop(i)
```

Where, `i` is the index for the element to be removed.

### Example

In this example, we will see how to use `pop()` method to remove elements from an array.

```
import array as arr

# creating array
numericArray = arr.array('i', [111, 211, 311, 411, 511])

# before removing array
print ("Before removing:", numericArray)

# removing array
numericArray.pop(3)

# after removing array
print ("After removing:", numericArray)
```

It will produce the following output –

```
Before removing: array('i', [111, 211, 311, 411, 511])
After removing: array('i', [111, 211, 311, 511])
```

# 107. Python - Loop Arrays

Loops are used to repeatedly execute a block of code. In Python, there are two types of loops named for loop and while loop. Since the array object behaves like a sequence, you can iterate through its elements with the help of loops.

The reason for looping through arrays is to perform operations such as accessing, modifying, searching, or aggregating elements of the array.

## Python for Loop with Array

The for loop is used when the number of iterations is known. If we use it with an iterable like array, the iteration continues until it has iterated over every element in the array.

### Example

The below example demonstrates how to iterate over an array using the "for" loop –

```
import array as arr  
  
newArray = arr.array('i', [56, 42, 23, 85, 45])  
  
for iterate in newArray:  
    print (iterate)
```

The above code will produce the following result –

```
56  
42  
23  
85  
45
```

## Python while Loop with Array

In while loop, the iteration continues as long as the specified condition is true. When you are using this loop with arrays, initialize a loop variable before entering the loop. This variable often represents an index for accessing elements in the array. Inside the while loop, iterate over the array elements and manually update the loop variable.

### Example

The following example shows how you can loop through an array using a while loop –

```
import array as arr  
  
# creating array  
  
a = arr.array('i', [96, 26, 56, 76, 46])  
  
# checking the length  
  
l = len(a)
```

```
# loop variable
idx = 0
# while loop
while idx < 1:
    print (a[idx])
    # incrementing the while loop
    idx+=1
```

On executing the above code, it will display the following output –

```
96
26
56
76
46
```

## Python for Loop with Array Index

We can find the length of array with built-in `len()` function. Use it to create a range object to get the series of indices and then access the array elements in a for loop.

### Example

The code below illustrates how to use for loop with array index.

```
import array as arr
a = arr.array('d', [56, 42, 23, 85, 45])
l = len(a)
for x in range(l):
    print (a[x])
```

On running the above code, it will show the below output –

```
56.0
42.0
23.0
85.0
45.0
```

# 108. Python - Copy Arrays

In Python, copying an array refers to the process of creating a new array that contains all the elements of the original array. This operation can be done using assignment operator (=) and deepcopy() method. In this chapter, we discuss how to copy an array object to another. But, before getting into the details let's briefly discuss arrays.

Python's built-in sequence types i.e. list, tuple, and string are indexed collection of items. However, unlike arrays in C/C++, Java etc. they are not homogenous, in the sense the elements in these types of collection may be of different types. Python's array module helps you to create object similar to Java like arrays.

Python arrays can be of string, integer or float type. The array class constructor is used as follows –

```
import array  
obj = array.array(typecode[, initializer])
```

Where, the typecode may be a character constant representing the data type.

## Copy Arrays Using Assignment Operator

We can assign an array to another by using the assignment operator (=). However, such assignment doesn't create a new array in the memory. Instead, it creates a new reference to the same array.

### Example

In the following example, we are using assignment operator to copy array in Python.

```
import array as arr  
a = arr.array('i', [110, 220, 330, 440, 550])  
b = a  
print("Copied array:",b)  
print (id(a), id(b))
```

It will produce the following output –

```
Copied array: array('i', [110, 220, 330, 440, 550])  
134485392383792 134485392383792
```

Check the id() of both a and b. Same value of id confirms that simple assignment doesn't create a copy. Since "a" and "b" refer to the same array object, any change in the array "a" will reflect in "b" too –

```
a[2] = 10  
print (a,b)
```

It will produce the following output –

```
array('i', [110, 220, 10, 440, 550]) array('i', [110, 220, 10, 440, 550])
```

## Copy Arrays using Deep Copy

To create another physical copy of an array, we use another module in Python library, named `copy` and use `deepcopy()` function in the module. A deep copy constructs a new compound object and then, recursively inserts copies into it of the objects found in the original.

### Example

The following example demonstrates how to copy array in Python –

```
import array as arr
import copy
a = arr.array('i', [110, 220, 330, 440, 550])
b = copy.deepcopy(a)
print("Copied array:",b)
```

On executing, it will produce the following output –

```
Copied array: array('i', [110, 220, 330, 440, 550])
```

Now check the `id()` of both "a" and "b". You will find the ids are different.

```
print (id(a), id(b))
```

It will produce the following output –

```
2771967069936 2771967068976
```

This proves that a new object "b" is created which is an actual copy of "a". If we change an element in "a", it is not reflected in "b".

```
a[2]=10
print (a,b)
```

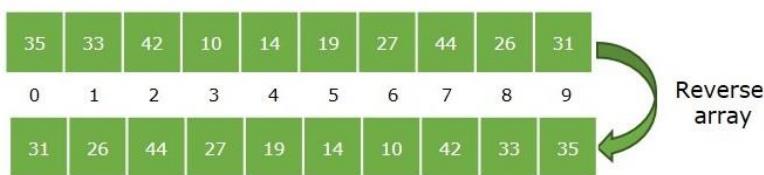
It will produce the following output –

```
array('i', [110, 220, 10, 440, 550]) array('i', [110, 220, 330, 440, 550])
```

# 109. Python - Reverse Arrays

Reversing an array is the operation of rearranging the array elements in the opposite order. There are various methods and approaches to reverse an array in Python including `reverse()` and `reversed()` methods.

In Python, array is not one of the built-in data types. However, Python's standard library has `array` module which helps us to create a homogenous collection of string, integer or float types.



## Ways to Reverse an Array in Python

To reverse an array, use the following approaches –

- Using slicing operation
- Using `reverse()` method
- Using `reversed()` method
- Using for loop

### Using slicing operation

Slicing operation is the process of extracting a part of array within the specified indices. In Python, if we use the slice operation in the form `[::-1]`, it will display a new array by reversing the original one.

In this process, the interpreter starts from the end and stepping backwards by 1 until it reaches the beginning of the array. As a result, we get a reverse copy of original array.

### Example

The below example demonstrates how to use the slicing operation to reverse an array in Python.

```
import array as arr

# creating array
numericArray = arr.array('i', [88, 99, 77, 55, 66])

print("Original array:", numericArray)
revArray = numericArray[::-1]
print("Reversed array:", revArray)
```

When you run the code, it will produce the following output –

```
Original array: array('i', [88, 99, 77, 55, 66])
Reversed array: array('i', [66, 55, 77, 99, 88])
```

## Reverse an Array Using reverse() Method

We can also reverse the sequence of numbers in an array using the reverse() method of list class. Here, list is a built-in type in Python.

Since reverse() is a method of list class, we cannot directly use it to reverse an array created through the Python array module. We have to first transfer the contents of an array to a list with tolist() method of array class, then we call the reverse() method and at the end, when we convert the list back to an array, we get the array with reversed order.

### Example

Here, we will see the use of reverse() method in reversing an array in Python.

```
import array as arr

# creating an array
numericArray = arr.array('i', [10,5,15,4,6,20,9])
print("Array before reversing:", numericArray)

# converting the array into list
newArray = numericArray.tolist()

# reversing the list
newArray.reverse()

# creating a new array from reversed list
revArray = arr.array('i', newArray)
print ("Array after reversing:",revArray)
```

It will produce the following output –

```
Array before reversing: array('i', [10, 5, 15, 4, 6, 20, 9])
Array after reversing: array('i', [9, 20, 6, 4, 15, 5, 10])
```

## Reverse an Array Using reversed() Method

The reversed() method is another way to reverse elements of an array. It accepts an array as a parameter value and returns an iterator object that displays array elements in reverse order.

### Example

In this example, we are using the reversed() method to reverse an array in Python.

```

import array as arr

# creating an array
numericArray = arr.array('i', [12, 10, 14, 16, 20, 18])
print("Array before reversing:", numericArray)

# reversing the array
newArray = list(reversed(numericArray))

# creating a new array from reversed list
revArray = arr.array('i', newArray)
print ("Array after reversing:", revArray)

```

On executing the above code, it will display the following output –

```

Array before reversing: array('i', [12, 10, 14, 16, 20, 18])
Array after reversing: array('i', [18, 20, 16, 14, 10, 12])

```

## Using for Loop

To reverse an array using a for loop, we first traverse the elements of the original array in reverse order and then append each element to a new array.

### Example

The following example shows how to reverse an array in Python using for loop.

```

import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i')
for i in range(len(a)-1, -1, -1):
    b.append(a[i])
print(a)
print(b)

```

It will produce the following output –

```

array('i', [10, 5, 15, 4, 6, 20, 9])
array('i', [9, 20, 6, 4, 15, 5, 10])

```

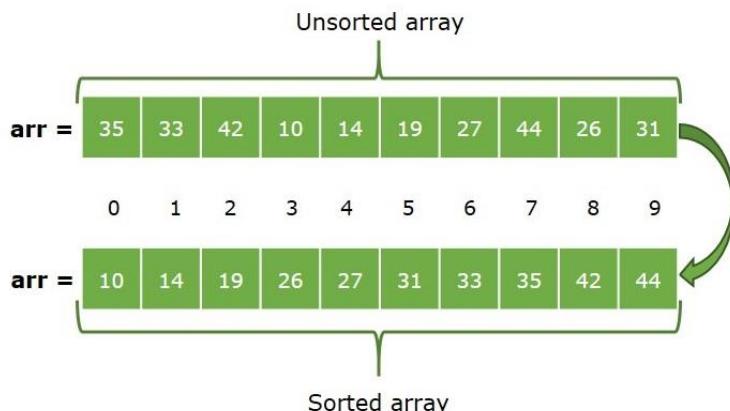
# 110. Python - Sort Arrays

Python's array module defines the array class. An object of array class is similar to the array as present in Java or C/C++. Unlike the built-in Python sequences, array is a homogenous collection of either strings, or integers, or float objects.

The array class doesn't have any function/method to give a sorted arrangement of its elements. However, we can achieve it with one of the following approaches –

- Using a sorting algorithm
- Using the sort() method from List
- Using the built-in sorted() function

Let's discuss each of these methods in detail.



## Sort Arrays Using a Sorting Algorithm

We implement the classical bubble sort algorithm to obtain the sorted array. To do it, we use two nested loops and swap the elements for rearranging in sorted order.

### Example

Run the following code using a Python code editor –

```
import array as arr  
a = arr.array('i', [10,5,15,4,6,20,9])  
for i in range(0, len(a)):  
    for j in range(i+1, len(a)):  
        if(a[i] > a[j]):  
            temp = a[i];  
            a[i] = a[j];  
            a[j] = temp;  
print (a)
```

It will produce the following output –

```
array('i', [4, 5, 6, 9, 10, 15, 20])
```

## Sort Arrays Using sort() Method of List

Even though array module doesn't have a sort() method, Python's built-in List class does have a sort method. We shall use it in the next example.

First, declare an array and obtain a list object from it, using tolist() method. Then, use the sort() method to get a sorted list. Lastly, create another array using the sorted list which will display a sorted array.

### Example

The following code shows how to get sorted array using the sort() method.

```
import array as arr

# creating array
orgnlArray = arr.array('i', [10,5,15,4,6,20,9])
print("Original array:", orgnlArray)

# converting to list
sortedList = orgnlArray.tolist()

# sorting the list
sortedList.sort()

# creating array from sorted list
sortedArray = arr.array('i', sortedList)
print("Array after sorting:",sortedArray)
```

The above code will display the following output –

```
Original array: array('i', [10, 5, 15, 4, 6, 20, 9])
Array after sorting: array('i', [4, 5, 6, 9, 10, 15, 20])
```

## Sort Arrays Using sorted() Method

The third technique to sort an array is with the sorted() function, which is a built-in function.

The syntax of sorted() function is as follows –

```
sorted(iterable, reverse=False)
```

The function returns a new list containing all items from the iterable in ascending order. Set reverse parameter to True to get a descending order of items.

The sorted() function can be used along with any iterable. Python array is an iterable as it is an indexed collection. Hence, an array can be used as a parameter to sorted() function.

### Example

In this example, we will see the use of sorted() method in sorting an array.

```
import array as arr
```

```
a = arr.array('i', [4, 5, 6, 9, 10, 15, 20])  
sorted(a)  
print(a)
```

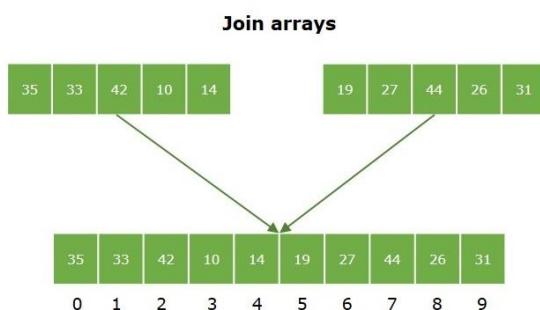
It will produce the following output –

```
array('i', [4, 5, 6, 9, 10, 15, 20])
```

# 111. Python - Join Arrays

The process of joining two arrays is termed as Merging or concatenating. Python provides multiple ways to merge two arrays such as `append()` and `extend()` methods. But, before merging two arrays always ensure that both arrays are of the same data type otherwise program will throw error.

In Python, array is a homogenous collection of Python's built in data types such as strings, integer or float objects. However, array itself is not a built-in type, instead we need to use the Python's built-in array module.



## Join two Arrays in Python

To join arrays in Python, use the following approaches –

- Using `append()` method
- Using `+` operator
- Using `extend()` method

### Using `append()` Method

To join two arrays, we can append each item from one array to other using `append()` method. To perform this operation, run a `for` loop on the original array, fetch each element and append it to a new array.

#### Example: Join Two Arrays by Appending Elements

Here, we use the `append()` method to join two arrays.

```
import array as arr

# creating two arrays
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])

# merging both arrays
for i in range(len(b)):
    a.append(b[i])

print (a)
```

It will produce the following output –

```
array('i', [10, 5, 15, 4, 6, 20, 9, 2, 7, 8, 11, 3, 10])
```

## Using + operator

We can also use + operator to concatenate or merge two arrays. In this approach, we first convert arrays to list objects, then concatenate the lists using the + operator and convert back to get merged array.

### Example: Join Two Arrays by Converting to List Objects

In this example, we will see how to join two arrays using + operator.

```
import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
b = arr.array('i', [2,7,8,11,3,10])
x = a.tolist()
y = b.tolist()
z = x+y
a = arr.array('i', z)
print (a)
```

The above code will display the following output –

```
array('i', [10, 5, 15, 4, 6, 20, 9, 2, 7, 8, 11, 3, 10])
```

## Using extend() Method

Another approach to concatenate arrays is the use of extend() method from the List class. Similar to above approach, we first convert the array to a list and then call the extend() method to merge the two lists.

### Example: Join Two Arrays using extend() Method

In the following example, we will use the extend() method to concatenate two arrays in Python.

```
import array as arr
a = arr.array('i', [88, 99, 77, 66, 44, 22])
b = arr.array('i', [12, 17, 18, 11, 13, 10])
a.extend(b)
print (a)
```

It will produce the following output –

```
array('i', [88, 99, 77, 66, 44, 22, 12, 17, 18, 11, 13, 10])
```

# 112. Python - Array Methods

The array module in Python offers an efficient object type for representing arrays of basic values like characters, integers, and floating point numbers. Arrays are similar to lists but they stores a collection of homogeneous data elements in order. At the time of creating array's, type is specified using a single character type code.

Array methods provides various operations on array objects, including appending, extending, and manipulating elements. These methods are used for efficient handling of homogeneous collections of basic data types, making them suitable for tasks requiring compact data storage, such as numerical computations.

## Python Array Class

The array class defines several methods, including adding and removing elements, obtaining information about the array, manipulating array elements, and converting arrays to and from other data types. Below are categorized methods based on their functionality. Let's explore and understand the functionality of each method.

Arrays are created using the `array.array(typecode[, initializer])` class, where `typecode` is a single character that defines the type of elements in the array, and `initializer` is an optional value used to initialize the array.

### Adding and Removing Elements

Below methods are used for appending, extending, inserting, and removing elements from arrays –

| Sr.No. | Methods with Description                                                                                                                |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><u>append(x)</u></a><br>Appends a new item with value x to the end of the array.                                            |
| 2      | <a href="#"><u>extend(iterable)</u></a><br>Appends items from iterable to the end of the array.                                         |
| 3      | <a href="#"><u>insert(i, x)</u></a><br>Inserts a new item with value x before position i.                                               |
| 4      | <a href="#"><u>pop([i])</u></a><br>Removes and returns the item with index i. If i is not specified, removes and returns the last item. |
| 5      | <a href="#"><u>remove(x)</u></a><br>Removes the first occurrence of x from the array.                                                   |

### Information and Utility Methods

These methods are used for obtaining information about arrays and to perform utility operations –

| Sr.No. | Methods with Description                                                                                                                                                                |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><b>buffer_info()</b></a><br>Returns a tuple (address, length) giving the current memory address and the length in elements of the buffer used to hold the array's contents. |
| 2      | <a href="#"><b>count(x)</b></a><br>Returns the number of occurrences of x in the array.                                                                                                 |
| 3      | <a href="#"><b>index(x[, start[, stop]])</b></a><br>Returns the smallest index where x is found in the array. Optional start and stop arguments can specify a sub-range to search.      |

## Manipulating Array Elements

Following methods are used for manipulating array elements, such as reversing the array or byteswapping values.

| Sr.No. | Methods with Description                                                                                                                                       |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><b>reverse()</b></a><br>Reverses the order of the items in the array.                                                                              |
| 2      | <a href="#"><b>byteswap()</b></a><br>"Byteswaps" all items of the array, useful for reading data from a file written on a machine with a different byte order. |

## Conversion Methods

These methods are used to convert arrays to and from bytes, files, lists, and Unicode strings.

| Sr.No. | Methods with Description                                                                                                                      |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><b>frombytes(buffer)</b></a><br>Appends items from the bytes-like object, interpreting its content as an array of machine values. |
| 2      | <a href="#"><b>tobytes()</b></a><br>Converts the array to a bytes representation.                                                             |
| 3      | <a href="#"><b>fromfile(f, n)</b></a><br>Reads n items from the file object f and appends them to the array.                                  |
| 4      | <a href="#"><b>tofile(f)</b></a><br>Writes all items to the file object f.                                                                    |
| 5      | <a href="#"><b>fromlist(list)</b></a><br>Appends items from the list to the array.                                                            |
| 6      | <a href="#"><b>tolist()</b></a><br>Converts the array to a list with the same items.                                                          |
| 7      | <a href="#"><b>fromunicode(s)</b></a><br>Extends the array with data from the given Unicode string. The array must have type code 'u'.        |

|   |                                                                            |
|---|----------------------------------------------------------------------------|
| 8 | <a href="#"><u>tounicode()</u></a>                                         |
|   | Converts the array to a Unicode string. The array must have type code 'u'. |

# 113. Python - Array Exercises

## Example 1

Python program to find the largest number in an array –

```
import array as arr  
a = arr.array('i', [10,5,15,4,6,20,9])  
print (a)  
largest = a[0]  
for i in range(1, len(a)):  
    if a[i]>largest:  
        largest=a[i]  
print ("Largest number:", largest)
```

It will produce the following output –

```
array('i', [10, 5, 15, 4, 6, 20, 9])  
Largest number: 20
```

## Example 2

Python program to store all even numbers from an array in another array –

```
import array as arr  
a = arr.array('i', [10,5,15,4,6,20,9])  
print (a)  
b = arr.array('i')  
for i in range(len(a)):  
    if a[i]%2 == 0:  
        b.append(a[i])  
print ("Even numbers:", b)
```

It will produce the following output –

```
array('i', [10, 5, 15, 4, 6, 20, 9])  
Even numbers: array('i', [10, 4, 6, 20])
```

## Example 3

Python program to find the average of all numbers in a Python array –

```

import array as arr
a = arr.array('i', [10,5,15,4,6,20,9])
print (a)
s = 0
for i in range(len(a)):
    s+=a[i]
avg = s/len(a)
print ("Average:", avg)

# Using sum() function
avg = sum(a)/len(a)
print ("Average:", avg)

```

It will produce the following output –

```

array('i', [10, 5, 15, 4, 6, 20, 9])
Average: 9.857142857142858
Average: 9.857142857142858

```

## Exercise Programs

Python program find difference between each number in the array and the average of all numbers

- Python program to convert a string in an array
- Python program to split an array in two and store even numbers in one array and odd numbers in the other.
- Python program to perform insertion sort on an array.
- Python program to store the Unicode value of each character in the given array.

# Python File Handling

# 114. Python - File Handling

## File Handling in Python

File handling in Python involves interacting with files on your computer to read data from them or write data to them. Python provides several built-in functions and methods for creating, opening, reading, writing, and closing files. This tutorial covers the basics of file handling in Python with examples.

### Opening a File in Python

To perform any file operation, the first step is to open the file. Python's built-in `open()` function is used to open files in various modes, such as reading, writing, and appending. The syntax for opening a file in Python is –

```
file = open("filename", "mode")
```

Where, `filename` is the name of the file to open and `mode` is the mode in which the file is opened (e.g., '`r`' for reading, '`w`' for writing, '`a`' for appending).

### File Opening Modes

Following are the file opening modes –

| Sr.No. | Modes & Description                                                                                                                             |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>r</b><br>Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.                   |
| 2      | <b>rb</b><br>Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| 3      | <b>r+</b><br>Opens a file for both reading and writing. The file pointer placed at the beginning of the file.                                   |
| 4      | <b>rb+</b><br>Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.                 |
| 5      | <b>w</b><br>Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.  |

|    |                                                                                                                                                                                                                                        |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 6  | <b>b</b><br>Opens the file in binary mode                                                                                                                                                                                              |
| 7  | <b>t</b><br>Opens the file in text mode (default)                                                                                                                                                                                      |
| 8  | <b>+</b><br>open file for updating (reading and writing)                                                                                                                                                                               |
| 9  | <b>wb</b><br>Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.                                                                       |
| 10 | <b>w+</b><br>Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.                                                       |
| 11 | <b>wb+</b><br>Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.                                     |
| 12 | <b>a</b><br>Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.                         |
| 13 | <b>ab</b><br>Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.       |
| 14 | <b>a+</b><br>Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| 15 | <b>ab+</b>                                                                                                                                                                                                                             |

Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

16

**x**

open for exclusive creation, failing if the file already exists

Once a file is opened and you have one file object, you can get various information related to that file.

### Example 1

In the following example, we are opening a file in different modes –

```
# Opening a file in read mode
file = open("example.txt", "r")

# Opening a file in write mode
file = open("example.txt", "w")

# Opening a file in append mode
file = open("example.txt", "a")

# Opening a file in binary read mode
file = open("example.txt", "rb")
```

### Example 2

Here, we are opening a file named "foo.txt" in binary write mode ("wb"), printing its name, whether it's closed, and its opening mode, and then closing the file –

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not: ", fo.closed)
print ("Opening mode: ", fo.mode)
fo.close()
```

It will produce the following output –

```
Name of the file: foo.txt
Closed or not: False
Opening mode: wb
```

## Reading a File in Python

Reading a file in Python involves opening the file in a mode that allows for reading, and then using various methods to extract the data from the file. Python provides several methods to read data from a file –

- **read()** – Reads the entire file.
- **readline()** – Reads one line at a time.
- **readlines** – Reads all lines into a list.

*To read a file, you need to open it in read mode. The default mode for the open() function is read mode ('r'), but it's good practice to specify it explicitly.*

### Example: Using read() method

In the following example, we are using the read() method to read the whole file into a single string –

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Following is the output obtained –

```
Hello!!!
Welcome to TutorialsPoint!!!
```

### Example: Using readline() method

In here, we are using the readline() method to read one line at a time, making it memory efficient for reading large files line by line –

```
with open("example.txt", "r") as file:
    line = file.readline()
    while line:
        print(line, end='')
        line = file.readline()
```

Output of the above code is as shown below –

```
Hello!!!
Welcome to TutorialsPoint!!!
```

### Example: Using readlines() method

Now, we are using the readlines() method to read the entire file and split it into a list where each element is a line –

```
with open("example.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
```

```
print(line, end='')
```

We get the output as follows –

```
Hello!!!
Welcome to TutorialsPoint!!!
```

## Writing to a File in Python

Writing to a file in Python involves opening the file in a mode that allows writing, and then using various methods to add content to the file.

To write data to a file, use the `write()` or `writelines()` methods. When opening a file in write mode ('w'), the file's existing content is erased.

### **Example: Using the `write()` method**

In this example, we are using the `write()` method to write the string passed to it to the file. If the file is opened in 'w' mode, it will overwrite any existing content. If the file is opened in 'a' mode, it will append the string to the end of the file –

```
with open("foo.txt", "w") as file:
    file.write("Hello, World!")
    print ("Content added Successfully!!")
```

Output of the above code is as follows –

```
Content added Successfully!!
```

### **Example: Using the `writelines()` method**

In here, we are using the `writelines()` method to take a list of strings and writes each string to the file. It is useful for writing multiple lines at once –

```
lines = ["First line\n", "Second line\n", "Third line\n"]
with open("example.txt", "w") as file:
    file.writelines(lines)
    print ("Content added Successfully!!")
```

The result obtained is as follows –

```
Content added Successfully!!
```

## Closing a File in Python

We can close a file in Python using the `close()` method. Closing a file is an essential step in file handling to ensure that all resources used by the file are properly released. It is important to close files after operations are completed to prevent data loss and free up system resources.

### **Example**

In this example, we open the file for writing, write data to the file, and then close the file using the `close()` method –

```
file = open("example.txt", "w")
file.write("This is an example.")
file.close()
print ("File closed successfully!!")
```

The output produced is as shown below –

```
File closed successfully!!
```

### Using "with" Statement for Automatic File Closing

The with statement is a best practice in Python for file operations because it ensures that the file is automatically closed when the block of code is exited, even if an exception occurs.

#### Example

In this example, the file is automatically closed at the end of the with block, so there is no need to call close() method explicitly –

```
with open("example.txt", "w") as file:
    file.write("This is an example using the with statement.")
    print ("File closed successfully!!")
```

Following is the output of the above code –

```
File closed successfully!!
```

### Handling Exceptions when closing a File

When performing file operations, it is important to handle potential exceptions to ensure your program can manage errors gracefully.

In Python, we use a try-finally block to handle exceptions when closing a file. The "finally" block ensures that the file is closed regardless of whether an error occurs in the try block –

```
try:
    file = open("example.txt", "w")
    file.write("This is an example with exception handling.")
finally:
    file.close()
    print ("File closed successfully!!")
```

After executing the above code, we get the following output –

```
File closed successfully!!
```

# 115. Python - Write to File

Writing to a file involves opening the file in a specific mode, writing data to it, and then closing the file to ensure that all data is saved and resources are released. Python provides a built-in function `open()` to handle file operations and various methods for writing data.

## Opening a File for Writing

Opening a file for writing is the first step in performing write operations in Python. The `open()` function is used to open files in different modes, each suited for specific use cases.

### The `open()` Function

The `open()` function in Python is used to open a file. It requires at least one argument, the name of the file, and can take an optional second argument that specifies the mode in which the file should be opened.

### File Modes for Writing

Following are the main modes you can use to open a file for writing –

- **w (Write mode)** – Opens the file for writing. If the file exists, it truncates (empties) the file before writing. If the file does not exist, it creates a new file.
- **a (Append Mode)** – Data is written at the end of the file. If the file does not exist, it creates a new file.
- **x (Exclusive Creation Mode)** – Opens the file for exclusive creation. If the file already exists, the operation fails.
- **b (Binary Mode)** – When used with other modes, opens the file in binary mode.
- **+(Update Mode)** – Opens the file for updating (reading and writing).

### Example: Opening a File in Write Mode

This mode is used when you want to write data to a file, starting from scratch each time the file is opened –

```
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()
print ("File opened successfully!!")
```

Following is the output obtained –

```
File opened successfully!!
```

### Example: Opening a File in Append Mode

This mode is used when you want to add data to the end of the file without altering its existing contents –

```
file = open("example.txt", "a")
file.write("Appending this line.\n")
```

```
file.close()
print ("File opened successfully!!")
```

This will produce the following result –

```
File opened successfully!!
```

## Writing to a File Using write() Method

The write() method is used to write a single string to a file. This makes it suitable for various text-based file operations.

The write() method takes a single argument: the string that you want to write to the file. It writes the exact content of the string to the file without adding any additional characters, such as newlines.

### Example

In the following example, we are opening the file "example.txt" in write mode. We then use the write() method to write a string to the file –

```
# Open a file in write mode
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a new line.\n")
print ("File opened successfully!!")
```

Following is the output of the above code –

```
File opened successfully!!
```

## Writing to a File Using writelines() Method

The writelines() method is used to write a list of strings to a file. Each string in the list is written to the file sequentially without adding any newline characters automatically.

### Example

In this example, we are creating a list of strings, lines, with each string ending in a newline character. We then open a file "example.txt" in write mode and use the writelines() method to write all the strings in the list to the file in one operation –

```
# List of lines to write to the file
lines = ["First line\n", "Second line\n", "Third line\n"]
# Open a file in write mode
with open("example.txt", "w") as file:
    file.writelines(lines)
print ("File opened successfully!!")
```

The output obtained is as shown below –

```
File opened successfully!!
```

## Writing to a New File

Writing to a new file in Python involves creating a new file (or overwriting an existing one) and writing the desired content to it. Here, we will explain the steps involved in writing to a new file –

- **Open the File** – Use the `open()` function to create or open a file in write mode ("w" or "wb" for binary files).
- **Write Data** – Use the `write()` or `writelines()` method to write data to the file.
- **Close the File** – Ensure the file is properly closed after writing, generally using the "with" statement for automatic handling.

### Example

In the example below, we create a "foo.txt" file and write given content in that file and finally close that file –

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")
# Close opened file
fo.close()
```

If you open this file with any text editor application such as Notepad, it will have the following content –

```
Python is a great language.
Yeah its great!!
```

## Writing to a New File in Binary Mode

By default, read/write operations on a file object are performed on text string data. If we need to handle files of different types, such as media files (mp3), executables (exe), or pictures (jpg), we must open the file in binary mode by adding the 'b' prefix to the read/write mode.

### Writing Binary Data to a File

To write binary data to a file, open the file in binary write mode ('wb'). The following example demonstrates this –

```
# Open a file in binary write mode
with open('test.bin', 'wb') as f:
    # Binary data
    data = b"Hello World"
    f.write(data)
```

## Converting Text Strings to Bytes

Conversion of a text string to bytes can be done using the encode() function. This is useful when you need to write text data as binary data –

```
# Open a file in binary write mode
with open('test.bin', 'wb') as f:
    # Convert text string to bytes
    data = "Hello World".encode('utf-8')
    f.write(data)
```

## Writing to an Existing File

When an existing file is opened in write mode ('w'), its previous contents are erased. Opening a file with write permission treats it as a new file. To add data to an existing file without erasing its contents, you should open the file in append mode ('a').

### Example

The following example demonstrates how to open a file in append mode and add new text to it –

```
# Open a file in append mode
fo = open("foo.txt", "a")
text = "TutorialsPoint has a fabulous Python tutorial"
fo.write(text)

# Close opened file
fo.close()
```

If you open this file with any text editor application such as Notepad, it will have the following content –

```
Python is a great language.
Yeah its great!!
TutorialsPoint has a fabulous Python tutorial
```

## Writing to a File in Reading and Writing Modes

When a file is opened for writing using 'w' or 'a', it is not possible to perform write operations at any earlier byte position in the file. The 'w+' mode, however, allows both reading and writing operations without closing the file. The seek() function is used to move the read/write pointer to any desired byte position within the file.

### Using the seek() Method

The seek() method is used to set the position of the read/write pointer within the file. The syntax for the seek() method is as follows –

```
fileObject.seek(offset[, whence])
```

Where,

- **offset** – This is the position of the read/write pointer within the file.
- **whence** – This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

### Example

The following program demonstrates how to open a file in read-write mode ('w+'), write some data, seek a specific position, and then overwrite part of the file's content –

```
# Open a file in read-write mode
fo = open("foo.txt", "w+")
# Write initial data to the file
fo.write("This is a rat race")
# Move the read/write pointer to the 10th byte
fo.seek(10, 0)
# Read 3 bytes from the current position
data = fo.read(3)
# Move the read/write pointer back to the 10th byte
fo.seek(10, 0)
# Overwrite the existing content with new text
fo.write('cat')
# Close the file
fo.close()
```

If we open the file in read mode (or seek to the starting position while in 'w+' mode) and read the contents, it will show the following –

```
This is a cat race
```

# 116. Python - Read Files

Reading from a file involves opening the file, reading its contents, and then closing the file to free up system resources. Python provides several methods to read from a file, each suited for different use cases.

## Opening a File for Reading

Opening a file is the first step in reading its contents. In Python, you use the `open()` function to open a file. This function requires at least one argument, the filename, and optionally a mode that specifies the purpose of opening the file.

To open a file for reading, you use the mode '`r`'. This is the default mode, so you can omit it if you only need to read from the file.

## Reading a File using `read()` Method

The `read()` method is used to read the contents of a file in Python. It reads the entire content of the file as a single string. This method is particularly useful when you need to process the whole file at once.

### Syntax

Following is the basic syntax of the `read()` method in Python –

```
file_object.read(size)
```

Where,

- **file\_object** is the file object returned by the `open()` function.
- **size** is the number of bytes to read from the file. This parameter is optional. If omitted or set to a negative value, the method reads until the end of the file.

### Example

In the following example, we are opening the file "example.txt" in read mode. We then use the `read()` method to read the entire content of the file –

```
# Open the file in read mode
file = open('example.txt', 'r')

# Read the entire content of the file
content = file.read()

# Print the content
print(content)

# Close the file
file.close()
```

After executing the above code, we get the following output –

```
welcome to TutorialsPoint.
```

## Reading a File Using readline() Method

The readline() method is used to read one line from a file at a time. This method is useful when you need to process a file line by line, especially for large files where reading the entire content at once is not practical.

### Syntax

Following is the basic syntax of the readline() method in Python –

```
file_object.readline(size)
```

Where,

- **file\_object** is the file object returned by the open() function.
- **size** is an optional parameter specifying the maximum number of bytes to read from the line. If omitted or set to a negative value, the method reads until the end of the line.

### Example

In the example below, we are opening the file "example.txt" in read mode. We then use the readline() method to read the first line of the file –

```
# Open the file in read mode
file = open('example.txt', 'r')

# Read the first line of the file
line = file.readline()

# Print the line
print(line)

# Close the file
file.close()
```

Following is the output of the above code –

```
welcome to TutorialsPoint.
```

## Reading a File Using readlines() Method

The readlines() method reads all the lines from a file and returns them as a list of strings. Each string in the list represents a single line from the file, including the newline character at the end of each line.

This method is particularly useful when you need to process or analyse all lines of a file at once.

## Syntax

Following is the basic syntax of the `readlines()` method in Python –

```
file_object.readlines(hint)
```

Where,

- **file\_object** is the file object returned by the `open()` function.
- **hint** is an optional parameter that specifies the number of bytes to read. If specified, it reads lines up to the specified bytes, not necessarily reading the entire file.

## Example

In this example, we are opening the file "example.txt" in read mode. We then use the `readlines()` method to read all the lines from the file and return them as a list of strings –

```
# Open the file in read mode
file = open('example.txt', 'r')

# Read all lines from the file
lines = file.readlines()

# Print the lines
for line in lines:
    print(line, end='')

# Close the file
file.close()
```

Output of the above code is as shown below –

```
welcome to TutorialsPoint.
Hi Surya.
How are you?.
```

## Using "with" Statement

The "with" statement in Python is used for exception handling. When dealing with files, using the "with" statement ensures that the file is properly closed after reading, even if an exception occurs.

*When you use a with statement to open a file, the file is automatically closed at the end of the block, even if an error occurs within the block.*

## Example

Following is a simple example of using the with statement to open, read, and print the contents of a file –

```
# Using the with statement to open a file
with open('example.txt', 'r') as file:
    content = file.read()
```

```
print(content)
```

We get the output as follows –

```
welcome to TutorialsPoint.  
Hi Surya.  
How are you?.
```

## Reading a File in Binary Mode

By default, read/write operation on a file object are performed on text string data. If we want to handle files of different types, such as media files (mp3), executables (exe), or pictures (jpg), we must open the file in binary mode by adding the 'b' prefix to the read/write mode.

### Writing to a Binary File

Assuming that the test.bin file has already been written in binary mode –

```
# Open the file in binary write mode  
with open('test.bin', 'wb') as f:  
    data = b"Hello World"  
    f.write(data)
```

### Example

To read a binary file, we need to open it in 'rb' mode. The returned value of the read() method is then decoded before printing –

```
# Open the file in binary read mode  
with open('test.bin', 'rb') as f:  
    data = f.read()  
    print(data.decode(encoding='utf-8'))
```

It will produce the following output –

```
Hello World
```

## Reading Integer Data from a File

To write integer data to a binary file, the integer object should be converted to bytes using the to\_bytes() method.

### Writing an Integer to a Binary File

Following is an example on how to write an integer to a binary file –

```
# Convert the integer to bytes and write to a binary file  
n = 25  
data = n.to_bytes(8, 'big')
```

```
with open('test.bin', 'wb') as f:
    f.write(data)
```

## Reading an Integer from a Binary File

To read back the integer data from the binary file, convert the output of the `read()` function back to an integer using the `from_bytes()` method –

```
# Read the binary data from the file and convert it back to an integer
with open('test.bin', 'rb') as f:
    data = f.read()
    n = int.from_bytes(data, 'big')
    print(n)
```

## Reading Float Data from a File

For handling floating-point data in binary files, we need to use the `struct` module from Python's standard library. This module helps convert between Python values and C structs represented as Python bytes objects.

### Writing a Float to a Binary File

To write floating-point data to a binary file, we use the `struct.pack()` method to convert the float into a bytes object –

```
import struct

# Define a floating-point number
x = 23.50

# Pack the float into a binary format
data = struct.pack('f', x)

# Open the file in binary write mode and write the packed data
with open('test.bin', 'wb') as f:
    f.write(data)
```

### Reading Float Numbers from a Binary File

To read floating-point data from a binary file, we use the `struct.unpack()` method to convert the bytes object back into a float –

```
import struct

# Open the file in binary read mode
with open('test.bin', 'rb') as f:
```

```
# Read the binary data from the file
data = f.read()

# Unpack the binary data to retrieve the float
x = struct.unpack('f', data)[0]

# Print the float value
print(x)
```

## Reading and Writing to a File Using "r+" Mode

When a file is opened for reading (with 'r' or 'rb'), writing data is not possible unless the file is closed and reopened in a different mode. To perform both read and write operations simultaneously, we add the '+' character to the mode parameter. Using 'w+' or 'r+' mode enables using both write() and read() methods without closing the file.

The File object also supports the seek() function, which allows repositioning the read/write pointer to any desired byte position within the file.

### Syntax

Following is the syntax for seek() method –

```
fileObject.seek(offset[, whence])
```

### Parameters

- **offset** – This is the position of the read/write pointer within the file.
- **whence** – This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

### Example

The following program opens a file in 'r+' mode (read-write mode), seeks a certain position in the file, and reads data from that position –

```
# Open the file in read-write mode
with open("foo.txt", "r+") as fo:
    # Move the read/write pointer to the 10th byte position
    fo.seek(10, 0)

    # Read 3 bytes from the current position
    data = fo.read(3)

    # Print the read data
    print(data)
```

After executing the above code, we get the following output –

```
rat
```

## Reading and Writing to a File Simultaneously in Python

When a file is opened for writing (with 'w' or 'a'), it is not possible to read from it, and attempting to do so will throw an `UnsupportedOperation` error.

Similarly, when a file is opened for reading (with 'r' or 'rb'), writing to it is not allowed. To switch between reading and writing, you would typically need to close the file and reopen it in the desired mode.

To perform both read and write operations simultaneously, you can add the '+' character to the mode parameter. Using 'w+' or 'r+' mode enables both `write()` and `read()` methods without needing to close the file.

Additionally, the `File` object supports the `seek()` function, which allows you to reposition the read/write pointer to any desired byte position within the file.

### Example

In this example, we open the file in 'r+' mode and write data to the file. The `seek(0)` method repositions the pointer to the beginning of the file –

```
# Open the file in read-write mode
with open("foo.txt", "r+") as fo:
    # Write data to the file
    fo.write("This is a rat race")

    # Rewind the pointer to the beginning of the file
    fo.seek(0)

    # Read data from the file
    data = fo.read()
    print(data)
```

## Reading a File from Specific Offset

We can set the file's current position at the specified offset using the `seek()` method.

- If the file is opened for appending using either 'a' or 'a+', any `seek()` operations will be undone at the next write.
- If the file is opened only for writing in append mode using 'a', this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+').
- If the file is opened in text mode using 't', only offsets returned by `tell()` are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

### Example

The following example demonstrates how to use the `seek()` method to perform simultaneous read/write operations on a file. The file is opened in `w+` mode (read-write mode), some data is added, and then the file is read and modified at a specific position –

```
# Open a file in read-write mode
fo = open("foo.txt", "w+")
# Write initial data to the file
fo.write("This is a rat race")
# Seek to a specific position in the file
fo.seek(10, 0)
# Read a few bytes from the current position
data = fo.read(3)
print("Data read from position 10:", data)
# Seek back to the same position
fo.seek(10, 0)
# Overwrite the earlier contents with new text
fo.write("cat")
# Rewind to the beginning of the file
fo.seek(0, 0)
# Read the entire file content
data = fo.read()
print("Updated file content:", data)
# Close the file
fo.close()
```

Following is the output of the above code –

```
Data read from position 10: rat
Updated file content: This is a cat race
```

# 117. Python - Renaming and Deleting Files

## Renaming and Deleting Files in Python

In Python, you can rename and delete files using built-in functions from the `os` module. These operations are important when managing files within a file system. In this tutorial, we will explore how to perform these actions step-by-step.

### Renaming Files in Python

To rename a file in Python, you can use the `os.rename()` function. This function takes two arguments: the current filename and the new filename.

#### Syntax

Following is the basic syntax of the `rename()` function in Python –

```
os.rename(current_file_name, new_file_name)
```

#### Parameters

Following are the parameters accepted by this function –

- **current\_file\_name** – It is the current name of the file you want to rename.
- **new\_file\_name** – It is the new name you want to assign to the file.

#### Example

Following is an example to rename an existing file "oldfile.txt" to "newfile.txt" using the `rename()` function –

```
import os

# Current file name
current_name = "oldfile.txt"

# New file name
new_name = "newfile.txt"

# Rename the file
os.rename(current_name, new_name)

print(f"File '{current_name}' renamed to '{new_name}' successfully.")
```

Following is the output of the above code –

```
File 'oldfile.txt' renamed to 'newfile.txt' successfully.
```

## Deleting Files in Python

You can delete a file in Python using the `os.remove()` function. This function deletes a file specified by its filename.

## Syntax

Following is the basic syntax of the remove() function in Python –

```
os.remove(file_name)
```

## Parameters

This function accepts the name of the file as a parameter which needs to be deleted.

## Example

Following is an example to delete an existing file "file\_to\_delete.txt" using the remove() function –

```
import os  
  
# File to be deleted  
  
file_to_delete = "file_to_delete.txt"  
  
# Delete the file  
  
os.remove(file_to_delete)  
  
print(f"File '{file_to_delete}' deleted successfully.")
```

After executing the above code, we get the following output –

```
File 'file_to_delete.txt' deleted successfully.
```

# 118. Python - Directories

## Directories in Python

In Python, directories, commonly known as folders in operating systems, are locations on the file system used to store files and other directories. They serve as a way to group and manage files hierarchically.

Python provides several modules, primarily `os` and `os.path`, along with `shutil`, that allows you to perform various operations on directories.

These operations include creating new directories, navigating through existing directories, listing directory contents, changing the current working directory, and removing directories.

### Checking if a Directory Exists

Before performing operations on a directory, you often need to check if it exists. We can check if a directory exists or not using the `os.path.exists()` function in Python.

This function accepts a single argument, which is a string representing a path in the filesystem. This argument can be –

- **Relative path** – A path relative to the current working directory.
- **Absolute path** – A complete path starting from the root directory.

### Example

In this example, we check whether the given directory path exists using the `os.path.exists()` function –

```
import os

directory_path = "D:\\Test\\MyFolder\\"

if os.path.exists(directory_path):
    print(f"The directory '{directory_path}' exists.")
else:
    print(f"The directory '{directory_path}' does not exist.")
```

Following is the output of the above code –

```
The directory 'D:\\Test\\MyFolder\\' exists.
```

### Creating a Directory

You create a new directory in Python using the `os.makedirs()` function. This function creates intermediate directories if they do not exist.

The `os.makedirs()` function accepts a "path" you want to create as an argument. It optionally accepts a "mode" argument that specifies the permissions to set for the newly created directories. It is an integer, represented in octal format (e.g., `0o755`). If not specified, the default permissions are used based on your system's umask.

## Example

In the following example, we are creating a new directory using the `os.makedirs()` function –

```
import os
new_directory = "new_dir.txt"
try:
    os.makedirs(new_directory)
    print(f"Directory '{new_directory}' created successfully.")
except OSError as e:
    print(f"Error: Failed to create directory '{new_directory}'. {e}")
```

After executing the above code, we get the following output –

```
Directory 'new_dir.txt' created successfully.
```

## The `mkdir()` Method

You can use the `mkdir()` method of the `os` module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Following is the syntax of the `mkdir()` method in Python –

```
os.mkdir("newdir")
```

## Example

Following is an example to create a directory `test` in the current directory –

```
import os
# Create a directory "test"
os.mkdir("test")
print ("Directory created successfully")
```

The result obtained is as shown below –

```
Directory created successfully
```

## Get Current Working Directory

To retrieve the current working directory in Python, you can use the `os.getcwd()` function. This function returns a string representing the current working directory where the Python script is executing.

## Syntax

Following is the basic syntax of the `getcwd()` function in Python –

```
os.getcwd()
```

## Example

Following is an example to display the current working directory using the `getcwd()` function –

```
import os
current_directory = os.getcwd()
print(f"Current working directory: {current_directory}")
```

We get the output as follows –

```
Current working directory: /home/cg/root/667ba7570a5b7
```

## **Listing Files and Directories**

You can list the contents of a directory using the `os.listdir()` function. This function returns a list of all files and directories within the specified directory path.

### **Example**

In the example below, we are listing the contents of the specified directory path using the `listdir()` function –

```
import os
directory_path = r"D:\MyFolder\Pictures"
try:
    contents = os.listdir(directory_path)
    print(f"Contents of '{directory_path}':")
    for item in contents:
        print(item)
except OSError as e:
    print(f"Error: Failed to list contents of directory '{directory_path}'. {e}")
```

Output of the above code is as shown below –

```
Contents of 'D:\MyFolder\Pictures':
Camera Roll
desktop.ini
Saved Pictures
Screenshots
```

## **Changing the Current Working Directory**

You can change the current directory using the `chdir()` method. This method takes an argument, which is the name of the directory that you want to make the current directory.

### **Syntax**

Following is the syntax of the `chdir()` method in Python –

```
os.chdir("newdir")
```

## Example

Following is an example to change the current directory to Desktop using the chdir() method –

```
import os
new_directory = r"D:\MyFolder\Pictures"
try:
    os.chdir(new_directory)
    print(f"Current working directory changed to '{new_directory}'")
except OSError as e:
    print(f"Error: Failed to change working directory to '{new_directory}'. {e}")
```

We get the output as shown below –

```
Current working directory changed to 'D:\MyFolder\Pictures'.
```

## Removing a Directory

You can remove an empty directory in Python using the os.rmdir() method. If the directory contains files or other directories, you can use shutil.rmtree() method to delete it recursively.

### Syntax

Following is the basic syntax to delete a directory in Python –

```
os.rmdir(directory_path)
# or
shutil.rmtree(directory_path)
```

## Example

In the following example, we remove an empty directory using the os.rmdir() method –

```
import os
directory_path = r"D:\MyFolder\new_dir"
try:
    os.rmdir(directory_path)
    print(f"Directory '{directory_path}' successfully removed.")
except OSError as e:
    print(f"Error: Failed to remove directory '{directory_path}'. {e}")
```

It will produce the following output –

```
Directory 'D:\MyFolder\new_dir' successfully removed.
```

# 119. Python - File Methods

A file object is created using open() function. The file class defines the following methods with which different file IO operations can be done. The methods can be used with any file like object such as byte stream or network stream.

| Sr.No. | Methods & Description                                                                                                                                                                                                                                                                                                          |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><u>file.close()</u></a><br>Close the file. A closed file cannot be read or written any more.                                                                                                                                                                                                                       |
| 2      | <a href="#"><u>file.flush()</u></a><br>Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.                                                                                                                                                                                          |
| 3      | <a href="#"><u>file.fileno()</u></a><br>Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.                                                                                                                                                 |
| 4      | <a href="#"><u>file.isatty()</u></a><br>Returns True if the file is connected to a tty(-like) device, else False.                                                                                                                                                                                                              |
| 5      | <a href="#"><u>file.next()</u></a><br>Returns the next line from the file each time it is being called.                                                                                                                                                                                                                        |
| 6      | <a href="#"><u>file.read([size])</u></a><br>Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).                                                                                                                                                                                    |
| 7      | <a href="#"><u>file.readline([size])</u></a><br>Reads one entire line from the file. A trailing newline character is kept in the string.                                                                                                                                                                                       |
| 8      | <a href="#"><u>file.readlines([sizehint])</u></a><br>Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. |
| 9      | <a href="#"><u>file.seek(offset[, whence])</u></a><br>Sets the file's current position                                                                                                                                                                                                                                         |
| 10     | <a href="#"><u>file.tell()</u></a><br>Returns the file's current position                                                                                                                                                                                                                                                      |
| 11     | <a href="#"><u>file.truncate([size])</u></a><br>Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.                                                                                                                                                             |
| 12     | <a href="#"><u>file.write(str)</u></a><br>Writes a string to the file. There is no return value.                                                                                                                                                                                                                               |

|    |                                                                                                                                   |
|----|-----------------------------------------------------------------------------------------------------------------------------------|
| 13 | <u><a href="#">file.writelines(sequence)</a></u>                                                                                  |
|    | Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. |

# 120. Python OS File/Directory Methods

The OS module of Python provides a wide range of useful methods to manage files and directories. These are the built-in methods that help in interacting with operating systems. Most of the useful methods are listed here –

| Sr.No. | Methods & Description                                                                                                                                |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><u>os.access(path, mode)</u></a><br>Use the real uid/gid to test for access to path.                                                     |
| 2      | <a href="#"><u>os.chdir(path)</u></a><br>Change the current working directory to path                                                                |
| 3      | <a href="#"><u>os.chflags(path, flags)</u></a><br>Set the flags of path to the numeric flags.                                                        |
| 4      | <a href="#"><u>os.chmod(path, mode)</u></a><br>Change the mode of path to the numeric mode.                                                          |
| 5      | <a href="#"><u>os.chown(path, uid, gid)</u></a><br>Change the owner and group id of path to the numeric uid and gid.                                 |
| 6      | <a href="#"><u>os.chroot(path)</u></a><br>Change the root directory of the current process to path.                                                  |
| 7      | <a href="#"><u>os.close(fd)</u></a><br>Close file descriptor fd.                                                                                     |
| 8      | <a href="#"><u>os.closerange(fd_low, fd_high)</u></a><br>Close all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors. |
| 9      | <a href="#"><u>os.dup(fd)</u></a><br>Return a duplicate of file descriptor fd.                                                                       |
| 10     | <a href="#"><u>os.dup2(fd, fd2)</u></a><br>Duplicate file descriptor fd to fd2, closing the latter first if necessary.                               |
| 11     | <a href="#"><u>os.fchdir(fd)</u></a><br>Change the current working directory to the directory represented by the file descriptor fd.                 |
| 12     | <a href="#"><u>os.fchmod(fd, mode)</u></a><br>Change the mode of the file given by fd to the numeric mode.                                           |
| 13     | <a href="#"><u>os.fchown(fd, uid, gid)</u></a><br>Change the owner and group id of the file given by fd to the numeric uid and gid.                  |
| 14     | <a href="#"><u>os.fdatasync(fd)</u></a><br>Force write of file with filedescriptor fd to disk.                                                       |
| 15     | <a href="#"><u>os.fdopen(fd[, mode[, bufsize]])</u></a><br>Return an open file object connected to the file descriptor fd.                           |
| 16     | <a href="#"><u>os.fpathconf(fd, name)</u></a>                                                                                                        |

|    |                                                                                                                                                                        |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Return system configuration information relevant to an open file.<br>name specifies the configuration value to retrieve.                                               |
| 17 | <a href="#"><u>os.fstat(fd)</u></a><br>Return status for file descriptor fd, like stat().                                                                              |
| 18 | <a href="#"><u>os.fstatvfs(fd)</u></a><br>Return information about the filesystem containing the file associated with file descriptor fd, like statvfs().              |
| 19 | <a href="#"><u>os.fsync(fd)</u></a><br>Force write of file with filedescriptor fd to disk.                                                                             |
| 20 | <a href="#"><u>os.ftruncate(fd, length)</u></a><br>Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.                  |
| 21 | <a href="#"><u>os.getcwd()</u></a><br>Return a string representing the current working directory.                                                                      |
| 22 | <a href="#"><u>os.getcwd()</u></a><br>Return a Unicode object representing the current working directory.                                                              |
| 23 | <a href="#"><u>os.isatty(fd)</u></a><br>Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.                                |
| 24 | <a href="#"><u>os.lchflags(path, flags)</u></a><br>Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links.                       |
| 25 | <a href="#"><u>os.lchmod(path, mode)</u></a><br>Change the mode of path to the numeric mode.                                                                           |
| 26 | <a href="#"><u>os.lchown(path, uid, gid)</u></a><br>Change the owner and group id of path to the numeric uid and gid.<br>This function will not follow symbolic links. |
| 27 | <a href="#"><u>os.link(src, dst)</u></a><br>Create a hard link pointing to src named dst.                                                                              |
| 28 | <a href="#"><u>os.listdir(path)</u></a><br>Return a list containing the names of the entries in the directory given by path.                                           |
| 29 | <a href="#"><u>os.lseek(fd, pos, how)</u></a><br>Set the current position of file descriptor fd to position pos, modified by how.                                      |
| 30 | <a href="#"><u>os.lstat(path)</u></a><br>Like stat(), but do not follow symbolic links.                                                                                |
| 31 | <a href="#"><u>os.major(device)</u></a><br>Extract the device major number from a raw device number.                                                                   |
| 32 | <a href="#"><u>os.makedev(major, minor)</u></a>                                                                                                                        |

|    |                                                                                                                                                                                                                                 |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Compose a raw device number from the major and minor device numbers.                                                                                                                                                            |
| 33 | <a href="#"><code>os.makedirs(path[, mode])</code></a><br>Recursive directory creation function.                                                                                                                                |
| 34 | <a href="#"><code>os.minor(device)</code></a><br>Extract the device minor number from a raw device number.                                                                                                                      |
| 35 | <a href="#"><code>os.mkdir(path[, mode])</code></a><br>Create a directory named path with numeric mode mode.                                                                                                                    |
| 36 | <a href="#"><code>os.mkfifo(path[, mode])</code></a><br>Create a FIFO (a named pipe) named path with numeric mode mode.<br>The default mode is 0666 (octal).                                                                    |
| 37 | <a href="#"><code>os.mknod(filename[, mode=0600, device])</code></a><br>Create a filesystem node (file, device special file or named pipe) named filename.                                                                      |
| 38 | <a href="#"><code>os.open(file, flags[, mode])</code></a><br>Open the file file and set various flags according to flags and possibly its mode according to mode.                                                               |
| 39 | <a href="#"><code>os.openpty()</code></a><br>Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively.                                                          |
| 40 | <a href="#"><code>os.pathconf(path, name)</code></a><br>Return system configuration information relevant to a named file.                                                                                                       |
| 41 | <a href="#"><code>os.pipe()</code></a><br>Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.                                                                                 |
| 42 | <a href="#"><code>os.popen(command[, mode[, bufsize]])</code></a><br>Open a pipe to or from command.                                                                                                                            |
| 43 | <a href="#"><code>os.read(fd, n)</code></a><br>Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned. |
| 44 | <a href="#"><code>os.readlink(path)</code></a><br>Return a string representing the path to which the symbolic link points.                                                                                                      |
| 45 | <a href="#"><code>os.remove(path)</code></a><br>Remove the file path.                                                                                                                                                           |
| 46 | <a href="#"><code>os.removedirs(path)</code></a><br>Remove directories recursively.                                                                                                                                             |
| 47 | <a href="#"><code>os.rename(src, dst)</code></a><br>Rename the file or directory src to dst.                                                                                                                                    |

|    |                                                                                                                                                                                                                |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 48 | <a href="#"><code>os.renames(old, new)</code></a><br>Recursive directory or file renaming function.                                                                                                            |
| 49 | <a href="#"><code>os.rmdir(path)</code></a><br>Remove the directory path                                                                                                                                       |
| 50 | <a href="#"><code>os.stat(path)</code></a><br>Perform a stat system call on the given path.                                                                                                                    |
| 51 | <a href="#"><code>os.stat_float_times([newvalue])</code></a><br>Determine whether stat_result represents time stamps as float objects.                                                                         |
| 52 | <a href="#"><code>os.statvfs(path)</code></a><br>Perform a statvfs system call on the given path.                                                                                                              |
| 53 | <a href="#"><code>os.symlink(src, dst)</code></a><br>Create a symbolic link pointing to src named dst.                                                                                                         |
| 54 | <a href="#"><code>os.tcgetpgrp(fd)</code></a><br>Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()).                                            |
| 55 | <a href="#"><code>os.tcsetpgrp(fd, pg)</code></a><br>Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg.                                     |
| 56 | <a href="#"><code>os.tempnam([dir[, prefix]])</code></a><br>Return a unique path name that is reasonable for creating a temporary file.                                                                        |
| 57 | <a href="#"><code>os.tmpfile()</code></a><br>Return a new file object opened in update mode (w+b).                                                                                                             |
| 58 | <a href="#"><code>os.tmpnam()</code></a><br>Return a unique path name that is reasonable for creating a temporary file.                                                                                        |
| 59 | <a href="#"><code>os.ttyname(fd)</code></a><br>Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised. |
| 60 | <a href="#"><code>os.unlink(path)</code></a><br>Remove the file path.                                                                                                                                          |
| 61 | <a href="#"><code>os.utime(path, times)</code></a><br>Set the access and modified times of the file specified by path.                                                                                         |
| 62 | <a href="#"><code>os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])</code></a><br>Generate the file names in a directory tree by walking the tree either top-down or bottom-up.                |
| 63 | <a href="#"><code>os.write(fd, str)</code></a><br>Write the string str to file descriptor fd. Return the number of bytes actually written.                                                                     |

# 121. Python OS.Path Methods

The os.path is another Python module, which also provides a big range of useful methods to manipulate files and directories. Most of the useful methods are listed here –

| Sr.No. | Methods with Description                                                                                                                                                                                     |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#"><u>os.path.abspath(path)</u></a><br>Returns a normalized absolutized version of the pathname path.                                                                                               |
| 2      | <a href="#"><u>os.path.basename(path)</u></a><br>Returns the base name of pathname path.                                                                                                                     |
| 3      | <a href="#"><u>os.path.commonprefix(list)</u></a><br>Returns the longest path prefix (taken character-by-character) that is a prefix of all paths in list.                                                   |
| 4      | <a href="#"><u>os.path.dirname(path)</u></a><br>Returns the directory name of pathname path.                                                                                                                 |
| 5      | <a href="#"><u>os.path.exists(path)</u></a><br>Returns True if path refers to an existing path. Returns False for broken symbolic links.                                                                     |
| 6      | <a href="#"><u>os.path.lexists(path)</u></a><br>Returns True if path refers to an existing path. Returns True for broken symbolic links.                                                                     |
| 7      | <a href="#"><u>os.path.expanduser(path)</u></a><br>On Unix and Windows, returns the argument with an initial component of ~ or ~user replaced by that user's home directory.                                 |
| 8      | <a href="#"><u>os.path.expandvars(path)</u></a><br>Returns the argument with environment variables expanded.                                                                                                 |
| 9      | <a href="#"><u>os.path.getatime(path)</u></a><br>Returns the time of last access of path.                                                                                                                    |
| 10     | <a href="#"><u>os.path.getmtime(path)</u></a><br>Returns the time of last modification of path.                                                                                                              |
| 11     | <a href="#"><u>os.path.getctime(path)</u></a><br>Returns the system's ctime, which on some systems (like Unix) is the time of the last change, and, on others (like Windows), is the creation time for path. |
| 12     | <a href="#"><u>os.path.getsize(path)</u></a><br>Returns the size, in bytes, of path.                                                                                                                         |

|    |                                                                                                                                         |
|----|-----------------------------------------------------------------------------------------------------------------------------------------|
|    | <a href="#"><code>os.path.isabs(path)</code></a>                                                                                        |
| 13 | Returns True if path is an absolute pathname.                                                                                           |
| 14 | <a href="#"><code>os.path.isfile(path)</code></a>                                                                                       |
| 14 | Returns True if path is an existing regular file.                                                                                       |
| 15 | <a href="#"><code>os.path.isdir(path)</code></a>                                                                                        |
| 15 | Returns True if path is an existing directory.                                                                                          |
| 16 | <a href="#"><code>os.path.islink(path)</code></a>                                                                                       |
| 16 | Returns True if path refers to a directory entry that is a symbolic link.                                                               |
| 17 | <a href="#"><code>os.path.ismount(path)</code></a>                                                                                      |
| 17 | Returns True if pathname path is a mount point: a point in a file system where a different file system has been mounted.                |
| 18 | <a href="#"><code>os.path.join(path1[, path2[, ...]])</code></a>                                                                        |
| 18 | Joins one or more path components intelligently.                                                                                        |
| 19 | <a href="#"><code>os.path.normcase(path)</code></a>                                                                                     |
| 19 | Normalizes the case of a pathname.                                                                                                      |
| 20 | <a href="#"><code>os.path.normpath(path)</code></a>                                                                                     |
| 20 | Normalizes a pathname.                                                                                                                  |
| 21 | <a href="#"><code>os.path.realpath(path)</code></a>                                                                                     |
| 21 | Returns the canonical path of the specified filename, eliminating any symbolic links encountered in the path                            |
| 22 | <a href="#"><code>os.path.relpath(path[, start])</code></a>                                                                             |
| 22 | Returns a relative filepath to path either from the current directory or from an optional start point.                                  |
| 23 | <a href="#"><code>os.path.samefile(path1, path2)</code></a>                                                                             |
| 23 | Returns True if both pathname arguments refer to the same file or directory                                                             |
| 24 | <a href="#"><code>os.path.sameopenfile(fp1, fp2)</code></a>                                                                             |
| 24 | Returns True if the file descriptors fp1 and fp2 refer to the same file.                                                                |
| 25 | <a href="#"><code>os.path.samestat(stat1, stat2)</code></a>                                                                             |
| 25 | Returns True if the stat tuples stat1 and stat2 refer to the same file.                                                                 |
| 26 | <a href="#"><code>os.path.split(path)</code></a>                                                                                        |
| 26 | Splits the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. |
| 27 | <a href="#"><code>os.path.splitdrive(path)</code></a>                                                                                   |

|    |                                                                                                                                                                                                                               |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Splits the pathname path into a pair (drive, tail) where drive is either a drive specification or the empty string.                                                                                                           |
| 28 | <a href="#"><u>os.path.splitext(path)</u></a><br>Splits the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period.                     |
| 29 | <a href="#"><u>os.path.walk(path, visit, arg)</u></a><br>Calls the function visit with arguments (arg, dirname, names) for each directory in the directory tree rooted at path (including path itself, if it is a directory). |

# Object Oriented Programming

## 122. Python - OOP Concepts

OOP stands for Object-oriented programming paradigm. It is defined as a programming model that uses the concept of objects which refers to real-world entities with state and behavior. This chapter helps you become an expert in using object-oriented programming support in Python language.

Python is a programming language that supports object-oriented programming. This makes it simple to create and use classes and objects. If you do not have any prior experience with object-oriented programming, you are at the right place. Let's start by discussing a small introduction of Object-Oriented Programming (OOP) to help you.

### Procedural Oriented Approach

Early programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.

A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.

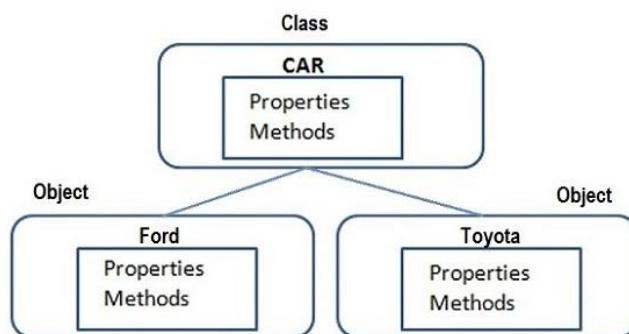
Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.

Prominent problems related to procedural approach are as follows –

- Its top-down approach makes the program difficult to maintain.
- It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.
- It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.
- Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process.

### Python - OOP Concepts

In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.



## Attributes

- Name, class, subjects, marks, etc., of student
- Name, designation, department, salary, etc., of employee
- Invoice number, customer, product code and name, price and quantity, etc., in an invoice
- Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

## Behavior

Processing attributes associated with an object.

- Compute percentage of student's marks
- Calculate incentives payable to employee
- Apply GST to invoice value
- Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

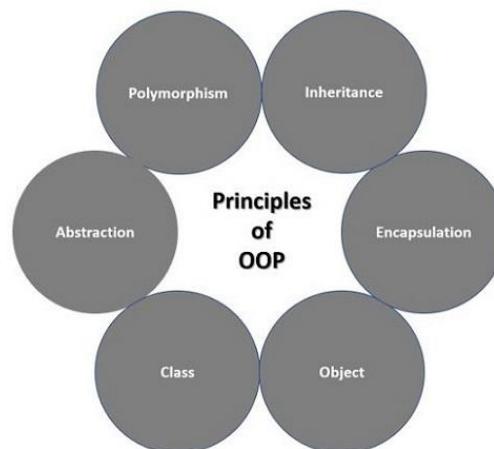
The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.

In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the other hand, is an instance of the class.

## Principles of OOPs Concepts

Object-oriented programming paradigm is characterized by the following principles –

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism



## Class & Object

A class is a user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

An object refers to an instance of a certain class. For example, an object named `obj` that belongs to a class `Circle` is an instance of that class. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

### Example

The below example illustrates how to create a class and its object in Python.

```
# defining class
class Smartphone:
    # constructor
    def __init__(self, device, brand):
        self.device = device
        self.brand = brand

    # method of the class
    def description(self):
        return f"{self.device} of {self.brand} supports Android 14"

# creating object of the class
phoneObj = Smartphone("Smartphone", "Samsung")
print(phoneObj.description())
```

On executing the above code, it will display the following output –

```
Smartphone of Samsung supports Android 14
```

### Encapsulation

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

### Example

In this example, we are using the concept of encapsulation to set the price of desktop.

```
class Desktop:
    def __init__(self):
        self.__max_price = 25000

    def sell(self):
```

```

        return f"Selling Price: {self.__max_price}"

    def set_max_price(self, price):
        if price > self.__max_price:
            self.__max_price = price

# Object
desktopObj = Desktop()
print(desktopObj.sell())

# modifying the price directly
desktopObj.__max_price = 35000
print(desktopObj.sell())

# modifying the price using setter function
desktopObj.set_max_price(35000)
print(desktopObj.sell())

```

When the above code is executed, it produces the following result –

```

Selling Price: 25000
Selling Price: 25000
Selling Price: 35000

```

## Inheritance

A software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called base or parent class, while new class is called child or sub class.

Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.

## Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```

class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite

```

## Example

The following example demonstrates the concept of Inheritance in Python –

```
#!/usr/bin/python

# define parent class
class Parent:

    parentAttr = 100

    def __init__(self):
        print ("Calling parent constructor")

    def parentMethod(self):
        print ("Calling parent method")

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print ("Parent attribute :", Parent.parentAttr)

# define child class
class Child(Parent):

    def __init__(self):
        print ("Calling child constructor")

    def childMethod(self):
        print ("Calling child method")

# instance of child
c = Child()
# child calls its method
c.childMethod()
# calls parent's method
c.parentMethod()
# again call parent's method
c.setAttr(200)
# again call parent's method
c.getAttr()
```

When the above code is executed, it produces the following result –

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows –

```
class A:      # define your class A
.....
class B:      # define your class B
.....
class C(A, B):  # subclass of A and B
.....
```

You can use `issubclass()` or `isinstance()` functions to check a relationships of two classes and instances.

- The **`issubclass(sub, sup)`** boolean function returns true if the given subclass `sub` is indeed a subclass of the superclass `sup`.
- The **`isinstance(obj, Class)`** boolean function returns true if `obj` is an instance of class `Class` or is an instance of a subclass of `Class`

## Polymorphism

Polymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

### Example

In this example, we are overriding the parent's method.

```
# define parent class
class Parent:
    def myMethod(self):
        print ("Calling parent method")

# define child class
class Child(Parent):
    def myMethod(self):
        print ("Calling child method")
```

```
# instance of child
c = Child()
# child calls overridden method
c.myMethod()
```

When the above code is executed, it produces the following result –

```
Calling child method
```

## Base Overloading Methods in Python

Following table lists some generic functionality that you can override in your own classes –

| Sr.No. | Method, Description & Sample Call                   |
|--------|-----------------------------------------------------|
| 1      | <u><a href="#">__init__</a></u> ( self [,args...] ) |
|        | Constructor (with any optional arguments)           |
|        | Sample Call : <i>obj</i> = <i>className(args)</i>   |
| 2      | <u><a href="#">__del__</a></u> ( self )             |
|        | Destructor, deletes an object                       |
|        | Sample Call : <i>del obj</i>                        |
| 3      | <u><a href="#">__repr__</a></u> ( self )            |
|        | Evaluatable string representation                   |
|        | Sample Call : <i>repr(obj)</i>                      |
| 4      | <u><a href="#">__str__</a></u> ( self )             |
|        | Printable string representation                     |
|        | Sample Call : <i>str(obj)</i>                       |
| 5      | <u><a href="#">__cmp__</a></u> ( self, x )          |
|        | Object comparison                                   |
|        | Sample Call : <i>cmp(obj, x)</i>                    |

## Overloading Operators in Python

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the [\\_\\_add\\_\\_](#) method in your class to perform vector addition and then the plus operator would behave as per expectation –

### Example

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)
```

```
def __add__(self,other):  
    return Vector(self.a + other.a, self.b + other.b)  
  
v1 = Vector(2,10)  
v2 = Vector(5,-2)  
print (v1 + v2)
```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

# 123. Python - Classes and Objects

Python is an object-oriented programming language, which means that it is based on principle of OOP concept. The entities used within a Python program is an object of one or another class. For instance, numbers, strings, lists, dictionaries, and other similar entities of a program are objects of the corresponding built-in class.

In Python, a class named Object is the base or parent class for all the classes, built-in as well as user defined.

## What is a Class in Python?

In Python, a class is a user defined entity (data type) that defines the type of data an object can contain and the actions it can perform. It is used as a template for creating objects. For instance, if we want to define a class for Smartphone in a Python program, we can use the type of data like RAM, ROM, screen-size and actions like call and message.

## Creating Classes in Python

The class keyword is used to create a new class in Python. The name of the class immediately follows the keyword class followed by a colon as shown below –

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

- The class has a documentation string, which can be accessed via `ClassName.__doc__`.
- The `class_suite` consists of all the component statements defining class members, data attributes and functions.

### Example

Following is the example of a simple Python class –

```
class Employee:  
    'Common base class for all employees'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print "Total Employee %d" % Employee.empCount
```

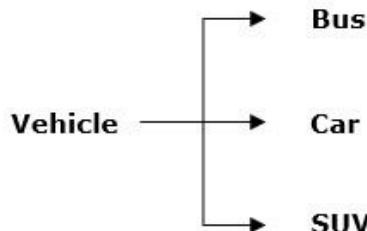
```
def displayEmployee(self):
    print "Name : ", self.name, " , Salary: ", self.salary
```

- The variable **empCount** is a class variable whose value is shared among all instances of this class. This can be accessed as Employee.empCount from inside the class or outside the class.
- The first method **\_\_init\_\_()** is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is **self**. Python adds the **self** argument to the list for you; you do not need to include it when you call the methods.

## What is an Object?

An object is referred to as an instance of a given Python class. Each object has its own attributes and methods, which are defined by its class.

When a class is created, it only describes the structure of objects. The memory is allocated when an object is instantiated from a class.



In the above figure, Vehicle is the class name and Car, Bus and SUV are its objects.

## Creating Objects of Classes in Python

To create instances of a class, you call the class using class name and pass in whatever arguments its **\_\_init\_\_** method accepts.

```
# This would create first object of Employee class
emp1 = Employee("Zara", 2000)

# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

## Accessing Attributes of Objects in Python

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows –

```
emp1.displayEmployee()
emp2.displayEmployee()
```

```
print ("Total Employee %d" % Employee.empCount)
```

Now, putting all the concepts together –

```
class Employee:
    "Common base class for all employees"
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print ("Total Employee %d" % Employee.empCount)

    def displayEmployee(self):
        print ("Name : ", self.name, ", Salary: ", self.salary)

# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

When the above code is executed, it produces the following result –

```
Name : Zara , Salary: 2000
Name : Manni , Salary: 5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time –

```
# Add an 'age' attribute
emp1.age = 7

# Modify 'age' attribute
emp1.age = 8

# Delete 'age' attribute
```

```
del emp1.age
```

Instead of using the normal statements to access attributes, you can also use the following functions –

- **getattr(obj, name[, default])** – to access the attribute of object.
- **hasattr(obj, name)** – to check if an attribute exists or not.
- **setattr(obj, name, value)** – to set an attribute. If attribute does not exist, then it would be created.
- **delattr(obj, name)** – to delete an attribute.

```
# Returns true if 'age' attribute exists
hasattr(emp1, 'age')

# Returns value of 'age' attribute
getattr(emp1, 'age')

# Set attribute 'age' at 8
setattr(emp1, 'age', 8)

# Delete attribute 'age'
delattr(emp1, 'age')
```

## Built-In Class Attributes in Python

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –

| SNo. | Attributes & Description                                                                                                         |
|------|----------------------------------------------------------------------------------------------------------------------------------|
| 1    | <b>__dict__</b><br>Dictionary containing the class's namespace.                                                                  |
| 2    | <b>__doc__</b><br>Class documentation string or none, if undefined.                                                              |
| 3    | <b>__name__</b><br>Class name                                                                                                    |
| 4    | <b>__module__</b><br>Module name in which the class is defined. This attribute is " <code>__main__</code> " in interactive mode. |
| 5    | <b>__bases__</b><br>A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list. |

### Example

For the above Employee class, let us try to access its attributes –

```
class Employee:
    'Common base class for all employees'
```

```

empCount = 0

def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

def displayCount(self):
    print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):
    print ("Name : ", self.name, " , Salary: ", self.salary)

print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__)

```

When the above code is executed, it produces the following result –

```

Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}

```

## Built-in Class of Python datatypes

As mentioned earlier, Python follows object-oriented programming paradigm. Entities like strings, lists and data types belongs to one or another built-in class.

If we want to see which data type belongs to which built-in class, we can use the Python `type()` function. This function accepts a data type and returns its corresponding class.

### Example

The below example demonstrates how to check built-in class of a given data type.

```

num = 20
print (type(num))
num1 = 55.50
print (type(num1))
s = "TutorialsPoint"
print (type(s))
dct = {'a':1,'b':2,'c':3}
print (type(dct))
def SayHello():
    print ("Hello World")
    return
print (type(SayHello))

```

When you execute this code, it will display the corresponding classes of Python data types

```

<class 'int'>
<class 'float'>
<class 'str'>
<class 'dict'>
<class 'function'>

```

## Garbage Collection(Destroying Objects) in Python

Python deletes unwanted objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with `del`, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```

# Create object <40>
a = 40
# Increase ref. count of <40>
b = a
# Increase ref. count of <40>
c = [b]

```

```
# Decrease ref. count of <40>
del a
# Decrease ref. count of <40>
b = 100
# Decrease ref. count of <40>
c[0] = -1
```

You normally will not notice when the garbage collector destroys an unused instance and reclaims its space. But a class can implement the special method `__del__()`, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

### Example

The `__del__()` destructor prints the class name of an instance that is about to be destroyed as shown in the below code block –

```
class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print (class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
# prints the ids of the obejcts
print (id(pt1), id(pt2), id(pt3))
del pt1
del pt2
del pt3
```

On executing, the above code will produces following result –

```
135007479444176 135007479444176 135007479444176
Point destroyed
```

### Data Hiding in Python

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

**Example**

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

When the above code is executed, it produces the following result –

```
1
2
ERROR!
Traceback (most recent call last):
  File <main.py>", line 11, in <module>
    AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as `object._className__attrName`. If you would replace your last line, then it works for you –

```
print(counter._JustCounter__secretCount)
```

When the above code is executed, it produces the following result –

```
1
2
2
```

# 124. Python - Class Attributes

The properties or variables defined inside a class are called as Attributes. An attribute provides information about the type of data a class contains. There are two types of attributes in Python namely instance attribute and class attribute.

The instance attribute is defined within the constructor of a Python class and is unique to each instance of the class. And, a class attribute is declared and initialized outside the constructor of the class.

## Class Attributes (Variables)

Class attributes are those variables that belong to a class and whose value is shared among all the instances of that class. A class attribute remains the same for every instance of the class.

Class attributes are defined in the class but outside any method. They cannot be initialized inside `__init__()` constructor. They can be accessed by the name of the class in addition to the object. In other words, a class attribute is available to the class as well as its object.

## Accessing Class Attributes

The object name followed by dot notation (.) is used to access class attributes.

### Example

The below example demonstrates how to access the attributes of a Python class.

```
class Employee:  
    name = "Bhavesh Aggarwal"  
    age = "30"  
    # instance of the class  
    emp = Employee()  
    # accessing class attributes  
    print("Name of the Employee:", emp.name)  
    print("Age of the Employee:", emp.age)
```

### Output

```
Name of the Employee: Bhavesh Aggarwal  
Age of the Employee: 30
```

## Modifying Class Attributes

To modify the value of a class attribute, we simply need to assign a new value to it using the class name followed by dot notation and attribute name.

In the below example, we are initializing a class variable called empCount in Employee class. For each object declared, the `__init__()` method is automatically called. This method initializes the instance variables as well as increments the empCount by 1.

```
class Employee:
    # class attribute
    empCount = 0

    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        # modifying class attribute
        Employee.empCount += 1
        print ("Name:", self.__name, ", Age: ", self.__age)
        # accessing class attribute
        print ("Employee Count:", Employee.empCount)

e1 = Employee("Bhavana", 24)
print()
e2 = Employee("Rajesh", 26)
```

## Output

We have declared two objects. Every time, the empCount increments by 1.

```
Name: Bhavana , Age: 24
```

```
Employee Count: 1
```

```
Name: Rajesh , Age: 26
```

```
Employee Count: 2
```

## Significance of Class Attributes

The class attributes are important because of the following reasons –

- They are used to define those properties of a class that should have the same value for every object of that class.
- Class attributes can be used to set default values for objects.
- This is also useful in creating singletons. They are objects that are instantiated only once and used in different parts of the code.

## Built-In Class Attributes

Every Python class keeps the following built-in attributes and they can be accessed using the dot operator like any other attribute –

- `__dict__` – Dictionary containing the class's namespace.
- `__doc__` – Class documentation string or none, if undefined.

- **`__name__`** – Class name.
- **`__module__`** – Module name in which the class is defined. This attribute is "`__main__`" in interactive mode.
- **`__bases__`** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

## Access Built-In Class Attributes

To access built-in class attributes in Python, we use the class name followed by a dot (.) and then attribute name.

### Example

For the Employee class, we are trying to access all the built-in class attributes –

```
class Employee:
    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age
    def displayEmployee(self):
        print ("Name : ", self.name, ", age: ", self.age)

    print ("Employee.__doc__:", Employee.__doc__)
    print ("Employee.__name__:", Employee.__name__)
    print ("Employee.__module__:", Employee.__module__)
    print ("Employee.__bases__:", Employee.__bases__)
    print ("Employee.__dict__:", Employee.__dict__)
```

### Output

It will produce the following output –

```
Employee.__doc__: None
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', '__init__': <function
Employee.__init__ at 0x0000022F866B8B80>, 'displayEmployee': <function
Employee.displayEmployee at 0x0000022F866B9760>, '__dict__': <attribute
 '__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of
 'Employee' objects>, '__doc__': None}
```

## Instance Attributes

As stated earlier, an instance attribute in Python is a variable that is specific to an individual object of a class. It is defined inside the `__init__()` method.

The first parameter of this method is `self` and using this parameter the instance attributes are defined.

## Example

In the following code, we are illustrating the working of instance attributes.

```
class Student:
    def __init__(self, name, grade):
        self.__name = name
        self.__grade = grade
        print ("Name:", self.__name, ", Grade:", self.__grade)

# Creating instances
student1 = Student("Ram", "B")
student2 = Student("Shyam", "C")
```

## Output

On running the above code, it will produce the following output –

```
Name: Ram , Grade: B
Name: Shyam , Grade: C
```

## Instance Attributes Vs Class Attributes

The below table shows the difference between instance attributes and class attributes –

| SNo. | Instance Attribute                                                                        | Class Attribute                                                                  |
|------|-------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| 1    | It is defined directly inside the <code>__init__()</code> function.                       | It is defined inside the class but outside the <code>__init__()</code> function. |
| 2    | Instance attribute is accessed using the object name followed by dot notation.            | Class attributes can be accessed by both class name and object name.             |
| 3    | The value of this attribute cannot be shared among other objects.                         | Its value is shared among other objects of the class.                            |
| 4    | Changes made to the instance attribute affect only the object within which it is defined. | Changes made to the class attribute affect all the objects of the given class.   |

# 125. Python - Class Methods

Methods belongs to an object of a class and used to perform specific operations. We can divide Python methods in three different categories, which are class method, instance method and static method.

A Python **class method** is bound to the class and not to the instance of the class. It can be called on the class itself, rather than on an instance of the class.

Most of us often get class methods confused with static methods. Always remember, while both are called on the class, **static methods** do not have access to the "cls" parameter and therefore it cannot modify the class state.

Unlike class method, the **instance method** can access the instance variables of the an object. It can also access the class variable as it is common to all the objects.

## Creating Class Methods in Python

There are two ways to create class methods in Python –

- Using classmethod() Function
- Using @classmethod Decorator

### Using classmethod() Function

Python has a built-in function `classmethod()` which transforms an instance method to a class method which can be called with the reference to the class only and not the object.

#### Syntax

```
classmethod(instance_method)
```

#### Example

In the Employee class, define a `showcount()` instance method with the "self" argument (reference to calling object). It prints the value of `empCount`. Next, transform the method to class method `counter()` that can be accessed through the class reference.

```
class Employee:  
    empCount = 0  
  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
        Employee.empCount += 1  
  
    def showcount(self):  
        print (self.empCount)  
  
    counter = classmethod(showcount)
```

```
e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)

e1.showcount()
Employee.counter()
```

**Output**

Call showcount() with object and call count() with class, both show the value of employee count.

```
3
3
```

**Using @classmethod Decorator**

Use of @classmethod() decorator is the prescribed way to define a class method as it is more convenient than first declaring an instance method and then transforming it into a class method.

**Syntax**

```
@classmethod
def method_name():
    # your code
```

**Example**

The class method acts as an alternate constructor. Define a newemployee() class method with arguments required to construct a new object. It returns the constructed object, something that the \_\_init\_\_() method does.

```
class Employee:
    empCount = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Employee.empCount += 1

    @classmethod
    def showcount(cls):
        print (cls.empCount)

    @classmethod
    def newemployee(cls, name, age):
```

```

        return cls(name, age)

e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)
e4 = Employee.newemployee("Anil", 21)

Employee.showcount()

```

There are four Employee objects now. If we run the above program, it will show the count of Employee object –

4

## Access Class Attributes in Class Method

Class attributes are those variables that belong to a class and whose value is shared among all the instances of that class.

To access class attributes within a class method, use the `cls` parameter followed by dot (.) notation and name of the attribute.

### Example

In this example, we are accessing a class attribute in class method.

```

class Cloth:
    # Class attribute
    price = 4000

    @classmethod
    def showPrice(cls):
        return cls.price

# Accessing class attribute
print(Cloth.showPrice())

```

On running the above code, it will show the following output –

4000

## Dynamically Add Class Method to a Class

The Python `setattr()` function is used to set an attribute dynamically. If you want to add a class method to a class, pass the method name as a parameter value to `setattr()` function.

### Example

The following example shows how to add a class method dynamically to a Python class.

```
class Cloth:
    pass

# class method
@classmethod
def brandName(cls):
    print("Name of the brand is Raymond")

# adding dynamically
setattr(Cloth, "brand_name", brandName)
newObj = Cloth()
newObj.brand_name()
```

When we execute the above code, it will show the following output –

```
Name of the brand is Raymond
```

## Dynamically Delete Class Methods

The Python `del` operator is used to delete a class method dynamically. If you try to access the deleted method, the code will raise `AttributeError`.

### Example

In the below example, we are deleting the class method named "brandName" using `del` operator.

```
class Cloth:
    # class method
    @classmethod
    def brandName(cls):
        print("Name of the brand is Raymond")

    # deleting dynamically
    del Cloth.brandName
    print("Method deleted")
```

On executing the above code, it will show the following output –

```
Method deleted
```

# 126. Python - Static Methods

## What is Python Static Method?

In Python, a **static method** is a type of method that does not require any instance to be called. It is very similar to the class method but the difference is that the static method doesn't have a mandatory argument like reference to the object – self or reference to the class – cls.

Static methods are used to access static fields of a given class. They cannot modify the state of a class since they are bound to the class, not instance.

## How to Create Static Method in Python?

There are two ways to create Python static methods –

- Using staticmethod() Function
- Using @staticmethod Decorator

### Using staticmethod() Function

Python's standard library function named staticmethod() is used to create a static method. It accepts a method as an argument and converts it into a static method.

#### Syntax

```
staticmethod(method)
```

#### Example

In the Employee class below, the showcount() method is converted into a static method. This static method can now be called by its object or reference of class itself.

```
class Employee:  
    empCount = 0  
  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
        Employee.empCount += 1  
  
    # creating staticmethod  
    def showcount():  
        print (Employee.empCount)  
        return  
  
    counter = staticmethod(showcount)
```

```
e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)

e1.counter()
Employee.counter()
```

Executing the above code will print the following result –

```
3
3
```

## Using @staticmethod Decorator

The second way to create a static method is by using the Python `@staticmethod` decorator. When we use this decorator with a method it indicates to the Interpreter that the specified method is static.

### Syntax

```
@staticmethod
def method_name():
    # your code
```

### Example

In the following example, we are creating a static method using the `@staticmethod` decorator.

```
class Student:
    stdCount = 0

    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        Student.stdCount += 1

    # creating staticmethod
    @staticmethod
    def showcount():
        print (Student.stdCount)

e1 = Student("Bhavana", 24)
e2 = Student("Rajesh", 26)
e3 = Student("John", 27)
```

```
print("Number of Students:")  
Student.showcount()
```

Running the above code will print the following result –

```
Number of Students:  
3
```

## Advantages of Static Method

There are several advantages of using static method, which includes –

- Since a static method cannot access class attributes, it can be used as a utility function to perform frequently re-used tasks.
- We can invoke this method using the class name. Hence, it eliminates the dependency on the instances.
- A static method is always predictable as its behavior remain unchanged regardless of the class state.
- We can declare a method as a static method to prevent overriding.

# 127. Python - Constructors

Python constructor is an instance method in a class, that is automatically called whenever a new object of the class is created. The constructor's role is to assign value to instance variables as soon as the object is declared.

Python uses a special method called `__init__()` to initialize the instance variables for the object, as soon as it is declared.

## Creating a constructor in Python

The `__init__()` method acts as a constructor. It needs a mandatory argument named `self`, which is the reference to the object.

```
def __init__(self, parameters):  
    #initialize instance variables
```

The `__init__()` method as well as any instance method in a class has a mandatory parameter, `self`. However, you can give any name to the first parameter, not necessarily `self`.

## Types of Constructor in Python

Python has two types of constructor –

- Default Constructor
- Parameterized Constructor

### Default Constructor in Python

The Python constructor which does not accept any parameter other than `self` is called as default constructor.

#### Example

Let us define the constructor in the `Employee` class to initialize name and age as instance variables. We can then access these attributes through its object.

```
class Employee:  
    'Common base class for all employees'  
    def __init__(self):  
        self.name = "Bhavana"  
        self.age = 24  
  
    e1 = Employee()  
    print ("Name: {}".format(e1.name))  
    print ("age: {}".format(e1.age))
```

It will produce the following output –

```
Name: Bhavana
```

```
age: 24
```

For the above Employee class, each object we declare will have same value for its instance variables name and age. To declare objects with varying attributes instead of the default, define arguments for the `__init__()` method.

### Parameterized Constructor

If a constructor is defined with multiple parameters along with `self` is called as parameterized constructor.

#### Example

In this example, the `__init__()` constructor has two formal arguments. We declare Employee objects with different values –

```
class Employee:
    'Common base class for all employees'
    def __init__(self, name, age):
        self.name = name
        self.age = age

    e1 = Employee("Bhavana", 24)
    e2 = Employee("Bharat", 25)

    print ("Name: {}".format(e1.name))
    print ("age: {}".format(e1.age))
    print ("Name: {}".format(e2.name))
    print ("age: {}".format(e2.age))
```

It will produce the following output –

```
Name: Bhavana
```

```
age: 24
```

```
Name: Bharat
```

```
age: 25
```

You can also assign default values to the formal arguments in the constructor so that the object can be instantiated with or without passing parameters.

```
class Employee:
    'Common base class for all employees'
    def __init__(self, name="Bhavana", age=24):
        self.name = name
```

```

    self.age = age

e1 = Employee()
e2 = Employee("Bharat", 25)

print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
print ("Name: {}".format(e2.name))
print ("age: {}".format(e2.age))

```

It will produce the following output –

```

Name: Bhavana
age: 24
Name: Bharat
age: 25

```

## Python - Instance Methods

In addition to the `__init__()` constructor, there may be one or more instance methods defined in a class. A method with `self` as one of the formal arguments is called instance method, as it is called by a specific object.

### Example

In the following example a `displayEmployee()` method has been defined as an instance method. It returns the name and age attributes of the `Employee` object that calls the method.

```

class Employee:

    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age

    def displayEmployee(self):
        print ("Name : ", self.name, ", age: ", self.age)

e1 = Employee()
e2 = Employee("Bharat", 25)

e1.displayEmployee()
e2.displayEmployee()

```

It will produce the following output –

```
Name : Bhavana , age: 24
Name : Bharat , age: 25
```

You can add, remove, or modify attributes of classes and objects at any time –

```
# Add a 'salary' attribute
emp1.salary = 7000

# Modify 'name' attribute
emp1.name = 'xyz'

# Delete 'salary' attribute
del emp1.salary
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj, name)** – to check if an attribute exists or not.
- The **setattr(obj, name, value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

```
# Returns true if 'salary' attribute exists
print (hasattr(e1, 'salary'))

# Returns value of 'name' attribute
print (getattr(e1, 'name'))

# Set attribute 'salary' at 8
setattr(e1, 'salary', 7000)

# Delete attribute 'age'
delattr(e1, 'age')
```

It will produce the following output –

```
False
Bhavana
```

## Python Multiple Constructors

As mentioned earlier, we define the `__init__()` method to create a constructor. However, unlike other programming languages like C++ and Java, Python does not allow multiple constructors.

If you try to create multiple constructors, Python will not throw an error, but it will only consider the last `__init__()` method in your class. Its previous definition will be overridden by the last one.

But, there is a way to achieve similar functionality in Python. We can overload constructors based on the type or number of arguments passed to the `__init__()` method. This will

allow a single constructor method to handle various initialization scenarios based on the arguments provided.

### Example

The following example shows how to achieve functionality similar to multiple constructors.

```
class Student:

    def __init__(self, *args):
        if len(args) == 1:
            self.name = args[0]

        elif len(args) == 2:
            self.name = args[0]
            self.age = args[1]

        elif len(args) == 3:
            self.name = args[0]
            self.age = args[1]
            self.gender = args[2]

    st1 = Student("Shrey")
    print("Name:", st1.name)
    st2 = Student("Ram", 25)
    print(f"Name: {st2.name} and Age: {st2.age}")
    st3 = Student("Shyam", 26, "M")
    print(f"Name: {st3.name}, Age: {st3.age} and Gender: {st3.gender}")
```

When we run the above code, it will produce the following output –

```
Name: Shrey
Name: Ram and Age: 25
Name: Shyam, Age: 26 and Gender: M
```

# 128. Python - Access Modifiers

The Python access modifiers are used to restrict access to class members (i.e., variables and methods) from outside the class. There are three types of access modifiers namely public, protected, and private.

- **Public members** – A class member is said to be public if it can be accessed from anywhere in the program.
- **Protected members** – They are accessible from within the class as well as by classes derived from that class.
- **Private members** – They can be accessed from within the class only.

Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures implementation of principle of encapsulation.

## Access Modifiers in Python

Unlike C++ and Java, Python does not use the Public, Protected and Private keywords to specify the type of access modifiers. By default, all the variables and methods in a Python class are public.

### Example

Here, we have Employee class with instance variables name and age. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

```
class Employee:  
    'Common base class for all employees'  
    def __init__(self, name="Bhavana", age=24):  
        self.name = name  
        self.age = age  
  
    e1 = Employee()  
    e2 = Employee("Bharat", 25)  
  
    print ("Name: {}".format(e1.name))  
    print ("age: {}".format(e1.age))  
    print ("Name: {}".format(e2.name))  
    print ("age: {}".format(e2.age))
```

It will produce the following output –

```
Name: Bhavana  
age: 24
```

```
Name: Bharat
age: 25
```

Python doesn't enforce restrictions on accessing any instance variable or method. However, Python prescribes a convention of prefixing name of variable/method with single or double underscore to emulate behavior of protected and private access modifiers.

To indicate that an instance variable is private, prefix it with double underscore (such as "`__age`").

To imply that a certain instance variable is protected, prefix it with single underscore (such as "`_salary`").

### **Another Example**

Let us modify the Employee class. Add another instance variable salary. Make age private and salary as protected by prefixing double and single underscores respectively.

```
class Employee:

    def __init__(self, name, age, salary):
        self.name = name # public variable
        self.__age = age # private variable
        self._salary = salary # protected variable

    def displayEmployee(self):
        print ("Name : ", self.name, ", age: ", self.__age, ", salary: ",
        self._salary)

e1=Employee("Bhavana", 24, 10000)

print (e1.name)
print (e1._salary)
print (e1.__age)
```

When you run this code, it will produce the following output –

```
Bhavana
10000
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 14, in <module>
    print (e1.__age)
               ^
AttributeError: 'Employee' object has no attribute '__age'
```

Python displays `AttributeError` because `__age` is private, and not available for use outside the class.

## Name Mangling

Python doesn't block access to private data, it just leaves for the wisdom of the programmer, not to write any code that access it from outside the class. You can still access the private members by Python's name mangling technique.

Name mangling is the process of changing name of a member with double underscore to the form `object._class__variable`. If so required, it can still be accessed from outside the class, but the practice should be refrained.

In our example, the private instance variable "`__name`" is mangled by changing it to the format –

```
obj._class__privatevar
```

So, to access the value of "`__age`" instance variable of "e1" object, change it to "`e1._Employee__age`".

Change the `print()` statement in the above program to –

```
print (e1._Employee__age)
```

It now prints 24, the age of e1.

## Python Property Object

Python's standard library has a built-in `property()` function. It returns a property object. It acts as an interface to the instance variables of a Python class.

The encapsulation principle of object-oriented programming requires that the instance variables should have a restricted private access. Python doesn't have efficient mechanism for the purpose. The `property()` function provides an alternative.

The `property()` function uses the getter, setter and delete methods defined in a class to define a property object for the class.

### Syntax

```
property(fget=None, fset=None, fdel=None, doc=None)
```

### Parameters

- **fget** – an instance method that retrieves value of an instance variable.
- **fset** – an instance method that assigns value to an instance variable.
- **fdel** – an instance method that removes an instance variable
- **fdoc** – Documentation string for the property.

The function uses getter and setter methods to return the property object.

## Getters and Setter Methods

A getter method retrieves the value of an instance variable, usually named as `get_varname`, whereas the setter method assigns value to an instance variable – named as `set_varname`.

### Example

Let us define getter methods `get_name()` and `get_age()`, and setters `set_name()` and `set_age()` in the `Employee` class.

```

class Employee:

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name
    def get_age(self):
        return self.__age
    def set_name(self, name):
        self.__name = name
        return
    def set_age(self, age):
        self.__age=age

e1=Employee("Bhavana", 24)
print ("Name:", e1.get_name(), "age:", e1.get_age())
e1.set_name("Archana")
e1.set_age(21)
print ("Name:", e1.get_name(), "age:", e1.get_age())

```

It will produce the following output –

```

Name: Bhavana age: 24
Name: Archana age: 21

```

The getter and setter methods can retrieve or assign value to instance variables. The property() function uses them to add property objects as class attributes.

The name property is defined as –

```
name = property(get_name, set_name, "name")
```

Similarly, you can add the age property –

```
age = property(get_age, set_age, "age")
```

The advantage of the property object is that you can use to retrieve the value of its associated instance variable, as well as assign value.

For example,

```

print (e1.name) displays value of e1.__name
e1.name = "Archana" assigns value to e1.__age

```

### Example

The complete program with property objects and their use is given below –

```
class Employee:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
  
    def get_name(self):  
        return self.__name  
    def get_age(self):  
        return self.__age  
    def set_name(self, name):  
        self.__name = name  
        return  
    def set_age(self, age):  
        self.__age=age  
        return  
    name = property(get_name, set_name, "name")  
    age = property(get_age, set_age, "age")  
  
e1=Employee("Bhavana", 24)  
print ("Name:", e1.name, "age:", e1.age)  
  
e1.name = "Archana"  
e1.age = 23  
print ("Name:", e1.name, "age:", e1.age)
```

It will produce the following output –

```
Name: Bhavana age: 24  
Name: Archana age: 23
```

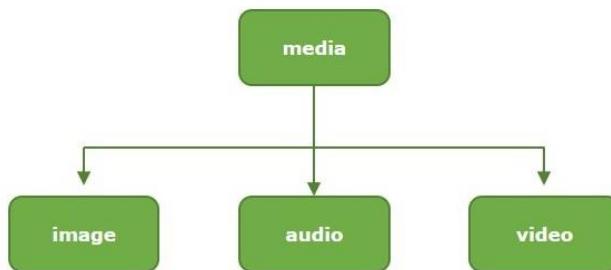
# 129. Python - Inheritance

## What is Inheritance in Python?

Inheritance is one of the most important features of object-oriented programming languages like Python. It is used to inherit the properties and behaviors of one class to another. The class that inherits another class is called a child class and the class that gets inherited is called a base class or parent class.

If you have to design a new class whose most of the attributes are already well defined in an existing class, then why redefine them? Inheritance allows capabilities of existing class to be reused and if required extended to design a new class.

Inheritance comes into picture when a new class possesses 'IS A' relationship with an existing class. For example, Car IS a vehicle, Bus IS a vehicle, Bike IS also a vehicle. Here, Vehicle is the parent class, whereas car, bus and bike are the child classes.



## Creating a Parent Class

The class whose attributes and methods are inherited is called as parent class. It is defined just like other classes i.e. using the class keyword.

### Syntax

The syntax for creating a parent class is shown below –

```
class ParentClassName:  
    {class body}
```

## Creating a Child Class

Classes that inherit from base classes are declared similarly to their parent class, however, we need to provide the name of parent classes within the parentheses.

### Syntax

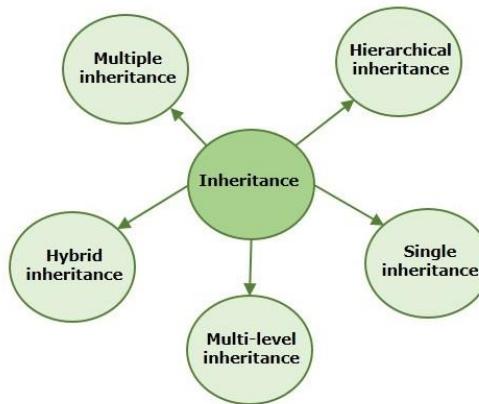
Following is the syntax of child class –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    {sub class body}
```

## Types of Inheritance

In Python, inheritance can be divided in five different categories –

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance



### Python - Single Inheritance

This is the simplest form of inheritance where a child class inherits attributes and methods from only one parent class.

#### Example

The below example shows single inheritance concept in Python –

```

# parent class
class Parent:
    def parentMethod(self):
        print ("Calling parent method")

# child class
class Child(Parent):
    def childMethod(self):
        print ("Calling child method")

# instance of child
c = Child()
# calling method of child class
c.childMethod()
# calling method of parent class
c.parentMethod()
  
```

On running the above code, it will print the following result –

```
Calling child method
Calling parent method
```

## Python - Multiple Inheritance

Multiple inheritance in Python allows you to construct a class based on more than one parent classes. The child class thus inherits the attributes and method from all parents. The child can override methods inherited from any parent.

### Syntax

```
class parent1:
    #statements

class parent2:
    #statements

class child(parent1, parent2):
    #statements
```

### Example

Python's standard library has a built-in `divmod()` function that returns a two-item tuple. First number is the division of two arguments, the second is the mod value of the two operands.

This example tries to emulate the `divmod()` function. We define two classes `division` and `modulus`, and then have a `div_mod` class that inherits them.

```
class division:
    def __init__(self, a,b):
        self.n=a
        self.d=b
    def divide(self):
        return self.n/self.d

class modulus:
    def __init__(self, a,b):
        self.n=a
        self.d=b
    def mod_divide(self):
        return self.n%self.d

class div_mod(division,modulus):
```

```

def __init__(self, a,b):
    self.n=a
    self.d=b
def div_and_mod(self):
    divval=division.divide(self)
    modval=modulus.mod_divide(self)
    return (divval, modval)

```

The child class has a new method `div_and_mod()` which internally calls the `divide()` and `mod_divide()` methods from its inherited classes to return the division and mod values.

```

x=div_mod(10,3)
print ("division:",x.divide())
print ("mod_division:",x.mod_divide())
print ("divmod:",x.div_and_mod())

```

### Output

```

division: 3.333333333333335
mod_division: 1
divmod: (3.333333333333335, 1)

```

## Method Resolution Order (MRO)

The term method resolution order is related to multiple inheritance in Python. In Python, inheritance may be spread over more than one levels. Let us say A is the parent of B, and B the parent for C. The class C can override the inherited method or its object may invoke it as defined in its parent. So, how does Python find the appropriate method to call.

Each Python class has a `mro()` method that returns the hierarchical order that Python uses to resolve the method to be called. The resolution order starts from bottom of inheritance order to top.

In our previous example, the `div_mod` class inherits `division` and `modulus` classes. So, the `mro` method returns the order as follows –

```

[<class '__main__.div_mod'>, <class '__main__.division'>, <class '__main__.modulus'>, <class 'object'>]

```

## Python - Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class. Multiple layers of inheritance exist here. We can imagine it as a grandparent-parent-child relationship.

### Example

In the following example, we are illustrating the working of multilevel inheritance.

```

# parent class
class Universe:

```

```

def universeMethod(self):
    print ("I am in the Universe")

# child class
class Earth(Universe):
    def earthMethod(self):
        print ("I am on Earth")

# another child class
class India(Earth):
    def indianMethod(self):
        print ("I am in India")

# creating instance
person = India()
# method calls
person.universeMethod()
person.earthMethod()
person.indianMethod()

```

When we execute the above code, it will produce the following result –

```

I am in the Universe
I am on Earth
I am in India

```

## Python - Hierarchical Inheritance

This type of inheritance contains multiple derived classes that are inherited from a single base class. This is similar to the hierarchy within an organization.

### Example

The following example illustrates hierarchical inheritance. Here, we have defined two child classes of Manager class.

```

# parent class
class Manager:
    def managerMethod(self):
        print ("I am the Manager")

# child class

```

```

class Employee1(Manager):
    def employee1Method(self):
        print ("I am Employee one")

# second child class
class Employee2(Manager):
    def employee2Method(self):
        print ("I am Employee two")

# creating instances
emp1 = Employee1()
emp2 = Employee2()

# method calls
emp1.managerMethod()
emp1.employee1Method()
emp2.managerMethod()
emp2.employee2Method()

```

On executing the above program, you will get the following output –

```

I am the Manager
I am Employee one
I am the Manager
I am Employee two

```

## Python - Hybrid Inheritance

Combination of two or more types of inheritance is called as Hybrid Inheritance. For instance, it could be a mix of single and multiple inheritance.

### Example

In this example, we have combined single and multiple inheritance to form a hybrid inheritance of classes.

```

# parent class
class CEO:
    def ceoMethod(self):
        print ("I am the CEO")

class Manager(CEO):
    def managerMethod(self):

```

```

print ("I am the Manager")

class Employee1(Manager):
    def employee1Method(self):
        print ("I am Employee one")

class Employee2(Manager, CEO):
    def employee2Method(self):
        print ("I am Employee two")

# creating instances
emp = Employee2()

# method calls
emp.managerMethod()
emp.ceoMethod()
emp.employee2Method()

```

On running the above program, it will give the below result –

```

I am the Manager
I am the CEO
I am Employee two

```

### The super() function

In Python, super() function allows you to access methods and attributes of the parent class from within a child class.

#### Example

In the following example, we create a parent class and access its constructor from a subclass using the super() function.

```

# parent class
class ParentDemo:
    def __init__(self, msg):
        self.message = msg

    def showMessage(self):
        print(self.message)

# child class

```

```
class ChildDemo(ParentDemo):  
    def __init__(self, msg):  
        # use of super function  
        super().__init__(msg)  
  
    # creating instance  
obj = ChildDemo("Welcome to Tutorialspoint!!")  
obj.showMessage()
```

On executing, the above program will give the following result –

```
Welcome to Tutorialspoint!!
```

# 130. Python - Polymorphism

## What is Polymorphism in Python?

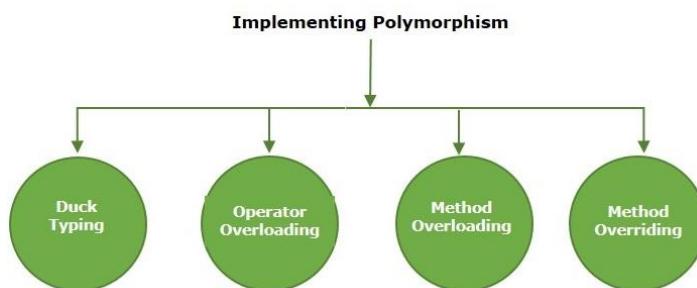
The term polymorphism refers to a function or method taking different forms in different contexts. Since Python is a dynamically typed language, polymorphism in Python is very easily implemented.

If a method in a parent class is overridden with different business logic in its different child classes, the base class method is a polymorphic method.

## Ways of implementing Polymorphism in Python

There are four ways to implement polymorphism in Python –

- Duck Typing
- Operator Overloading
- Method Overriding
- Method Overloading



## Duck Typing in Python

Duck typing is a concept where the type or class of an object is less important than the methods it defines. Using this concept, you can call any method on an object without checking its type, as long as the method exists.

This term is defined by a very famous quote that states: Suppose there is a bird that walks like a duck, swims like a duck, looks like a duck, and quacks like a duck then it probably is a duck.

### Example

In the code given below, we are practically demonstrating the concept of duck typing.

```
class Duck:  
    def sound(self):  
        return "Quack, quack!"  
  
class AnotherBird:  
    def sound(self):
```

```

        return "I'm similar to a duck!"

def makeSound(duck):
    print(duck.sound())

# creating instances
duck = Duck()
anotherBird = AnotherBird()

# calling methods
makeSound(duck)
makeSound(anotherBird)

```

When you execute this code, it will produce the following output –

```

Quack, quack!
I'm similar to a duck!

```

## Method Overriding in Python

In method overriding, a method defined inside a subclass has the same name as a method in its superclass but implements a different functionality.

### Example

As an example of polymorphism given below, we have shape which is an abstract class. It is used as parent by two classes circle and rectangle. Both classes override parent's draw() method in different ways.

```

from abc import ABC, abstractmethod

class shape(ABC):
    @abstractmethod
    def draw(self):
        "Abstract method"
        return

class circle(shape):
    def draw(self):
        super().draw()
        print ("Draw a circle")
        return

class rectangle(shape):

```

```

def draw(self):
    super().draw()
    print ("Draw a rectangle")
    return

shapes = [circle(), rectangle()]
for shp in shapes:
    shp.draw()

```

**Output**

When you run the above code, it will produce the following output –

```

Draw a circle
Draw a rectangle

```

The variable `shp` first refers to `circle` object and calls `draw()` method from `circle` class. In next iteration, it refers to `rectangle` object and calls `draw()` method from `rectangle` class. Hence `draw()` method in `shape` class is polymorphic.

**Overloading Operators in Python**

Suppose you have created a `Vector` class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the `__add__` method in your class to perform vector addition and then the plus operator would behave as per expectation –

**Example**

```

class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)

```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

## Method Overloading in Python

When a class contains two or more methods with the same name but different number of parameters then this scenario can be termed as method overloading.

Python does not allow overloading of methods by default, however, we can use the techniques like variable-length argument lists, multiple dispatch and default parameters to achieve this.

### Example

In the following example, we are using the variable-length argument lists to achieve method overloading.

```
def add(*nums):
    return sum(nums)

# Call the function with different number of parameters
result1 = add(10, 25)
result2 = add(10, 25, 35)

print(result1)
print(result2)
```

When the above code is executed, it produces the following result –

```
35
70
```

# 131. Python - Method Overriding

## Method Overriding in Python

The Python method overriding refers to defining a method in a subclass with the same name as a method in its superclass. In this case, the Python interpreter determines which method to call at runtime based on the actual object being referred to.

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

### Example

In the code below, we are overriding a method named myMethod of Parent class.

```
# define parent class
class Parent:
    def myMethod(self):
        print ('Calling parent method')

# define child class
class Child(Parent):
    def myMethod(self):
        print ('Calling child method')

# instance of child
c = Child()
# child calls overridden method
c.myMethod()
```

When the above code is executed, it produces the following output –

```
Calling child method
```

To understand Method Overriding in Python, let us take another example. We use following Employee class as parent class –

```
class Employee:
    def __init__(self,nm, sal):
        self.name=nm
        self.salary=sal
    def getName(self):
        return self.name
```

```
def getSalary(self):
    return self.salary
```

Next, we define a SalesOfficer class that uses Employee as parent class. It inherits the instance variables name and salary from the parent. Additionally, the child class has one more instance variable incentive.

We shall use built-in function super() that returns reference of the parent class and call the parent constructor within the child constructor `__init__()` method.

```
class SalesOfficer(Employee):
    def __init__(self, nm, sal, inc):
        super().__init__(nm, sal)
        self.incnt=inc
    def getSalary(self):
        return self.salary+self.incnt
```

The `getSalary()` method is overridden to add the incentive to salary.

### Example

Declare the object of parent and child classes and see the effect of overriding. Complete code is below –

```
class Employee:
    def __init__(self, nm, sal):
        self.name=nm
        self.salary=sal
    def getName(self):
        return self.name
    def getSalary(self):
        return self.salary

class SalesOfficer(Employee):
    def __init__(self, nm, sal, inc):
        super().__init__(nm, sal)
        self.incnt=inc
    def getSalary(self):
        return self.salary+self.incnt

e1=Employee("Rajesh", 9000)
print ("Total salary for {} is Rs {}".format(e1.getName(),e1.getSalary()))
s1=SalesOfficer('Kiran', 10000, 1000)
```

```
print ("Total salary for {} is Rs {}".format(s1.getName(),s1.getSalary()))
```

When you execute this code, it will produce the following output –

```
Total salary for Rajesh is Rs 9000
Total salary for Kiran is Rs 11000
```

## Base Overridable Methods

The following table lists some generic functionality of the object class, which is the parent class for all Python classes. You can override these methods in your own class –

| Sr.No | Method, Description & Sample Call          |
|-------|--------------------------------------------|
| 1     | <b><u>__init__ ( self [,args...] )</u></b> |
|       | Constructor (with any optional arguments)  |
|       | Sample Call : <i>obj = className(args)</i> |
| 2     | <b><u>__del__ ( self )</u></b>             |
|       | Destructor, deletes an object              |
|       | Sample Call : <i>del obj</i>               |
| 3     | <b><u>__repr__ ( self )</u></b>            |
|       | Evaluatable string representation          |
|       | Sample Call : <i>repr(obj)</i>             |
| 4     | <b><u>__str__ ( self )</u></b>             |
|       | Printable string representation            |
|       | Sample Call : <i>str(obj)</i>              |

# 132. Python - Method Overloading

Method overloading is a feature of object-oriented programming where a class can have multiple methods with the same name but different parameters. To overload method, we must change the number of parameters or the type of parameters, or both.

## Method Overloading in Python

Unlike other programming languages like Java, C++, and C#, Python does not support the feature of method overloading by default. However, there are alternative ways to achieve it.

### Example

If you define a method multiple times as shown in the below code, the last definition will override the previous ones. Therefore, this way of achieving method overloading in Python generates error.

```
class example:  
    def add(self, a, b):  
        x = a+b  
        return x  
  
    def add(self, a, b, c):  
        x = a+b+c  
        return x  
  
  
obj = example()  
  
print (obj.add(10,20,30))  
print (obj.add(10,20))
```

The first call to add() method with three arguments is successful. However, calling add() method with two arguments as defined in the class fails.

```
60  
Traceback (most recent call last):  
  File "C:\Users\user\example.py", line 12, in <module>  
    print (obj.add(10,20))  
               ^^^^^^^^^^  
TypeError: example.add() missing 1 required positional argument: 'c'
```

The output tells you that Python considers only the latest definition of add() method, discarding the earlier definitions.

To simulate method overloading, we can use a workaround by defining default value to method arguments as None, so that it can be used with one, two or three arguments.

### Example

The below example shows how to achieve method overloading in Python –

```
class example:

    def add(self, a = None, b = None, c = None):
        x=0
        if a !=None and b != None and c != None:
            x = a+b+c
        elif a !=None and b != None and c == None:
            x = a+b
        return x

    obj = example()

    print (obj.add(10,20,30))
    print (obj.add(10,20))
```

It will produce the following output –

```
60
30
```

With this workaround, we are able to incorporate method overloading in Python class.

## Implement Method Overloading Using MultipleDispatch

Python's standard library doesn't have any other provision for implementing method overloading. However, we can use a dispatch function from a third-party module named `MultipleDispatch` for this purpose.

First, you need to install the `MultipleDispatch` module using the following command –

```
pip install multipledispatch
```

This module has a `@dispatch` decorator. It takes the number of arguments to be passed to the method to be overloaded. Define multiple copies of `add()` method with `@dispatch` decorator as below –

### Example

In this example, we are using `multipledispatch` to overload a method in Python.

```
from multipledispatch import dispatch
class example:
    @dispatch(int, int)
    def add(self, a, b):
```

```
x = a+b
return x
@dispatch(int, int, int)
def add(self, a, b, c):
    x = a+b+c
    return x

obj = example()

print (obj.add(10,20,30))
print (obj.add(10,20))
```

**Output**

```
60
30
```

# 133. Python - Dynamic Binding

In object-oriented programming, the concept of dynamic binding is closely related to polymorphism. In Python, dynamic binding is the process of resolving a method or attribute at runtime, instead of at compile time.

According to the polymorphism feature, different objects respond differently to the same method call based on their implementations. This behavior is achieved through method overriding, where a subclass provides its implementation of a method defined in its superclass.

The Python interpreter determines which is the appropriate method or attribute to invoke based on the object's type or class hierarchy at runtime. This means that the specific method or attribute to be called is determined dynamically, based on the actual type of the object.

## Example

The following example illustrates dynamic binding in Python –

```
class shape:  
    def draw(self):  
        print ("draw method")  
        return  
  
class circle(shape):  
    def draw(self):  
        print ("Draw a circle")  
        return  
  
class rectangle(shape):  
    def draw(self):  
        print ("Draw a rectangle")  
        return  
  
shapes = [circle(), rectangle()]  
for shp in shapes:  
    shp.draw()
```

It will produce the following output –

```
Draw a circle  
Draw a rectangle
```

As you can see, the `draw()` method is bound dynamically to the corresponding implementation based on the object's type. This is how dynamic binding is implemented in Python.

## Duck Typing

Another concept closely related to dynamic binding is duck typing. Whether an object is suitable for a particular use is determined by the presence of certain methods or attributes, rather than its type. This allows for greater flexibility and code reuse in Python.

Duck typing is an important feature of dynamic typing languages like Python (Perl, Ruby, PHP, JavaScript, etc.) that focuses on an object's behavior rather than its specific type. According to the "duck typing" concept, "If it walks like a duck and quacks like a duck, then it must be a duck."

Duck typing allows objects of different types to be used interchangeably as long as they have the required methods or attributes. The goal is to promote flexibility and code reuse. It is a broader concept that emphasizes object behavior and interface rather than formal types.

Here is an example of duck typing –

```
class circle:
    def draw(self):
        print ("Draw a circle")
        return

class rectangle:
    def draw(self):
        print ("Draw a rectangle")
        return

class area:
    def area(self):
        print ("calculate area")
        return

def duck_function(obj):
    obj.draw()

objects = [circle(), rectangle(), area()]
for obj in objects:
    duck_function(obj)
```

It will produce the following output –

```
Draw a circle
Draw a rectangle
Traceback (most recent call last):
  File "C:\Python311\hello.py", line 21, in <module>
    duck_function(obj)
  File "C:\Python311\hello.py", line 17, in duck_function
    obj.draw()
AttributeError: 'area' object has no attribute 'draw'
```

The most important idea behind duck typing is that the `duck_function()` doesn't care about the specific types of objects it receives. It only requires the objects to have a `draw()` method. If an object "quacks like a duck" by having the necessary behavior, it is treated as a "duck" for the purpose of invoking the `draw()` method.

Thus, in duck typing, the focus is on the object's behavior rather than its explicit type, allowing different types of objects to be used interchangeably as long as they exhibit the required behavior.

## 134. Python - Dynamic Typing

One of the standout features of Python language is that it is a dynamically typed language. The compiler-based languages C/C++, Java, etc. are statically typed. Let us try to understand the difference between static typing and dynamic typing.

In a statically typed language, each variable and its data type must be declared before assigning it a value. Any other type of value is not acceptable to the compiler, and it raises a compile-time error.

Let us take the following snippet of a Java program –

```
public class MyClass {  
    public static void main(String args[]) {  
        int var;  
        var="Hello";  
  
        System.out.println("Value of var = " + var);  
    }  
}
```

Here, var is declared as an integer variable. When we try to assign it a string value, the compiler gives the following error message –

```
/MyClass.java:4: error: incompatible types: String cannot be converted to int  
    x="Hello";  
    ^  
1 error
```

### Why Python is Called Dynamically Typed?

A variable in Python is only a label, or reference to the object stored in the memory, and not a named memory location. Hence, the prior declaration of type is not needed. Because it's just a label, it can be put on another object, which may be of any type.

In Java, the type of the variable decides what it can store and what not. In Python, it is the other way around. Here, the type of data (i.e. object) decides the type of the variable. To begin with, let us store a string in the variable and check its type.

```
>>> var="Hello"  
>>> print ("id of var is ", id(var))  
id of var is 2822590451184  
>>> print ("type of var is ", type(var))  
type of var is <class 'str'>
```

So, var is of string type. However, it is not permanently bound. It's just a label; and can be assigned to any other type of object, say a float, which will be stored with a different id() –

```
>>> var=25.50
>>> print ("id of var is ", id(var))
id of var is 2822589562256
>>> print ("type of var is ", type(var))
type of var is <class 'float'>
```

or a tuple. The var label now sits on a different object.

```
>>> var=(10,20,30)
>>> print ("id of var is ", id(var))
id of var is 2822592028992
>>> print ("type of var is ", type(var))
type of var is <class 'tuple'>
```

We can see that the type of var changes every time it refers to a new object. That's why Python is a dynamically typed language.

Dynamic typing feature of Python makes it flexible compared to C/C++ and Java. However, it is prone to runtime errors, so the programmer has to be careful.

# 135. Python - Abstraction

Abstraction is one of the important principles of object-oriented programming. It refers to a programming approach by which only the relevant data about an object is exposed, hiding all the other details. This approach helps in reducing the complexity and increasing the efficiency of application development.

## Types of Python Abstraction

There are two types of abstraction. One is data abstraction, wherein the original data entity is hidden via a data structure that can internally work through the hidden data entities. Another type is called process abstraction. It refers to hiding the underlying implementation details of a process.

## Python Abstract Class

In object-oriented programming terminology, a class is said to be an abstract class if it cannot be instantiated, that is you can have an object of an abstract class. You can however use it as a base or parent class for constructing other classes.

### Create an Abstract Class

To create an abstract class in Python, it must inherit the ABC class that is defined in the ABC module. This module is available in Python's standard library. Moreover, the class must have at least one abstract method. Again, an abstract method is the one which cannot be called but can be overridden. You need to decorate it with @abstractmethod decorator.

#### Example: Create an Abstract Class

```
from abc import ABC, abstractmethod

class demo(ABC):

    @abstractmethod
    def method1(self):
        print ("abstract method")
        return

    def method2(self):
        print ("concrete method")
```

The demo class inherits ABC class. There is a method1() which is an abstract method. Note that the class may have other non-abstract (concrete) methods.

If you try to declare an object of demo class, Python raises TypeError –

```
obj = demo()
^^^^^
TypeError: Can't instantiate abstract class demo with abstract method method1
```

The demo class here may be used as parent for another class. However, the child class must override the abstract method in parent class. If not, Python throws this error –

```
TypeError: Can't instantiate abstract class concreteclass with abstract method  
method1
```

## Abstract Method Overriding

Hence, the child class with the abstract method overridden is given in the following example –

### Example

```
from abc import ABC, abstractmethod  
  
class democlass(ABC):  
    @abstractmethod  
    def method1(self):  
        print ("abstract method")  
        return  
    def method2(self):  
        print ("concrete method")  
  
class concreteclass(democlass):  
    def method1(self):  
        super().method1()  
        return  
  
obj = concreteclass()  
obj.method1()  
obj.method2()
```

### Output

When you execute this code, it will produce the following output –

```
abstract method  
concrete method
```

# 136. Python - Encapsulation

Encapsulation is the process of bundling attributes and methods within a single unit. It is one of the main pillars on which the object-oriented programming paradigm is based.

We know that a class is a user-defined prototype for an object. It defines a set of data members and methods, capable of processing the data.

According to the principle of data encapsulation, the data members that describe an object are hidden from the environment external to the class. They can only be accessed through the methods within the same class. Methods themselves on the other hand are accessible from outside class context. Hence, object data is said to be encapsulated by the methods. In this way, encapsulation prevents direct access to the object data.

## Implementing Encapsulation in Python

Languages such as C++ and Java use access modifiers to restrict access to class members (i.e., variables and methods). These languages have keywords public, protected, and private to specify the type of access.

A class member is said to be public if it can be accessed from anywhere in the program. Private members are allowed to be accessed from within the class only. Usually, methods are defined as public, and instance variables are private. This arrangement of private instance variables and public methods ensures the implementation of encapsulation.

Unlike these languages, Python has no provision to specify the type of access that a class member may have. By default, all the variables and methods in a Python class are public, as demonstrated by the following example.

### Example 1

Here, we have an Employee class with instance variables, name and age. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

```
class Student:  
    def __init__(self, name="Rajaram", marks=50):  
        self.name = name  
        self.marks = marks  
  
    s1 = Student()  
    s2 = Student("Bharat", 25)  
  
    print ("Name: {} marks: {}".format(s1.name, s2.marks))  
    print ("Name: {} marks: {}".format(s2.name, s2.marks))
```

It will produce the following output –

```
Name: Rajaram marks: 50
```

```
Name: Bharat marks: 25
```

In the above example, the instance variables are initialized inside the class. However, there is no restriction on accessing the value of instance variables from outside the class, which is against the principle of encapsulation.

Although there are no keywords to enforce visibility, Python has a convention of naming the instance variables in a peculiar way. In Python, prefixing name of a variable/method with a single or double underscore to emulate the behavior of protected and private access modifiers.

If a variable is prefixed by a double underscore (such as "`__age`"), the instance variable is private. Similarly if a variable name is prefixed with a single underscore (such as "`_salary`"), it becomes a private variable.

### **Example 2**

Let us modify the Student class. Add another instance variable salary. Make name private and marks as private by prefixing double underscores to them.

```
class Student:

    def __init__(self, name="Rajaram", marks=50):
        self.__name = name
        self.__marks = marks

    def studentdata(self):
        print ("Name: {} marks: {}".format(self.__name, self.__marks))

s1 = Student()
s2 = Student("Bharat", 25)

s1.studentdata()
s2.studentdata()

print ("Name: {} marks: {}".format(s1.__name, s2.__marks))
print ("Name: {} marks: {}".format(s2.__name, s2.marks))
```

When you run this code, it will produce the following output –

```
Name: Rajaram marks: 50
Name: Bharat marks: 25
Traceback (most recent call last):
  File "C:\Python311\hello.py", line 14, in <module>
    print ("Name: {} marks: {}".format(s1.__name, s2.__marks))
AttributeError: 'Student' object has no attribute '__name'
```

The above output makes it clear that the instance variables name and age, can be accessed by a method declared inside the class (the studentdata() method), but the double underscores prefix makes the variables private, and hence, accessing them outside the class is restricted which raises Attribute error.

## What is Name Mangling?

Python doesn't block access to private data entirely. It just leaves it to the wisdom of the programmer, not to write any code that accesses it from outside the class. You can still access the private members by Python's name mangling technique.

Name mangling is the process of changing name of a member with double underscore to the form object.\_class\_\_variable. If so required, it can still be accessed from outside the class, but the practice should be refrained.

In our example, the private instance variable "\_\_name" is mangled by changing it to the format

```
obj._class__privatevar
```

So, to access the value of "\_\_marks" instance variable of "s1" object, change it to "s1.\_Student\_\_marks".

Change the print() statement in the above program to –

```
print (s1._Student__marks)
```

It now prints 50, the marks of s1.

Hence, we can conclude that Python doesn't implement encapsulation exactly as per the theory of object-oriented programming. It adapts a more mature approach towards it by prescribing a name convention and letting the programmer use name mangling if it is really required to have access to private data in the public scope.

# 137. Python - Interfaces

In software engineering, an interface is a software architectural pattern. It is similar to a class but its methods just have prototype signature definition without any executable code or implementation body. The required functionality must be implemented by the methods of any class that inherits the interface.

*The method defined without any executable code is known as abstract method.*

## Interfaces in Python

In languages like Java and Go, there is keyword called interface which is used to define an interface. Python doesn't have it or any similar keyword. It uses abstract base classes (in short ABC module) and @abstractmethod decorator to create interfaces.

**NOTE:** In Python, abstract classes are also created using ABC module.

An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.

## Rules for implementing Python Interfaces

We need to consider the following points while creating and implementing interfaces in Python –

- Methods defined inside an interface must be abstract.
- Creating object of an interface is not allowed.
- A class implementing an interface needs to define all the methods of that interface.
- In case, a class is not implementing all the methods defined inside the interface, the class must be declared abstract.

## Ways to implement Interfaces in Python

We can create and implement interfaces in two ways –

- Formal Interface
- Informal Interface

### Formal Interface

Formal interfaces in Python are implemented using abstract base class (ABC). To use this class, you need to import it from the abc module.

#### Example

In this example, we are creating a formal interface with two abstract methods.

```
from abc import ABC, abstractmethod

# creating interface
```

```

class demoInterface(ABC):
    @abstractmethod
    def method1(self):
        print ("Abstract method1")
        return

    @abstractmethod
    def method2(self):
        print ("Abstract method1")
        return

```

Let us provide a class that implements both the abstract methods.

```

# class implementing the above interface
class concreteclass(demoInterface):

    def method1(self):
        print ("This is method1")
        return

    def method2(self):
        print ("This is method2")
        return

# creating instance
obj = concreteclass()

# method call
obj.method1()
obj.method2()

```

## Output

When you execute this code, it will produce the following output –

```

This is method1
This is method2

```

## Informal Interface

In Python, the informal interface refers to a class with methods that can be overridden. However, the compiler cannot strictly enforce the implementation of all the provided methods.

This type of interface works on the principle of duck typing. It allows us to call any method on an object without checking its type, as long as the method exists.

### Example

In the below example, we are demonstrating the concept of informal interface.

```
class demoInterface:  
    def displayMsg(self):  
        pass  
  
class newClass(demoInterface):  
    def displayMsg(self):  
        print ("This is my message")  
  
# creating instance  
obj = newClass()  
  
# method call  
obj.displayMsg()
```

### Output

On running the above code, it will produce the following output –

```
This is my message
```

# 138. Python - Packages

In Python, the module is a Python script with a .py extension and contains objects such as classes, functions, etc. Packages in Python extend the concept of the modular approach further. The package is a folder containing one or more module files; additionally, a special file "\_\_init\_\_.py" file may be empty but may contain the package list.

## Create a Python Package

Let us create a Python package with the name mypackage. Follow the steps given below –

- Create an outer folder to hold the contents of mypackage. Let its name be packagedemo.
- Inside it, create another folder mypackage. This will be the Python package we are going to construct. Two Python modules areafunctions.py and mathfunctions.py will be created inside mypackage.
- Create an empty "\_\_init\_\_.py" file inside mypackage folder.
- Inside the outer folder, we shall later store a Python script example.py to test our package.

The file/folder structure should be as shown below –

```
PackageDemo
    • example.py
    • testpackage.py
    • setup.py
    • mypackage
        • __init__.py
        • areafunctions.py
        • mathfunctions.py
```

Using your favorite code editor, save the following two Python modules in mypackage folder.

### Example to Create a Python Package

```
# mathfunctions.py

def sum(x,y):
    val = x+y
    return val

def average(x,y):
    val = (x+y)/2
    return val
```

```
def power(x,y):
    val = x**y
    return val
```

Create another Python script –

```
# areafuctions.py
def rectangle(w,h):
    area = w*h
    return area

def circle(r):
    import math
    area = math.pi*math.pow(r,2)
    return area
```

Let us now test the myexample package with the help of a Python script above this package folder. Refer to the folder structure above.

```
#example.py
from mypackage.areafuctions import rectangle
print ("Area :", rectangle(10,20))

from mypackage.mathsfunctions import average
print ("average:", average(10,20))
```

This program imports functions from mypackage. If the above script is executed, you should get following output –

```
Area : 200
average: 15.0
```

## Define Package List

You can put selected functions or any other resources from the package in the "`__init__.py`" file. Let us put the following code in it.

```
from .areafuctions import circle
from .mathsfunctions import sum, power
```

To import the available functions from this package, save the following script as `testpackage.py`, above the package folder as before.

### Example to Define a Package List

```
#testpackage.py

from mypackage import power, circle

print ("Area of circle:", circle(5))
print ("10 raised to 2:", power(10,2))
```

It will produce the following output –

```
Area of circle: 78.53981633974483
10 raised to 2: 100
```

## Package Installation

Right now, we are able to access the package resources from a script just above the package folder. To be able to use the package anywhere in the file system, you need to install it using the PIP utility.

First of all, save the following script in the parent folder, at the level of package folder.

```
#setup.py

from setuptools import setup

setup(name='mypackage',
      version='0.1',
      description='Package setup script',
      url='#',
      author='anonymous',
      author_email='test@gmail.com',
      license='MIT',
      packages=['mypackage'],
      zip_safe=False)
```

Run the PIP utility from command prompt, while remaining in the parent folder.

```
C:\Users\user\packagedemo>pip3 install .

Processing c:\users\user\packagedemo
Preparing metadata (setup.py) ... done
Installing collected packages: mypackage
  Running setup.py install for mypackage ... done
Successfully installed mypackage-0.1
```

You should now be able to import the contents of the package in any environment.

```
C:\Users>python
```

```
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit  
(AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import mypackage  
>>> mypackage.circle(5)  
78.53981633974483
```

# 139. Python - Inner Classes

## Inner Class in Python

A class defined inside another class is known as an inner class in Python. Sometimes inner class is also called nested class. If the inner class is instantiated, the object of inner class can also be used by the parent class. Object of inner class becomes one of the attributes of the outer class. Inner class automatically inherits the attributes of the outer class without formally establishing inheritance.

### Syntax

```
class outer:  
    def __init__(self):  
        pass  
  
    class inner:  
        def __init__(self):  
            pass
```

An inner class lets you group classes. One of the advantages of nesting classes is that it becomes easy to understand which classes are related. The inner class has a local scope. It acts as one of the attributes of the outer class.

### Example

In the following code, we have student as the outer class and subjects as the inner class. The `__init__()` constructor of student initializes name attribute and an instance of subjects class. On the other hand, the constructor of inner subjects class initializes two instance variables `sub1`, `sub2`.

A `show()` method of outer class calls the method of inner class with the object that has been instantiated.

```
class student:  
    def __init__(self):  
        self.name = "Ashish"  
        self.subs = self.subjects()  
        return  
  
    def show(self):  
        print ("Name:", self.name)  
        self.subs.display()  
  
    class subjects:  
        def __init__(self):  
            self.sub1 = "Phy"
```

```

        self.sub2 = "Che"
        return
    def display(self):
        print ("Subjects:",self.sub1, self.sub2)

s1 = student()
s1.show()

```

When you execute this code, it will produce the following output –

```

Name: Ashish
Subjects: Phy Che

```

It is quite possible to declare an object of outer class independently, and make it call its own display() method.

```
sub = student().subjects().display()
```

It will list out the subjects.

## Types of Inner Class

In Python, inner classes are of two types –

- Multiple Inner Class
- Multilevel Inner Class

## Multiple Inner Class

In multiple inner class, a single outer class contains more than one inner class. Each inner class works independently but it can interact with the members of outer class.

In the below example, we have created an outer class named Organization and two inner classes.

```

class Organization:
    def __init__(self):
        self.inner1 = self.Department1()
        self.inner2 = self.Department2()

    def showName(self):
        print("Organization Name: Tutorials Point")

    class Department1:
        def displayDepartment1(self):
            print("In Department 1")

```

```

class Department2:
    def displayDepartment2(self):
        print("In Department 2")

# instance of OuterClass
outer = Organization()
# Calling show method
outer.showName()

# InnerClass instance 1
inner1 = outer.inner1
# Calling display method
inner1.displayDepartment1()

# InnerClass instance 2
inner2 = outer.inner2
# Calling display method
inner2.displayDepartment2()

```

On executing, this code will produce the following output –

```

Organization Name: Tutorials Point
In Department 1
In Department 2

```

## Multilevel Inner Class

It refers to an inner class that itself contains another inner class. It creates multiple levels of nested classes.

### Example

The following code explains the working of Multilevel Inner Class in Python –

```

class Organization:
    def __init__(self):
        self.inner = self.Department()

    def showName(self):
        print("Organization Name: Tutorials Point")

    class Department:
        def __init__(self):
            self.innerTeam = self.Team1()

```

```
def displayDep(self):
    print("In Department")

class Team1:
    def displayTeam(self):
        print("Team 1 of the department")

# instance of outer class
outer = Organization()
# call the method of outer class
outer.showName()

# Inner Class instance
inner = outer.inner
inner.displayDep()

# Access Team1 instance
innerTeam = inner.innerTeam
# Calling display method
innerTeam.displayTeam()
```

When you run the above code, it will produce the below output –

```
Organization Name: Tutorials Point
In Department
Team 1 of the department
```

# 140. Python - Anonymous Class and Objects

Python's built-in type() function returns the class that an object belongs to. In Python, a class, both a built-in class or a user-defined class are objects of type class.

## Example

```
class myclass:  
    def __init__(self):  
        self.myvar=10  
        return  
  
    obj = myclass()  
  
    print ('class of int', type(int))  
    print ('class of list', type(list))  
    print ('class of dict', type(dict))  
    print ('class of myclass', type(myclass))  
    print ('class of obj', type(obj))
```

It will produce the following output –

```
class of int <class 'type'>  
class of list <class 'type'>  
class of dict <class 'type'>  
class of myclass <class 'type'>
```

The type() has a three argument version as follows –

## Syntax

```
newclass=type(name, bases, dict)
```

Using above syntax, a class can be dynamically created. Three arguments of type function are –

- **name** – name of the class which becomes \_\_name\_\_ attribute of new class
- **bases** – tuple consisting of parent classes. Can be blank if not a derived class
- **dict** – dictionary forming namespace of the new class containing attributes and methods and their values.

## Create an Anonymous Class

We can create an anonymous class with the above version of type() function. The name argument is a null string, second argument is a tuple of the object class (note that each

class in Python is inherited from object class). We add certain instance variables as the third argument dictionary. We keep it empty for now.

```
anon=type(' ', (object, ), {})
```

## Create an Anonymous Object

To create an object of this anonymous class –

```
obj = anon()  
print ("type of obj:", type(obj))
```

The result shows that the object is of anonymous class

```
type of obj: <class '__main__.'>
```

## Anonymous Class and Object Example

We can also add instance variables and instance methods dynamically. Take a look at this example –

```
def getA(self):  
    return self.a  
  
obj = type(' ',(object,),{'a':5,'b':6,'c':7,'getA':getA,'getB':lambda self :  
    self.b}())  
  
print (obj.getA(), obj.getB())
```

It will produce the following output –

```
5 6
```

# 141. Python - Singleton Class

In Python, a Singleton class is the implementation of singleton design pattern which means this type of class can have only one object. This helps in optimizing memory usage when you perform some heavy operation, like creating a database connection.

If we try to create multiple objects for a singleton class, the object will be created only for the first time. After that, the same object instance will be returned.

## Creating Singleton Classes in Python

We can create and implement singleton classes in Python using the following ways –

- using `__init__`
- using `__new__`

### Using `__init__`

The `__init__` method is an instance method that is used for initializing a newly created object. It's automatically called when an object is created from a class.

If we use this method with a static method and provide necessary checks i.e., whether an instance of the class already exists or not, we can restrict the creation of a new object after the first one is created.

### Example

In the below example, we are creating a singleton class using the `__init__` method.

```
class Singleton:  
    __uniqueInstance = None  
  
    @staticmethod  
    def createInstance():  
        if Singleton.__uniqueInstance == None:  
            Singleton()  
        return Singleton.__uniqueInstance  
  
    def __init__(self):  
        if Singleton.__uniqueInstance != None:  
            raise Exception("Object exist!")  
        else:  
            Singleton.__uniqueInstance = self  
  
obj1 = Singleton.createInstance()
```

```
print(obj1)
obj2 = Singleton.createInstance()
print(obj2)
```

When we run the above code, it will show the following result –

```
<__main__.Singleton object at 0x7e4da068a910>
<__main__.Singleton object at 0x7e4da068a910>
```

### **Using \_\_new\_\_**

The `__new__` method is a special static method in Python that is called to create a new instance of a class. It takes the class itself as the first argument and returns a new instance of that class.

When an instance of a Python class is declared, it internally calls the `__new__()` method. If you want to implement a Singleton class, you can override this method.

In the overridden method, you first check whether an instance of the class already exists. If it doesn't (i.e., if the instance is `None`), you call the `super()` method to create a new object. At the end, save this instance in a class attribute and return the result.

### **Example**

In the following example, we are creating a singleton class using the `__new__` method.

```
class SingletonClass:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            print('Creating the object')
            cls._instance = super(SingletonClass, cls).__new__(cls)
        return cls._instance
obj1 = SingletonClass()
print(obj1)
obj2 = SingletonClass()
print(obj2)
```

The above code gives the following result –

```
Creating the object
<__main__.SingletonClass object at 0x000002A5293A6B50>
<__main__.SingletonClass object at 0x000002A5293A6B50>
```

## 142. Python - Wrapper Classes

A function in Python is a first-order object. A function can have another function as its argument and wrap another function definition inside it. This helps in modifying a function without actually changing it. Such functions are called decorators.

This feature is also available for wrapping a class. This technique is used to manage the class after it is instantiated by wrapping its logic inside a decorator.

### Example

```
def decorator_function(Wrapped):  
    class Wrapper:  
        def __init__(self,x):  
            self.wrap = Wrapped(x)  
        def print_name(self):  
            return self.wrap.name  
    return Wrapper  
  
@decorator_function  
class Wrapped:  
    def __init__(self,x):  
        self.name = x  
  
obj = Wrapped('TutorialsPoint')  
print(obj.print_name())
```

Here, `Wrapped` is the name of the class to be wrapped. It is passed as argument to a function. Inside the function, we have a `Wrapper` class, modify its behavior with the attributes of the passed class, and return the modified class. The returned class is instantiated and its method can now be called.

When you execute this code, it will produce the following output –

```
TutorialsPoint
```

# 143. Python - Enums

## Enums in Python

In Python, the term enumeration refers to the process of assigning fixed constant values to a set of strings so that each string can be identified by the value bound to it. The `Enum` class included in `enum` module (which is a part of Python's standard library) is used as the parent class to define enumeration of a set of identifiers – conventionally written in upper case.

### Example

In the below code, "subjects" is the enumeration. It has different enumeration members and each member is an object of the enumeration class `subjects`. These members have name and value attributes.

```
# importing enum
from enum import Enum

class subjects(Enum):
    ENGLISH = 1
    MATHS = 2
    SCIENCE = 3
    SANSKRIT = 4

obj = subjects.MATHS
print (type(obj))
```

It results in following output –

```
<enum 'subjects'>
```

An enum class cannot have the same member appearing twice, however, more than one member may be assigned the same value. To ensure that each member has a unique value bound to it, use the `@unique` decorator.

### Example

In this example, we are using the `@unique` decorator to restrict duplicacy.

```
from enum import Enum, unique

@unique
class subjects(Enum):
    ENGLISH = 1
```

```
MATHS = 2
GEOGRAPHY = 3
SANSKRIT = 2
```

This will raise an exception as shown below –

```
@unique
^^^^^
raise ValueError('duplicate values found in %r: %s' %
ValueError: duplicate values found in <enum 'subjects'>: SANSKRIT -> MATHS
```

The `Enum` class is a callable class, hence you can use its constructor to create an enumeration. This constructor accepts two arguments, which are the name of enumeration and a string consisting of enumeration member symbolic names separated by a whitespace.

### Example

Following is an alternative method of defining an enumeration –

```
from enum import Enum
subjects = Enum("subjects", "ENGLISH MATHS SCIENCE SANSKRIT")
print(subjects.ENGLISH)
print(subjects.MATHS)
print(subjects.SCIENCE)
print(subjects.SANSKRIT)
```

This code will give the following output –

```
subjects.ENGLISH
subjects.MATHS
subjects.SCIENCE
subjects.SANSKRIT
```

### Accessing Modes in Enums

Members of an enum class can be accessed in two modes –

- **Value** – In this mode, value of the enum member is accessed using the "value" keyword followed by object of the enum class.
- **Name** – Similarly, we use the "name" keyword to access name of the enum member.

### Example

The following example illustrates how to access value and name of the enum member.

```
from enum import Enum
class subjects(Enum):
```

```

ENGLISH = "E"
MATHS = "M"
GEOGRAPHY = "G"
SANSKRIT = "S"

obj = subjects.SANSKRIT
print(type(obj))
print(obj.name)
print(obj.value)

```

It will produce the following output –

```

<enum 'subjects'>
SANSKRIT
S

```

## Iterating through Enums

You can iterate through the enum members in the order of their appearance in the definition, with the help of a for loop.

### Example

The following example shows how to iterate through an enumeration using for loop –

```

from enum import Enum

class subjects(Enum):
    ENGLISH = "E"
    MATHS = "M"
    GEOGRAPHY = "G"
    SANSKRIT = "S"

for sub in subjects:
    print (sub.name, sub.value)

```

It will produce the following output –

```

ENGLISH E
MATHS M
GEOGRAPHY G
SANSKRIT S

```

We know that enum member can be accessed with the unique value assigned to it, or by its name attribute. Hence, `subjects("E")` as well as `subjects["ENGLISH"]` returns `subjects.ENGLISH` member.

# 144. Python - Reflection

In object-oriented programming, reflection refers to the ability to extract information about any object in use. You can get to know the type of object, whether it is a subclass of any other class, its attributes, and much more. Python's standard library has several functions that reflect on different properties of an object. Reflection is also sometimes called introspect.

## Reflection Functions in Python

Following is the list of reflection functions in Python –

- `type()` Function
- `isinstance()` Function
- `issubclass()` Function
- `callable()` Function
- `getattr()` Function
- `setattr()` Function
- `hasattr()` Function
- `dir()` Function

### The `type()` Function

We have used this function many times. It tells you which class an object belongs to.

#### Example

Following statements print the respective class of different built-in data type objects

```
print (type(10))
print (type(2.56))
print (type(2+3j))
print (type("Hello World"))
print (type([1,2,3]))
print (type({1:'one', 2:'two'}))
```

Here, you will get the following output –

```
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'str'>
<class 'list'>
<class 'dict'>
```

Let us verify the type of an object of a user-defined class –

```
class test:
```

```

pass

obj = test()
print (type(obj))

```

It will produce the following output –

```
<class '__main__.test'>
```

## The `isinstance()` Function

This is another built-in function in Python which ascertains if an object is an instance of the given class.

### Syntax

```
isinstance(obj, class)
```

This function always returns a Boolean value, true if the object is indeed belongs to the given class and false if not.

### Example

Following statements return True –

```

print (isinstance(10, int))
print (isinstance(2.56, float))
print (isinstance(2+3j, complex))
print (isinstance("Hello World", str))

```

It will produce the following output –

```

True
True
True
True

```

In contrast, these statements print False.

```

print (isinstance([1,2,3], tuple))
print (isinstance({1:'one', 2:'two'}, set))

```

It will produce the following output –

```

False
False

```

You can also perform check with a user defined class

```

class test:
    pass

```

```
obj = test()
print (isinstance(obj, test))
```

It will produce the following output –

```
True
```

In Python, even the classes are objects. All classes are objects of object class. It can be verified by following code –

```
class test:
    pass

print (isinstance(int, object))
print (isinstance(str, object))
print (isinstance(test, object))
```

All the above print statements print True.

### The `issubclass()` Function

This function checks whether a class is a subclass of another class. Pertains to classes, not their instances.

As mentioned earlier, all Python classes are from the subclass of the object class. Hence, output of following print statements is True for all.

```
class test:
    pass

print (issubclass(int, object))
print (issubclass(str, object))
print (issubclass(test, object))
```

It will produce the following output –

```
True
True
True
```

### The `callable()` Function

An object is callable if it invokes a certain process. A Python function, which performs a certain process, is a callable object. Hence `callable(function)` returns True. Any function, built-in, user-defined, or method is callable. Objects of built-in data types such as int, str, etc., are not callable.

**Example**

```
def test():
    pass

print (callable("Hello"))
print (callable(abs))
print (callable(list.clear([1,2])))
print (callable(test))
```

A string object is not callable. But abs is a function which is callable. The pop method of list is callable, but clear() is actually call to the function and not a function object, hence not a callable

It will produce the following output –

```
False
True
True
False
True
```

A class instance is callable if it has a `__call__()` method. In the example below, the `test` class includes `__call__()` method. Hence, its object can be used as if we are calling function. Hence, object of a class with `__call__()` function is a callable.

```
class test:
    def __init__(self):
        pass
    def __call__(self):
        print ("Hello")

obj = test()
obj()
print ("obj is callable?", callable(obj))
```

It will produce the following output –

```
Hello
obj is callable? True
```

**The `getattr()` Function**

The `getattr()` built-in function retrieves the value of the named attribute of object.

**Example**

```
class test:
    def __init__(self):
        self.name = "Manav"

obj = test()
print (getattr(obj, "name"))
```

It will produce the following output –

```
Manav
```

### The setattr() Function

The setattr() built-in function adds a new attribute to the object and assigns it a value. It can also change the value of an existing attribute.

In the example below, the object of test class has a single attribute – name. We use setattr() to add age attribute and to modify the value of name attribute.

```
class test:
    def __init__(self):
        self.name = "Manav"

obj = test()
setattr(obj, "age", 20)
setattr(obj, "name", "Madhav")
print (obj.name, obj.age)
```

It will produce the following output –

```
Madhav 20
```

### The hasattr() Function

This built-in function returns True if the given attribute is available to the object argument, and false if not. We use the same test class and check if it has a certain attribute or not.

```
class test:
    def __init__(self):
        self.name = "Manav"

obj = test()
print (hasattr(obj, "age"))
print (hasattr(obj, "name"))
```

It will produce the following output –

```
False
True
```

## The dir() Function

If this built-in function is called without an argument, it returns the names in the current scope. For any object as an argument, it returns a list of the attributes of the given object and attributes reachable from it.

- **For a module object** – the function returns the module's attributes.
- **For a class object** – the function returns its attributes, and recursively the attributes of its bases.
- **For any other object** – its attributes, its class's attributes, and recursively the attributes of its class's base classes.

### Example

```
print ("dir(int):", dir(int))
```

It will produce the following output –

```
dir(int): ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio',
'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

### Example

```
print ("dir(dict):", dir(dict))
```

It will produce the following output –

```
dir(dict): ['__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__ior__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__or__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__ror__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get',
'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

### Example

```
class test:
    def __init__(self):
```

```
self.name = "Manav"

obj = test()
print ("dir(obj):", dir(obj))
```

It will produce the following output –

```
dir(obj): ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'name']
```

# Python Errors & Exceptions

# 145. Python - Syntax Errors

## Python Syntax Errors

In Python, syntax errors are among the most common errors encountered by programmers, especially those who are new to the language. This tutorial will help you understand what syntax errors are, how to identify them, and how to fix them.

### What is a Syntax Error?

A syntax error in Python (or any programming language) is an error that occurs when the code does not follow the syntax rules of the language. Syntax errors are detected by the interpreter or compiler at the time of parsing the code, and they prevent the code from being executed.

These errors occur because the written code does not conform to the grammatical rules of Python, making it impossible for the interpreter to understand and execute the commands.

### Common Causes of Syntax Errors

Following are the common causes of syntax errors –

- Missing colons (:) after control flow statements (e.g., if, for, while) – Colons are used to define the beginning of an indented block, such as in functions, loops, and conditionals.

```
# Error: Missing colon (:) after the if statement
if True
    print("This will cause a syntax error")
```

- Incorrect indentation – Python uses indentation to define the structure of code blocks. Incorrect indentation can lead to syntax errors.

```
# Error: The print statement is not correctly indented
def example_function():
    print("This will cause a syntax error")
```

- Misspelled keywords or incorrect use of keywords.

```
# Error: 'print' is misspelled as 'prnt'
prnt("Hello, World!")
```

- Unmatched parentheses, brackets, or braces – Python requires that all opening parentheses (, square brackets [, and curly braces { have corresponding closing characters ), ], and }.

```
# Error: The closing parenthesis is missing.
print("This will cause a syntax error")
```

## How to Identify Syntax Errors?

Identifying syntax errors in Python can sometimes be easy, especially when you get a clear error message from the interpreter. However, other times, it can be a bit tricky. Here are several ways to help you identify and resolve syntax errors effectively –

### Reading Error Messages

When you run a Python script, the interpreter will stop execution and display an error message if it encounters a syntax error. Understanding how to read these error messages is very important.

#### Example Error Message

```
File "script.py", line 1
    print("Hello, World!"
          ^
SyntaxError: EOL while scanning string literal
```

This error message can be broken down into parts –

- **File "script.py":** Indicates the file where the error occurred.
- **line 1:** Indicates the line number in the file where the interpreter detected the error.
- **print("Hello, World!"):** Shows the line of code with the error.
- **^:** Points to the location in the line where the error was detected.

### Using an Integrated Development Environment (IDE)

IDEs are helpful in identifying syntax errors as they often provide real-time feedback. Here are some features of IDEs that helps in identifying syntax errors –

- **Syntax Highlighting:** IDEs highlight code syntax in different colors. If a part of the code is incorrectly colored, it may indicate a syntax error.
- **Linting:** Tools like pylint or flake8 check your code for errors and stylistic issues.
- **Error Underlining:** Many IDEs underline syntax errors with a red squiggly line.
- **Tooltips and Error Messages:** Hovering over the underlined code often provides a tooltip with a description of the error.

Popular IDEs with these features include PyCharm, Visual Studio Code, and Jupyter Notebook.

### Running Code in Small Chunks

If you have a large script, it can be useful to run the code in smaller chunks. This can help isolate the part of the code causing the syntax error.

For example, if you have a script with multiple functions and you get a syntax error, try running each function independently to narrow down where the error might be.

### Using Version Control

Version control systems like Git can help you track changes to your code. If you encounter a syntax error, you can compare the current version of the code with previous versions to see what changes might have introduced the error.

## Fixing Syntax Errors

Fixing syntax errors in Python involves understanding the error message provided by the interpreter, identifying the exact issue in the code, and then making the necessary corrections. Here is a detailed guide on how to systematically approach and fix syntax errors –

### Read the Error Message Carefully

Python's error messages are quite informative. They indicate the file name, line number, and the type of syntax error –

#### Example Error Message

Assume we have written a print statement as shown below –

```
print("Hello, World!"
```

The following message indicates that there is a syntax error on line 1, showing that somewhere in the code, a parenthesis was left unclosed, which leads to a syntax error.

```
File "/home/cg/root/66634a37734ad/main.py", line 1
    print("Hello, World!"
          ^
SyntaxError: '(' was never closed
```

To fix this error, you need to ensure that every opening parenthesis has a corresponding closing parenthesis. Here is the corrected code –

```
print("Hello, World!")
```

### Locate the Error

To locate the error, you need to go to the line number mentioned in the error message. Additionally, check not only the indicated line but also the lines around it, as sometimes the issue might stem from previous lines.

### Understand the Nature of the Error

To understand the nature of the error, you need to identify what type of syntax error it is (e.g., missing parenthesis, incorrect indentation, missing colon, etc.). Also, refer to common syntax errors and their patterns.

### Correct the Syntax

Based on the error type, fix the code.

# 146. Python - Exceptions Handling

## Exception Handling in Python

Exception handling in Python refers to managing runtime errors that may occur during the execution of a program. In Python, exceptions are raised when errors or unexpected situations arise during program execution, such as division by zero, trying to access a file that does not exist, or attempting to perform an operation on incompatible data types.

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them –

- **Exception Handling** – This would be covered in this tutorial. Here is a list of standard Exceptions available in Python: Standard Exceptions.
- **Assertions** – This would be covered in Assertions in Python tutorial.

## Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

*Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.*

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

### The assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception.

The syntax for assert is –

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a trace back.

### Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature –

```
def KelvinToFahrenheit(Temperature):
```

```

assert (Temperature >= 0),"Colder than absolute zero!"

return ((Temperature-273)*1.8)+32

print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))

```

When the above code is executed, it produces the following result –

```

32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print (KelvinToFahrenheit(-5))
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!

```

## What is Exception?

An exception is an event, which occurs during the execution of a program and disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

## Handling an Exception in Python

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the **try:** block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

- The **try:** block contains statements which are susceptible for exception
- If exception occurs, the program jumps to the **except:** block.
- If no exception in the **try:** block, the **except:** block is skipped.

## Syntax

Here is the simple syntax of **try...except...else** blocks –

```

try:
    You do your operations here
    .....
except ExceptionI:
    If there is ExceptionI, then execute this block.

```

```

except ExceptionII:
    If there is ExceptionII, then execute this block.

    .....
else:
    If there is no exception then execute this block.

```

Here are few important points about the above-mentioned syntax –

- A single **try** statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic **except** clause, which handles any exception.
- After the except clause(s), you can include an **else** clause. The code in the **else** block executes if the code in the try: block does not raise an exception.
- The **else** block is a good place for code that does not need the try: block's protection.

### Example

This example opens a file, writes content in the file and comes out gracefully because there is no problem at all.

```

try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print ("Error: can't find file or read data")
else:
    print ("Written content in the file successfully")
    fh.close()

```

It will produce the following output –

```
Written content in the file successfully
```

However, change the mode parameter in open() function to "w". If the testfile is not already present, the program encounters IOError in except block, and prints following error message –

```
Error: can't find file or read data
```

### Example

This example tries to open a file where you do not have write permission, so it raises an exception –

```

try:
    fh = open("testfile", "r")
    fh.write("This is my test file for exception handling!!")
except IOError:

```

```

print ("Error: can't find file or read data")
else:
    print ("Written content in the file successfully")

```

This produces the following result –

```
Error: can't find file or read data
```

### The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows –

```

try:
    You do your operations here;
    .....
except:
    If there is any exception, then execute this block.
    .....
else:
    If there is no exception then execute this block.

```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

### The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows –

```

try:
    You do your operations here;
    .....
except(Exception1[, Exception2[,...ExceptionN]]):
    If there is any exception from the given exception list,
    then execute this block.
    .....
else:
    If there is no exception then execute this block.

```

### The try-finally Clause

You can use a finally: block along with a try: block. The code in the finally block executes, whether the try-block raised an exception or not. The syntax of the try-finally statement is this –

```

try:
    You do your operations here;
    .....
    Due to any exception, this may be skipped.

finally:
    This would always be executed.
    .....

```

You cannot use else clause as well along with a finally clause.

### Example

```

try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
finally:
    print ("Error: can't find file or read data")

```

If you do not have permission to open the file in writing mode, then this will produce the following result –

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows –

```

try:
    fh = open("testfile", "w")
try:
    fh.write("This is my test file for exception handling!!")
finally:
    print ("Going to close the file")
    fh.close()
except IOError:
    print ("Error: can't find file or read data")

```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

## Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```

try:
    You do your operations here;
    .....
except ExceptionType, Argument:
    You can print value of Argument here...

```

If you write the code to handle a single exception, you can have a variable that follows the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable that follows the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

### **Example**

Following is an example for a single exception –

```

# Define a function here.
def temp_convert(var):
    try:
        return int(var)
    except ValueError as Argument:
        print ("The argument does not contain numbers\n", Argument)

# Call above function here.
temp_convert("xyz")

```

This produces the following result –

```

The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'

```

## **Raising Exceptions**

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows.

### **Syntax**

```
raise [Exception [, args [, traceback]]]
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, trace back, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

### **Example**

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
    if level < 1:
        raise "Invalid level!", level
    # The code below to this would not be executed
    # if we raise the exception
```

**Note:** In order to catch an exception, an "except" clause must refer to the same exception thrown either as a class object or simple string. For example, to capture above exception, we must write the except clause as follows –

```
try:
    Business Logic here...
except "Invalid level!":
    Exception handling here...
else:
    Rest of the code here...
```

## User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to `RuntimeError`. Here, a class is created that is subclassed from `RuntimeError`. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable `e` is used to create an instance of the class `Networkerror`.

```
class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg
```

So once you defined above class, you can raise the exception as follows –

```
try:
    raise Networkerror("Bad hostname")
except Networkerror,e:
    print (e.args)
```

## Standard Exceptions

Here is a list of Standard Exceptions available in Python –

| <b>Sr.No.</b> | <b>Exception Name &amp; Description</b>                                                                                          |
|---------------|----------------------------------------------------------------------------------------------------------------------------------|
| 1             | <b>Exception</b><br>Base class for all exceptions                                                                                |
| 2             | <b>StopIteration</b><br>Raised when the next() method of an iterator does not point to any object.                               |
| 3             | <b>SystemExit</b><br>Raised by the sys.exit() function.                                                                          |
| 4             | <b>StandardError</b><br>Base class for all built-in exceptions except StopIteration and SystemExit.                              |
| 5             | <b>ArithmeticError</b><br>Base class for all errors that occur for numeric calculation.                                          |
| 6             | <b>OverflowError</b><br>Raised when a calculation exceeds maximum limit for a numeric type.                                      |
| 7             | <b>FloatingPointError</b><br>Raised when a floating point calculation fails.                                                     |
| 8             | <b>ZeroDivisionError</b><br>Raised when division or modulo by zero takes place for all numeric types.                            |
| 9             | <b>AssertionError</b><br>Raised in case of failure of the Assert statement.                                                      |
| 10            | <b>AttributeError</b><br>Raised in case of failure of attribute reference or assignment.                                         |
| 11            | <b>EOFError</b><br>Raised when there is no input from either the raw_input() or input() function and the end of file is reached. |
| 12            | <b>ImportError</b><br>Raised when an import statement fails.                                                                     |
| 13            | <b>KeyboardInterrupt</b><br>Raised when the user interrupts program execution, usually by pressing Ctrl+c.                       |
| 14            | <b>LookupError</b><br>Base class for all lookup errors.                                                                          |
| 15            | <b>IndexError</b><br>Raised when an index is not found in a sequence.                                                            |
| 16            | <b>KeyError</b><br>Raised when the specified key is not found in the dictionary.                                                 |
| 17            | <b>NameError</b><br>Raised when an identifier is not found in the local or global namespace.                                     |
| 18            | <b>UnboundLocalError</b>                                                                                                         |

|    |                                                                                                                                                                    |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Raised when trying to access a local variable in a function or method but no value has been assigned to it.                                                        |
| 19 | <b>EnvironmentError</b><br>Base class for all exceptions that occur outside the Python environment.                                                                |
| 20 | <b>IOError</b><br>Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| 21 | <b>IOError</b><br>Raised for operating system-related errors.                                                                                                      |
| 22 | <b>SyntaxError</b><br>Raised when there is an error in Python syntax.                                                                                              |
| 23 | <b>IndentationError</b><br>Raised when indentation is not specified properly.                                                                                      |
| 24 | <b>SystemError</b><br>Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.              |
| 25 | <b>SystemExit</b><br>Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.          |
| 26 | <b>TypeError</b><br>Raised when an operation or function is attempted that is invalid for the specified data type.                                                 |
| 27 | <b>ValueError</b><br>Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.           |
| 28 | <b>RuntimeError</b><br>Raised when a generated error does not fall into any category.                                                                              |
| 29 | <b>NotImplementedError</b><br>Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.                       |

# 147. Python - The try-except Block

## Python Try-Except Block

In Python, the try-except block is used to handle exceptions and errors gracefully, ensuring that your program can continue running even when something goes wrong. This tutorial will cover the basics of using the try-except block, its syntax, and best practices.

*Exception handling allows you to manage errors in your code by capturing exceptions and taking appropriate actions instead of letting the program crash. An exception is an error that occurs during the execution of a program, and handling these exceptions ensures your program can respond to unexpected situations.*

The try-except block in Python is used to catch and handle exceptions. The code that might cause an exception is placed inside the try block, and the code to handle the exception is placed inside the except block.

### Syntax

Following is the basic syntax of the try-except block in Python –

```
try:  
    # Code that might cause an exception  
    risky_code()  
  
except SomeException as e:  
    # Code that runs if an exception occurs  
    handle_exception(e)
```

### Example

In this example, if you enter a non-numeric value, a ValueError will be raised. If you enter zero, a ZeroDivisionError will be raised. The except blocks handle these exceptions and prints appropriate error messages –

```
try:  
    number = int(input("Enter a number: "))  
    result = 10 / number  
    print(f"Result: {result}")  
  
except ZeroDivisionError as e:  
    print("Error: Cannot divide by zero.")  
  
except ValueError as e:  
    print("Error: Invalid input. Please enter a valid number.")
```

## Handling Multiple Exceptions

In Python, you can handle multiple exceptions using multiple except blocks within a single try-except statement. This allows your code to respond differently to different types of errors that may occur during execution.

### Syntax

Following is the basic syntax for handling multiple exceptions in Python –

```
try:
    # Code that might raise exceptions
    risky_code()

except FirstExceptionType:
    # Handle the first type of exception
    handle_first_exception()

except SecondExceptionType:
    # Handle the second type of exception
    handle_second_exception()

# Add more except blocks as needed for other exception types
```

### Example

In the following example –

- If you enter zero as the divisor, a "ZeroDivisionError" will be raised, and the corresponding except ZeroDivisionError block will handle it by printing an error message.
- If you enter a non-numeric input for either the dividend or the divisor, a "ValueError" will be raised, and the except ValueError block will handle it by printing a different error message.

```
try:
    dividend = int(input("Enter the dividend: "))
    divisor = int(input("Enter the divisor: "))
    result = dividend / divisor
    print(f"Result of division: {result}")

except ZeroDivisionError:
    print("Error: Cannot divide by zero.")

except ValueError:
    print("Error: Invalid input. Please enter valid integers.")
```

## Using Else Clause with Try-Except Block

In Python, the else clause can be used in conjunction with the try-except block to specify code that should run only if no exceptions occur in the try block. This provides a way to

differentiate between the main code that may raise exceptions and additional code that should only execute under normal conditions.

## Syntax

Following is the basic syntax of the else clause in Python –

```
try:
    # Code that might raise exceptions
    risky_code()

except SomeExceptionType:
    # Handle the exception
    handle_exception()

else:
    # Code that runs if no exceptions occurred
    no_exceptions_code()
```

## Example

In the following example –

- If you enter a non-integer input, a ValueError will be raised, and the corresponding except ValueError block will handle it.
- If you enter zero as the denominator, a ZeroDivisionError will be raised, and the corresponding except ZeroDivisionError block will handle it.
- If the division is successful (i.e., no exceptions are raised), the else block will execute and print the result of the division.

```
try:
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    result = numerator / denominator
except ValueError:
    print("Error: Invalid input. Please enter valid integers.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print(f"Result of division: {result}")
```

## The Finally Clause

The finally clause provides a mechanism to guarantee that specific code will be executed, regardless of whether an exception is raised or not. This is useful for performing cleanup actions such as closing files or network connections, releasing locks, or freeing up resources.

## Syntax

Following is the basic syntax of the finally clause in Python –

```
try:
    # Code that might raise exceptions
    risky_code()

except SomeExceptionType:
    # Handle the exception
    handle_exception()

else:
    # Code that runs if no exceptions occurred
    no_exceptions_code()

finally:
    # Code that always runs, regardless of exceptions
    cleanup_code()
```

### Example

In this example –

- If the file "example.txt" exists, its content is read and printed, and the else block confirms the successful operation.
- If the file is not found (FileNotFoundException), an appropriate error message is printed in the except block.
- The finally block ensures that the file is closed (file.close()) regardless of whether the file operation succeeds or an exception occurs.

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)

except FileNotFoundError:
    print("Error: The file was not found.")

else:
    print("File read operation successful.")

finally:
    if 'file' in locals():
        file.close()

    print("File operation is complete.")
```

# 148. Python - The try-finally Block

## Python Try-Finally Block

In Python, the try-finally block is used to ensure that certain code executes, regardless of whether an exception is raised or not. Unlike the try-except block, which handles exceptions, the try-finally block focuses on cleanup operations that must occur, ensuring resources are properly released and critical tasks are completed.

### Syntax

The syntax of the try-finally statement is as follows –

```
try:  
    # Code that might raise exceptions  
    risky_code()  
  
finally:  
    # Code that always runs, regardless of exceptions  
    cleanup_code()
```

*In Python, when using exception handling with try blocks, you have the option to include either except clauses to catch specific exceptions or a finally clause to ensure certain cleanup operations are executed, but not both together.*

### Example

Let us consider an example where we want to open a file in write mode ("w"), writes some content to it, and ensures the file is closed regardless of success or failure using a finally block –

```
try:  
    fh = open("testfile", "w")  
    fh.write("This is my test file for exception handling!!")  
  
finally:  
    print ("Error: can't find file or read data")  
    fh.close()
```

If you do not have permission to open the file in writing mode, then it will produce the following output –

```
Error: can't find file or read data
```

The same example can be written more cleanly as follows –

```
try:  
    fh = open("testfile", "w")
```

```

try:
    fh.write("This is my test file for exception handling!!")
finally:
    print ("Going to close the file")
    fh.close()
except IOError:
    print ("Error: can't find file or read data")

```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

## Exception with Arguments

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```

try:
    You do your operations here
    .....
except ExceptionType as Argument:
    You can print value of Argument here...

```

If you write the code to handle a single exception, you can have a variable that follows the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable that follows the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

### Example

Following is an example for a single exception –

```

# Define a function here.

def temp_convert(var):
    try:
        return int(var)
    except ValueError as Argument:
        print("The argument does not contain numbers\n",Argument)

# Call above function here.

temp_convert("xyz")

```

It will produce the following output –

```
The argument does not contain numbers  
invalid literal for int() with base 10: 'xyz'
```

# 149. Python - Raising Exceptions

## Raising Exceptions in Python

In Python, you can raise exceptions explicitly using the `raise` statement. Raising exceptions allows you to indicate that an error has occurred and to control the flow of your program by handling these exceptions appropriately.

Raising an exception refers to explicitly trigger an error condition in your program. This can be useful for handling situations where the normal flow of your program cannot continue due to an error or an unexpected condition.

In Python, you can raise built-in exceptions like `ValueError` or `TypeError` to indicate common error conditions. Additionally, you can create and raise custom exceptions.

## Raising Built-in Exceptions

You can raise any built-in exception by creating an instance of the exception class and using the `raise` statement. Following is the syntax –

```
raise Exception("This is a general exception")
```

### Example

Here is an example where we raise a `ValueError` when a function receives an invalid argument –

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(e)
```

Following is the output of the above code –

```
Cannot divide by zero
```

## Raising Custom Exceptions

In addition to built-in exceptions, you can define and raise your own custom exceptions by creating a new exception class that inherits from the base `Exception` class or any of its subclasses –

```
class MyCustomError(Exception):
```

```

pass

def risky_function():
    raise MyCustomError("Something went wrong in risky_function")

try:
    risky_function()
except MyCustomError as e:
    print(e)

```

Output of the above code is as shown below –

```
Something went wrong in risky_function
```

## Creating Custom Exceptions

Custom exceptions is useful for handling specific error conditions that are unique to your application, providing more precise error reporting and control.

To create a custom exception in Python, you define a new class that inherits from the built-in `Exception` class or any other appropriate built-in exception class. This custom exception class can have additional attributes and methods to provide more detailed context about the error condition.

### Example

In this example –

- We define a custom exception class "InvalidAgeError" that inherits from "Exception".
- The `__init__()` method initializes the exception with the invalid age and a default error message.
- The `set_age()` function raises "InvalidAgeError" if the provided age is outside the valid range.

```

class InvalidAgeError(Exception):

    def __init__(self, age, message="Age must be between 18 and 100"):
        self.age = age
        self.message = message
        super().__init__(self.message)

    def set_age(age):
        if age < 18 or age > 100:
            raise InvalidAgeError(age)
        print(f"Age is set to {age}")

```

```
try:
    set_age(150)
except InvalidAgeError as e:
    print(f"Invalid age: {e.age}. {e.message}")
```

The result obtained is as shown below –

```
Invalid age: 150. Age must be between 18 and 100
```

## Re-Raising Exceptions

Sometimes, you may need to catch an exception, perform specific actions (such as logging, cleanup, or providing additional context), and then re-raise the same exception to be handled further up the call stack

This is useful when you want to ensure certain actions are taken when an exception occurs, but still allow the exception to propagate for higher-level handling.

To re-raise an exception in Python, you use the "raise" statement without specifying an exception, which will re-raise the last exception that was active in the current scope.

### Example

In the following example –

- The **process\_file()** function attempts to open and read a file.
- If the file is not found, it prints an error message and re-raises the "FileNotFoundException" exception.
- The exception is then caught and handled at a higher level in the call stack.

```
def process_file(filename):
    try:
        with open(filename, "r") as file:
            data = file.read()
            # Process data
    except FileNotFoundError as e:
        print(f"File not found: {filename}")
        # Re-raise the exception
        raise

    try:
        process_file("nonexistentfile.txt")
    except FileNotFoundError as e:
        print("Handling the exception at a higher level")
```

After executing the above code, we get the following output –

```
File not found: nonexistentfile.txt
```

Handling the exception at a higher level

# 150. Python - Exception Chaining

## Exception Chaining

Exception chaining is a technique of handling exceptions by re-throwing a caught exception after wrapping it inside a new exception. The original exception is saved as a property (such as `cause`) of the new exception.

During the handling of one exception 'A', it is possible that another exception 'B' may occur. It is useful to know about both exceptions in order to debug the problem. Sometimes it is useful for an exception handler to deliberately re-raise an exception, either to provide extra information or to translate an exception to another type.

In Python 3.x, it is possible to implement exception chaining. If there is any unhandled exception inside an `except` section, it will have the exception being handled attached to it and included in the error message.

### Example

In the following code snippet, trying to open a non-existent file raises `FileNotFoundException`. It is detected by the `except` block. While handling another exception is raised.

```
try:  
    open("nofile.txt")  
except OSError:  
    raise RuntimeError("unable to handle error")
```

It will produce the following output –

```
Traceback (most recent call last):  
  File "/home/cg/root/64afcadc39c651/main.py", line 2, in <module>  
    open("nofile.txt")  
  FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'  
  
During handling of the above exception, another exception occurred:  
  
Traceback (most recent call last):  
  File "/home/cg/root/64afcadc39c651/main.py", line 4, in <module>  
    raise RuntimeError("unable to handle error")  
RuntimeError: unable to handle error
```

### The `raise .. from` Statement

If you use an optional `from` clause in the `raise` statement, it indicates that an exception is a direct consequence of another. This can be useful when you are transforming exceptions. The token after `from` keyword should be the exception object.

```
try:
    open("nofile.txt")
except OSError as exc:
    raise RuntimeError from exc
```

It will produce the following output –

```
Traceback (most recent call last):
  File "/home/cg/root/64afcadc39c651/main.py", line 2, in <module>
    open("nofile.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "/home/cg/root/64afcadc39c651/main.py", line 4, in <module>
    raise RuntimeError from exc
RuntimeError
```

### The raise . . from None Statement

If we use `None` in `from` clause instead of exception object, the automatic exception chaining that was found in the earlier example is disabled.

```
try:
    open("nofile.txt")
except OSError as exc:
    raise RuntimeError from None
```

It will produce the following output –

```
Traceback (most recent call last):
  File "C:\Python311\hello.py", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

### The \_\_context\_\_ and \_\_cause\_\_ Expression

Raising an exception in the `except` block will automatically add the captured exception to the `__context__` attribute of the new exception. Similarly, you can also add `__cause__` to any exception using the expression `raise ... from` syntax.

```
try:
    try:
        raise ValueError("ValueError")
    except ValueError as e1:
```

```
    raise TypeError("TypeError") from e1
except TypeError as e2:
    print("The exception was", repr(e2))
    print("Its __context__ was", repr(e2.__context__))
    print("Its __cause__ was", repr(e2.__cause__))
```

It will produce the following output –

```
The exception was TypeError('TypeError')
Its __context__ was ValueError('ValueError')
Its __cause__ was ValueError('ValueError')
```

# 151. Python - Nested try Block

## Nested try Block in Python

In a Python program, if there is another try-except construct either inside either a try block or inside its except block, it is known as a nested-try block. This is needed when different blocks like outer and inner may cause different errors. To handle them, we need nested try blocks.

We start with an example having a single "try – except – finally" construct. If the statements inside try encounter exception, it is handled by except block. With or without exception occurred, the finally block is always executed.

### Example 1

Here, the try block has "division by 0" situation, hence the except block comes into play. It is equipped to handle the generic exception with Exception class.

```
a=10  
b=0  
try:  
    print (a/b)  
except Exception:  
    print ("General Exception")  
finally:  
    print ("inside outer finally block")
```

It will produce the following output –

```
General Exception  
inside outer finally block
```

### Example 2

Let us now see how to nest the try constructs. We put another "try – except – finally" blocks inside the existing try block. The except keyword for inner try now handles generic Exception, while we ask the except block of outer try to handle ZeroDivisionError.

Since exception doesn't occur in the inner try block, its corresponding generic Except isn't called. The division by 0 situation is handled by outer except clause.

```
a=10  
b=0  
try:  
    print (a/b)  
    try:  
        print ("This is inner try block")
```

```

except Exception:
    print ("General exception")
finally:
    print ("inside inner finally block")

except ZeroDivisionError:
    print ("Division by 0")
finally:
    print ("inside outer finally block")

```

It will produce the following output –

```

Division by 0
inside outer finally block

```

### Example 3

Now we reverse the situation. Out of the nested try blocks, the outer one doesn't have any exception raised, but the statement causing division by 0 is inside inner try, and hence the exception handled by inner except block. Obviously, the except part corresponding to outer try: will not be called upon.

```

a=10
b=0
try:
    print ("This is outer try block")
    try:
        print (a/b)
    except ZeroDivisionError:
        print ("Division by 0")
    finally:
        print ("inside inner finally block")

except Exception:
    print ("General Exception")
finally:
    print ("inside outer finally block")

```

It will produce the following output –

```

This is outer try block
Division by 0
inside inner finally block

```

```
inside outer finally block
```

In the end, let us discuss another situation which may occur in case of nested blocks. While there isn't any exception in the outer try:, there isn't a suitable except block to handle the one inside the inner try: block.

#### **Example 4**

In the following example, the inner try: faces "division by 0", but its corresponding except: is looking for KeyError instead of ZeroDivisionError. Hence, the exception object is passed on to the except: block of the subsequent except statement matching with outer try: statement. There, the zeroDivisionError exception is trapped and handled.

```
a=10
b=0
try:
    print ("This is outer try block")
    try:
        print (a/b)
    except KeyError:
        print ("Key Error")
    finally:
        print ("inside inner finally block")

except ZeroDivisionError:
    print ("Division by 0")
finally:
    print ("inside outer finally block")
```

It will produce the following output –

```
This is outer try block
inside inner finally block
Division by 0
inside outer finally block
```

# 152. Python - User-Defined Exceptions

## User-Defined Exceptions in Python

User-defined exceptions in Python are custom error classes that you create to handle specific error conditions in your code. They are derived from the built-in `Exception` class or any of its sub classes.

User-defined exceptions provide more precise control over error handling in your application –

- **Clarity** – They provide specific error messages that make it clear what went wrong.
- **Granularity** – They allow you to handle different error conditions separately.
- **Maintainability** – They centralize error handling logic, making your code easier to maintain.

## How to Create a User-Defined Exception?

To create a user-defined exception, follow these steps –

### Step 1 – Define the Exception Class

Create a new class that inherits from the built-in "Exception" class or any other appropriate base class. This new class will serve as your custom exception.

```
class MyCustomError(Exception):  
    pass
```

#### Explanation

- **Inheritance** – By inheriting from "Exception", your custom exception will have the same behavior and attributes as the built-in exceptions.
- **Class Definition** – The class is defined using the standard Python class syntax. For simple custom exceptions, you can define an empty class body using the "pass" statement.

### Step 2 – Initialize the Exception

Implement the `"__init__"` method to initialize any attributes or provide custom error messages. This allows you to pass specific information about the error when raising the exception.

```
class InvalidAgeError(Exception):  
    def __init__(self, age, message="Age must be between 18 and 100"):  
        self.age = age  
        self.message = message  
        super().__init__(self.message)
```

#### Explanation

- **Attributes** – Define attributes such as "age" and "message" to store information about the error.
- **Initialization** – The "`__init__`" method initializes these attributes. The "`super().__init__(self.message)`" call ensures that the base "Exception" class is properly initialized with the error message.
- **Default Message** – A default message is provided, but you can override it when raising the exception.

### Step 3 – Optionally Override "`__str__`" or "`__repr__`"

Override the "`__str__`" or "`__repr__`" method to provide a custom string representation of the exception. This is useful for printing or logging the exception.

```
class InvalidAgeError(Exception):
    def __init__(self, age, message="Age must be between 18 and 100"):
        self.age = age
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return f"{self.message}. Provided age: {self.age}"
```

### Explanation

- **`__str__` Method** – The "`__str__`" method returns a string representation of the exception. This is what will be displayed when the exception is printed.
- **Custom Message** – Customize the message to include relevant information, such as the provided age in this example.

## Raising User-Defined Exceptions

Once you have defined a custom exception, you can raise it in your code to signify specific error conditions. Raising user-defined exceptions involves using the `raise` statement, which can be done with or without custom messages and attributes.

### Syntax

Following is the basic syntax for raising an exception –

```
raise ExceptionType(args)
```

### Example

In this example, the "`set_age`" function raises an "InvalidAgeError" if the age is outside the valid range –

```
def set_age(age):
    if age < 18 or age > 100:
        raise InvalidAgeError(age)
    print(f"Age is set to {age}")
```

## Handling User-Defined Exceptions

Handling user-defined exceptions in Python refers to using "try-except" blocks to catch and respond to the specific conditions that your custom exceptions represent. This allows your program to handle errors gracefully and continue running or to take specific actions based on the type of exception raised.

### Syntax

Following is the basic syntax for handling exceptions –

```
try:
    # Code that may raise an exception
except ExceptionType as e:
    # Code to handle the exception
```

### Example

In the below example, the "try" block calls "set\_age" with an invalid age. The "except" block catches the "InvalidAgeError" and prints the custom error message –

```
try:
    set_age(150)
except InvalidAgeError as e:
    print(f"Invalid age: {e.age}. {e.message}")
```

### Complete Example

Combining all the steps, here is a complete example of creating and using a user-defined exception –

```
class InvalidAgeError(Exception):
    def __init__(self, age, message="Age must be between 18 and 100"):
        self.age = age
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return f"{self.message}. Provided age: {self.age}"

    def set_age(age):
        if age < 18 or age > 100:
            raise InvalidAgeError(age)
        print(f"Age is set to {age}")

try:
```

```
set_age(150)  
except InvalidAgeError as e:  
    print(f"Invalid age: {e.age}. {e.message}")
```

Following is the output of the above code –

```
Invalid age: 150. Age must be between 18 and 100
```

# 153. Python - Logging

## Logging in Python

Logging is the process of recording messages during the execution of a program to provide runtime information that can be useful for monitoring, debugging, and auditing.

In Python, logging is achieved through the built-in logging module, which provides a flexible framework for generating log messages.

## Benefits of Logging

Following are the benefits of using logging in Python –

- **Debugging** – Helps identify and diagnose issues by capturing relevant information during program execution.
- **Monitoring** – Provides insights into the application's behavior and performance.
- **Auditing** – Keeps a record of important events and actions for security purposes.
- **Troubleshooting** – Facilitates tracking of program flow and variable values to understand unexpected behavior.

## Components of Python Logging

Python logging consists of several key components that work together to manage and output log messages effectively –

- **Logger** – It is the main entry point that you use to emit log messages. Each logger instance is named and can be configured independently.
- **Handler** – It determines where log messages are sent. Handlers send log messages to different destinations such as the console, files, sockets, etc.
- **Formatter** – It specifies the layout of log messages. Formatters define the structure of log records by specifying which information to include (e.g., timestamp, log level, message).
- **Logger Level** – It defines the severity level of log messages. Messages below this level are ignored. Common levels include DEBUG, INFO, WARNING, ERROR, and CRITICAL.
- **Filter** – It is the optional components that provide finer control over which log records are processed and emitted by a handler.

## Logging Levels

Logging levels in Python define the severity of log messages, allowing developers to categorize and filter messages based on their importance. Each logging level has a specific purpose and helps in understanding the significance of the logged information –

- **DEBUG** – Detailed information, typically useful only for debugging purposes. These messages are used to trace the flow of the program and are usually not seen in production environments.
- **INFO** – Confirmation that things are working as expected. These messages provide general information about the progress of the application.

- **WARNING** – Indicates potential issues that do not prevent the program from running but might require attention. These messages can be used to alert developers about unexpected situations.
- **ERROR** – Indicates a more serious problem that prevents a specific function or operation from completing successfully. These messages highlight errors that need immediate attention but do not necessarily terminate the application.
- **CRITICAL** – The most severe level, indicating a critical error that may lead to the termination of the program. These messages are reserved for critical failures that require immediate intervention.

## Usage

Following are the usage scenarios for each logging level in Python applications –

- **Choosing the Right Level** – Selecting the appropriate logging level ensures that log messages provide relevant information without cluttering the logs.
- **Setting Levels** – Loggers, handlers, and specific log messages can be configured with different levels to control which messages are recorded and where they are outputted.
- **Hierarchy** – Logging levels are hierarchical, meaning that setting a level on a logger also affects the handlers and log messages associated with it.

## Basic Logging Example

Following is a basic logging example in Python to demonstrate its usage and functionality –

```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s -
%(message)s')

# Example usage
def calculate_sum(a, b):
    logging.debug(f"Calculating sum of {a} and {b}")
    result = a + b
    logging.info(f"Sum calculated successfully: {result}")
    return result

# Main program
if __name__ == "__main__":
    logging.info("Starting the program")
    result = calculate_sum(10, 20)
    logging.info("Program completed")
Output
```

Following is the output of the above code –

```
2024-06-19 09:00:06,774 - INFO - Starting the program
2024-06-19 09:00:06,774 - DEBUG - Calculating sum of 10 and 20
2024-06-19 09:00:06,774 - INFO - Sum calculated successfully: 30
2024-06-19 09:00:06,775 - INFO - Program completed
```

## Configuring Logging

Configuring logging in Python refers to setting up various components such as loggers, handlers, and formatters to control how and where log messages are stored and displayed. This configuration allows developers to customize logging behavior according to their application's requirements and deployment environment.

### Example

In the following example, the `getLogger()` function retrieves or creates a named logger. Loggers are organized hierarchically based on their names. Then, handlers like "StreamHandler" (console handler) are created to define where log messages go. They can be configured with specific log levels and formatters.

The formatters specify the layout of log records, determining how log messages appear when printed or stored –

```
import logging

# Create logger
logger = logging.getLogger('my_app')
logger.setLevel(logging.DEBUG) # Set global log level

# Create console handler and set level to debug
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)

# Create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
console_handler.setFormatter(formatter)

# Add console handler to logger
logger.addHandler(console_handler)

# Example usage
logger.debug('This is a debug message')
```

```
logger.info('This is an info message')
logger.warning('This is a warning message')
logger.error('This is an error message')
logger.critical('This is a critical message')
```

The result produced is as shown below –

```
2024-06-19 09:05:20,852 - my_app - DEBUG - This is a debug message
2024-06-19 09:05:20,852 - my_app - INFO - This is an info message
2024-06-19 09:05:20,852 - my_app - WARNING - This is a warning message
2024-06-19 09:05:20,852 - my_app - ERROR - This is an error message
2024-06-19 09:05:20,852 - my_app - CRITICAL - This is a critical message
```

## Logging Handlers

Logging handlers in Python determine where and how log messages are processed and outputted. They play an important role in directing log messages to specific destinations such as the console, files, email, databases, or even remote servers.

Each handler can be configured independently to control the format, log level, and other properties of the messages it processes.

### Types of Logging Handlers

Following are the various types of logging handlers in Python –

- **StreamHandler** – Sends log messages to streams such as `sys.stdout` or `sys.stderr`. Useful for displaying log messages in the console or command line interface.
- **FileHandler** – Writes log messages to a specified file on the file system. Useful for persistent logging and archiving of log data.
- **RotatingFileHandler** – Similar to FileHandler but automatically rotates log files based on size or time intervals. Helps manage log file sizes and prevent them from growing too large.
- **SMTPHandler** – Sends log messages as emails to designated recipients via SMTP. Useful for alerting administrators or developers about critical issues.
- **SysLogHandler** – Sends log messages to the system log on Unix-like systems (e.g., `syslog`). Allows integration with system-wide logging facilities.
- **MemoryHandler** – Buffers log messages in memory and sends them to a target handler after reaching a certain buffer size or timeout. Useful for batching and managing bursts of log messages.
- **HTTPHandler** – Sends log messages to a web server via HTTP or HTTPS. Enables logging messages to a remote server or logging service.

# 154. Python - Assertions

## Assertions in Python

Assertions in Python are statements that assert or assume a condition to be true. If the condition turns out to be false, Python raises an `AssertionError` exception. They are used to detect programming errors that should never occur if the code is correct.

- The easiest way to think of an assertion is to liken it to a `raise-if` statement (or to be more accurate, a `raise-if-not` statement). An expression is tested, and if the result comes up false, an exception is raised.
- Assertions are carried out by the `assert` statement, the newest keyword to Python, introduced in version 1.5.
- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

## The `assert` Statement

In Python, assertions use the `assert` keyword followed by an expression. If the expression evaluates to `False`, an `AssertionError` is raised. Following is the syntax of assertion –

```
assert condition, message
```

Where,

- **condition** – A boolean expression that should be true.
- **message (optional)** – An optional message to be displayed if the assertion fails.

## Using Assertions

Assertions are generally used during development and testing phases to check conditions that should always hold true.

### Example

In the following example, we are using assertions to ensure that the variable "num" falls within the valid range of "0" to "100". If the assertion fails, Python raises an "`AssertionError`", preventing further execution of the subsequent print statement –

```
print('Enter marks out of 100:')

num = 75

assert num >= 0 and num <= 100

print('Marks obtained:', num)

num = 125

assert num >= 0 and num <= 100

print('Marks obtained:', num) # This line won't be reached if assertion fails
```

Following is the output of the above code –

```
Enter marks out of 100:
Marks obtained: 75
Traceback (most recent call last):
  File "/home/cg/root/66723bd115007/main.py", line 7, in <module>
    assert num >= 0 and num <= 100
AssertionError
```

## Custom Error Messages

To display a custom error message when an assertion fails, include a string after the expression in the assert statement –

```
assert num >= 0 and num <= 100, "Only numbers in the range 0-100 are accepted"
```

## Handling AssertionError

Assertions can be caught and handled like any other exception using a try-except block. If they are not handled, they will terminate the program and produce a traceback –

```
try:
    num = int(input('Enter a number: '))
    assert num >= 0, "Only non-negative numbers are accepted"
    print(num)
except AssertionError as msg:
    print(msg)
```

It will produce the following output –

```
Enter a number: -87
Only non-negative numbers are accepted
```

## Assertions vs. Exceptions

Assertions are used to check internal state and invariants that should always be true whereas, exceptions help in handling runtime errors and exceptional conditions that may occur during normal execution.

Assertions are disabled by default in Python's optimized mode (-O or python -O script.py). Therefore, they should not be used to enforce constraints that are required for the correct functioning of the program in production environments.

# 155. Python - Built-in Exceptions

Built-in exceptions are pre-defined error classes in Python that handle errors and exceptional conditions in programs. They are derived from the base class "BaseException" and are part of the standard library.

## Standard Built-in Exceptions in Python

Here is a list of Standard Exceptions available in Python –

| Sr.No. | Exception Name & Description                                                                                                     |
|--------|----------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>Exception</b><br>Base class for all exceptions                                                                                |
| 2      | <b>StopIteration</b><br>Raised when the next() method of an iterator does not point to any object.                               |
| 3      | <b>SystemExit</b><br>Raised by the sys.exit() function.                                                                          |
| 4      | <b>StandardError</b><br>Base class for all built-in exceptions except StopIteration and SystemExit.                              |
| 5      | <b>ArithmeticError</b><br>Base class for all errors that occur for numeric calculation.                                          |
| 6      | <b>OverflowError</b><br>Raised when a calculation exceeds maximum limit for a numeric type.                                      |
| 7      | <b>FloatingPointError</b><br>Raised when a floating point calculation fails.                                                     |
| 8      | <b>ZeroDivisionError</b><br>Raised when division or modulo by zero takes place for all numeric types.                            |
| 9      | <b>AssertionError</b><br>Raised in case of failure of the Assert statement.                                                      |
| 10     | <b>AttributeError</b><br>Raised in case of failure of attribute reference or assignment.                                         |
| 11     | <b>EOFError</b><br>Raised when there is no input from either the raw_input() or input() function and the end of file is reached. |

|    |                                                                                                                                                  |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <b>ImportError</b>                                                                                                                               |
| 12 | Raised when an import statement fails.                                                                                                           |
|    | <b>KeyboardInterrupt</b>                                                                                                                         |
| 13 | Raised when the user interrupts program execution, usually by pressing Ctrl+C.                                                                   |
|    | <b>LookupError</b>                                                                                                                               |
| 14 | Base class for all lookup errors.                                                                                                                |
|    | <b>IndexError</b>                                                                                                                                |
| 15 | Raised when an index is not found in a sequence.                                                                                                 |
|    | <b>KeyError</b>                                                                                                                                  |
| 16 | Raised when the specified key is not found in the dictionary.                                                                                    |
|    | <b>NameError</b>                                                                                                                                 |
| 17 | Raised when an identifier is not found in the local or global namespace.                                                                         |
|    | <b>UnboundLocalError</b>                                                                                                                         |
| 18 | Raised when trying to access a local variable in a function or method but no value has been assigned to it.                                      |
|    | <b>EnvironmentError</b>                                                                                                                          |
| 19 | Base class for all exceptions that occur outside the Python environment.                                                                         |
|    | <b>IOError</b>                                                                                                                                   |
| 20 | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
|    | <b>OSError</b>                                                                                                                                   |
| 21 | Raised for operating system-related errors.                                                                                                      |
|    | <b>SyntaxError</b>                                                                                                                               |
| 22 | Raised when there is an error in Python syntax.                                                                                                  |
|    | <b>IndentationError</b>                                                                                                                          |
| 23 | Raised when indentation is not specified properly.                                                                                               |
|    | <b>SystemError</b>                                                                                                                               |
| 24 | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.                  |
| 25 | <b>SystemExit</b>                                                                                                                                |

|    |                                                                                                                                                          |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.                     |
| 26 | <b>TypeError</b><br>Raised when an operation or function is attempted that is invalid for the specified data type.                                       |
| 27 | <b>ValueError</b><br>Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| 28 | <b>RuntimeError</b><br>Raised when a generated error does not fall into any category.                                                                    |
| 29 | <b>NotImplementedError</b><br>Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.             |

Here are some examples of standard exceptions –

### IndexError

It is shown when trying to access item at invalid index.

```
numbers=[10,20,30,40]
for n in range(5):
    print (numbers[n])
```

It will produce the following output –

```
10
20
30
40
Traceback (most recent call last):

    print (numbers[n])
IndexError: list index out of range
```

### ModuleNotFoundError

This is displayed when module could not be found.

```
import notamodule
```

```
Traceback (most recent call last):
```

```
    import notamodule
ModuleNotFoundError: No module named 'notamodule'
```

## KeyError

It occurs as dictionary key is not found.

```
D1={'1':"aa", '2':"bb", '3':"cc"}
print ( D1['4'])

Traceback (most recent call last):

    D1['4']
KeyError: '4'
```

## ImportError

It is shown when specified function is not available for import.

```
from math import cube
Traceback (most recent call last):

    from math import cube
ImportError: cannot import name 'cube'
```

## StopIteration

This error appears when next() function is called after iterator stream exhausts.

```
.it=iter([1,2,3])
next(it)
next(it)
next(it)
next(it)
Traceback (most recent call last):

    next(it)
StopIteration
```

## TypeError

This is shown when operator or function is applied to an object of inappropriate type.

```
print ('2'+2)
Traceback (most recent call last):

  '2'+2
TypeError: must be str, not int
```

## ValueError

It is displayed when function's argument is of inappropriate type.

```
print (int('xyz'))
Traceback (most recent call last):

  int('xyz')
ValueError: invalid literal for int() with base 10: 'xyz'
```

## NameError

This is encountered when object could not be found.

```
print (age)
Traceback (most recent call last):

  age
NameError: name 'age' is not defined
```

## ZeroDivisionError

It is shown when second operator in division is zero.

```
x=100/0
Traceback (most recent call last):

  x=100/0
ZeroDivisionError: division by zero
```

## KeyboardInterrupt

When user hits the interrupt key normally Control-C during execution of program.

```
name=input('enter your name')
enter your name^c
Traceback (most recent call last):
```

```
name=input('enter your name')
KeyboardInterrupt
```

## Hierarchy of Built-in Exceptions

The exceptions in Python are organized in a hierarchical structure, with "BaseException" at the top. Here is a simplified hierarchy –

- BaseException
  - SystemExit
  - KeyboardInterrupt
- Exception
  - ArithmeticError
    - FloatingPointError
    - OverflowError
    - ZeroDivisionError
  - AttributeError
  - EOFError
  - ImportError
  - LookupError
    - IndexError
    - KeyError
  - MemoryError
  - NameError
    - UnboundLocalError
  - OSError
    - FileNotFoundError
  - TypeError
  - ValueError
  - ---(Many others)---

## How to Use Built-in Exceptions

As we already know that built-in exceptions in Python are pre-defined classes that handle specific error conditions. Now, here is a detailed guide on how to use them effectively in your Python programs –

### Handling Exceptions with try-except Blocks

The primary way to handle exceptions in Python is using "try-except" blocks. This allows you to catch and respond to exceptions that may occur during the execution of your code.

#### Example

In the following example, the code that may raise an exception is placed inside the "try" block. The "except" block catches the specified exception "ZeroDivisionError" and handles it

```
try:
    result = 1 / 0
except ZeroDivisionError as e:
    print(f"Caught an exception: {e}")
```

Following is the output obtained –

```
Caught an exception: division by zero
```

## Handling Multiple Exceptions

You can handle multiple exceptions by specifying them in a tuple within the "except" block as shown in the example below –

```
try:
    result = int('abc')
except (ValueError, TypeError) as e:
    print(f"Caught a ValueError or TypeError: {e}")
```

Output of the above code is as shown below –

```
Caught a ValueError or TypeError: invalid literal for int() with base 10: 'abc'
```

## Using "else" and "finally" Blocks

The "else" block is executed if the code block in the "try" clause does not raise an exception –

```
try:
    number = int(input("Enter a number: "))
except ValueError as e:
    print(f"Invalid input: {e}")
else:
    print(f"You entered: {number}")
```

Output of the above code varies as per the input given –

```
Enter a number: bn
Invalid input: invalid literal for int() with base 10: 'bn'
```

The "finally" block is always executed, regardless of whether an exception occurred or not. It's typically used for clean-up actions, such as closing files or releasing resources –

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError as e:
    print(f"File not found: {e}")
finally:
    file.close()
    print("File closed.")
```

Following is the output of the above code –

```
File closed.
```

## Explicitly Raising Built-in Exceptions

In Python, you can raise built-in exceptions to indicate errors or exceptional conditions in your code. This allows you to handle specific error scenarios and provide informative error messages to users or developers debugging your application.

### Syntax

Following is the basic syntax for raising built-in exception –

```
raise ExceptionClassName("Error message")
```

### Example

In this example, the "divide" function attempts to divide two numbers "a" and "b". If "b" is zero, it raises a "ZeroDivisionError" with a custom message –

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return a / b

try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

The output obtained is as shown below –

```
Error: Cannot divide by zero
```

# Python Multithreading

# 156. Python - Multithreading

In Python, multithreading allows you to run multiple threads concurrently within a single process, which is also known as thread-based parallelism. This means a program can perform multiple tasks at the same time, enhancing its efficiency and responsiveness.

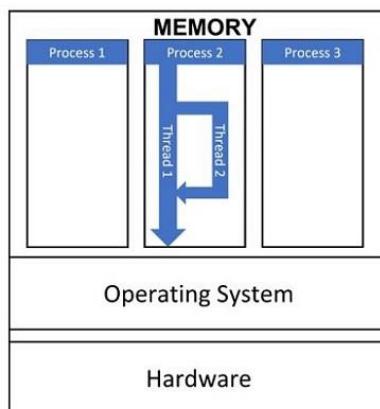
Multithreading in Python is especially useful for multiple I/O-bound operations, rather than for tasks that require heavy computation.

Generally, a computer program sequentially executes the instructions, from start to the end. Whereas, Multithreading divides the main task into more than one sub-task and executes them in an overlapping manner.

## Comparison with Processes

An operating system is capable of handling multiple processes concurrently. It allocates a separate memory space to each process so that one process cannot access or write anything in other's space.

On the other hand, a thread can be considered a lightweight sub-process in a single program that shares the memory space allocated to it, facilitating easier communication and data sharing. As they are lightweight and do not require much memory overhead; they are cheaper than processes.



A process always starts with a single thread (main thread). As and when required, a new thread can be started and sub task is delegated to it. Now the two threads are working in an overlapping manner. When the task assigned to the secondary thread is over, it merges with the main thread.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where it is currently running within its context.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

## Thread Handling Modules in Python

Python's standard library provides two main modules for managing threads: `_thread` and `threading`.

## The `_thread` Module

The `_thread` module, also known as the low-level thread module, has been a part of Python's standard library since version 2. It offers a basic API for thread management, supporting concurrent execution of threads within a shared global data space. The module includes simple locks (mutexes) for synchronization purposes.

## The `threading` Module

The `threading` module, introduced in Python 2.4, builds upon `_thread` to provide a higher-level and more comprehensive threading API. It offers powerful tools for managing threads, making it easier to work with threads in Python applications.

### Key Features of the `threading` Module

The `threading` module exposes all the methods of the `thread` module and provides some additional methods –

- **`threading.activeCount()`** – Returns the number of thread objects that are active.
- **`threading.currentThread()`** – Returns the number of thread objects in the caller's thread control.
- **`threading.enumerate()`** – Returns a list of all thread objects that are currently active.

In addition to the methods, the `threading` module has the `Thread` class that implements threading. The methods provided by the `Thread` class are as follows –

- **`run()`** – The `run()` method is the entry point for a thread.
- **`start()`** – The `start()` method starts a thread by calling the `run` method.
- **`join([time])`** – The `join()` waits for threads to terminate.
- **`isAlive()`** – The `isAlive()` method checks whether a thread is still executing.
- **`getName()`** – The `getName()` method returns the name of a thread.
- **`setName()`** – The `setName()` method sets the name of a thread.

## Starting a New Thread

To create and start a new thread in Python, you can use either the low-level `_thread` module or the higher-level `threading` module. The `threading` module is generally recommended due to its additional features and ease of use. Below, you can see both approaches.

### Starting a New Thread Using the `_thread` Module

The `start_new_thread()` method of the `_thread` module provides a basic way to create and start new threads. This method provides a fast and efficient way to create new threads in both Linux and Windows. Following is the syntax of the method –

```
thread.start_new_thread(function, args[, kwargs] )
```

This method call returns immediately, and the new thread starts executing the specified function with the given arguments. When the function returns, the thread terminates.

### Example

This example demonstrates how to use the `_thread` module to create and run threads. Each thread runs the `print_name` function with different arguments. The `time.sleep(0.5)` call ensures that the main program waits for the threads to complete their execution before exiting.

```

import _thread
import time

def print_name(name, *arg):
    print(name, *arg)

name="Tutorialspoint..."
_thread.start_new_thread(print_name, (name, 1))
_thread.start_new_thread(print_name, (name, 1, 2))

time.sleep(0.5)

```

When the above code is executed, it produces the following result –

```

Tutorialspoint... 1
Tutorialspoint... 1 2

```

Although it is very effective for low-level threading, but the `_thread` module is limited compared to the `threading` module, which offers more features and higher-level thread management.

## Starting a New Thread using the Threading Module

The `threading` module provides the `Thread` class, which is used to create and manage threads.

Here are a few steps to start a new thread using the `threading` module –

- Create a function that you want the thread to execute.
- Then create a `Thread` object using the `Thread` class by passing the target function and its arguments.
- Call the `start` method on the `Thread` object to begin execution.
- Optionally, call the `join` method to wait for the thread to complete before proceeding.

### Example

The following example demonstrates how to create and start threads using the `threading` module. It runs a function `print_name` that prints a name along with some arguments. This example creates two threads, starts them using the `start()` method, and waits for them to complete using the `join` method.

```

import threading
import time

def print_name(name, *args):
    print(name, *args)

```

```

name = "Tutorialspoint..."

# Create and start threads
thread1 = threading.Thread(target=print_name, args=(name, 1))
thread2 = threading.Thread(target=print_name, args=(name, 1, 2))

thread1.start()
thread2.start()

# Wait for threads to complete
thread1.join()
thread2.join()

print("Threads are finished...exiting")

```

When the above code is executed, it produces the following result –

```

Tutorialspoint... 1
Tutorialspoint... 1 2
Threads are finished...exiting

```

## Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Lock() method, which returns the new lock.

The acquire(blocking) method of the new lock object is used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock.

If blocking is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The release() method of the new lock object is used to release the lock when it is no longer required.

### Example

```

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):

```

```
threading.Thread.__init__(self)
    self.threadID = threadID
    self.name = name
    self.counter = counter

def run(self):
    print ("Starting " + self.name)
    # Get lock to synchronize threads
    threadLock.acquire()
    print_time(self.name, self.counter, 3)
    # Free lock to release next thread
    threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
for t in threads:
    t.join()
```

```
print ("Exiting Main Thread")
```

When the above code is executed, it produces the following result –

```
Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread
```

## Multithreaded Priority Queue

The Queue module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue –

- **get()** – The get() removes and returns an item from the queue.
- **put()** – The put adds item to a queue.
- **qsize()** – The qsize() returns the number of items that are currently in the queue.
- **empty()** – The empty( ) returns True if queue is empty; otherwise, False.
- **full()** – the full() returns True if queue is full; otherwise, False.

### Example

```
import queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q

    def run(self):
        print ("Starting " + self.name)
        process_data(self.name, self.q)
        print ("Exiting " + self.name)
```

```
def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
            time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# Fill the queue
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# Wait for queue to empty
while not workQueue.empty():
    pass
```

```
# Notify threads it's time to exit  
exitFlag = 1  
  
# Wait for all threads to complete  
for t in threads:  
    t.join()  
print ("Exiting Main Thread")
```

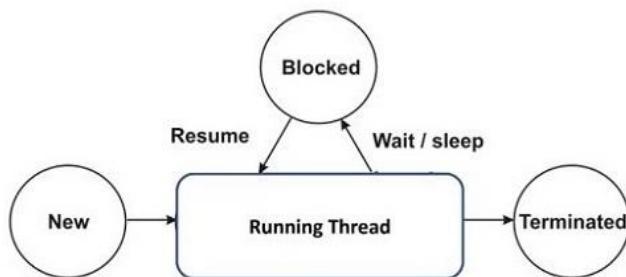
When the above code is executed, it produces the following result –

```
Starting Thread-1  
Starting Thread-2  
Starting Thread-3  
Thread-1 processing One  
Thread-2 processing Two  
Thread-3 processing Three  
Thread-1 processing Four  
Thread-2 processing Five  
Exiting Thread-3  
Exiting Thread-1  
Exiting Thread-2  
Exiting Main Thread
```

# 157. Python - Thread Lifecycle

A thread object goes through different stages during its life cycle. When a new thread object is created, it must be started, which calls the `run()` method of `thread` class. This method contains the logic of the process to be performed by the new thread. The thread completes its task as the `run()` method is over, and the newly created thread merges with the main thread.

While a thread is running, it may be paused either for a predefined duration or it may be asked to pause till a certain event occurs. The thread resumes after the specified interval or the process is over.



## States of a Thread Life Cycle in Python

Following are the stages of the Python Thread life cycle –

- [Creating a Thread](#) – To create a new thread in Python, you typically use the `Thread` class from the `threading` module.
- [Starting a Thread](#) – Once a thread object is created, it must be started by calling its `start()` method. This initiates the thread's activity and invokes its `run()` method in a separate thread.
- [Paused/Blocked State](#) – Threads can be paused or blocked for various reasons, such as waiting for I/O operations to complete or another thread to perform a task. This is typically managed by calling its `join()` method. This blocks the calling thread until the thread being joined terminates.
- [Synchronizing Threads](#) – Synchronization ensures orderly execution and shared resource management among threads. This can be done by using synchronization primitives like locks, semaphores, or condition variables.
- Termination – A thread terminates when its `run()` method completes execution, either by finishing its task or encountering an exception.

## Example: Python Thread Life Cycle Demonstration

This example demonstrates the thread life cycle in Python by showing thread creation, starting, execution, and synchronization with the main thread.

```
import threading

def func(x):
    print('Current Thread Details:', threading.current_thread())
```

```

for n in range(x):
    print('{} Running'.format(threading.current_thread().name), n)
    print('Internal Thread Finished...')

# Create thread objects
t1 = threading.Thread(target=func, args=(2,))
t2 = threading.Thread(target=func, args=(3,))

# Start the threads
print('Thread State: CREATED')
t1.start()
t2.start()

# Wait for threads to complete
t1.join()
t2.join()
print('Threads State: FINISHED')

# Simulate main thread work
for i in range(3):
    print('Main Thread Running', i)

print("Main Thread Finished...")

```

## Output

When the above code is executed, it produces the following output –

```

Thread State: CREATED
Current Thread Details: <Thread(Thread-1 (func), started 140051032258112)>
Thread-1 (func) Running 0
Thread-1 (func) Running 1
Internal Thread Finished...
Current Thread Details: <Thread(Thread-2 (func), started 140051023865408)>
Thread-2 (func) Running 0
Thread-2 (func) Running 1
Thread-2 (func) Running 2
Internal Thread Finished...

```

```
Threads State: FINISHED
Main Thread Running 0
Main Thread Running 1
Main Thread Running 2
Main Thread Finished...
```

### **Example: Using a Synchronization Primitive**

Here is another example demonstrates the thread life cycle in Python, including creation, starting, running, and termination states, along with synchronization using a semaphore.

```
import threading
import time

# Create a semaphore
semaphore = threading.Semaphore(2)

def worker():
    with semaphore:
        print('{} has started working'.format(threading.current_thread().name))
        time.sleep(2)
        print('{} has finished working'.format(threading.current_thread().name))

# Create a list to keep track of thread objects
threads = []

# Create and start 5 threads
for i in range(5):
    t = threading.Thread(target=worker, name='Thread-{}'.format(i+1))
    threads.append(t)
    print('{} has been created'.format(t.name))
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()
    print('{} has terminated'.format(t.name))

print('Threads State: All are FINISHED')
```

```
print("Main Thread Finished...")
```

### Output

When the above code is executed, it produces the following output –

```
Thread-1 has been created
Thread-1 has started working
Thread-2 has been created
Thread-2 has started working
Thread-3 has been created
Thread-4 has been created
Thread-5 has been created
Thread-1 has finished working
Thread-2 has finished working
Thread-3 has started working
Thread-1 has terminated
Thread-2 has terminated
Thread-4 has started working
Thread-3 has finished working
Thread-5 has started working
Thread-3 has terminated
Thread-4 has finished working
Thread-4 has terminated
Thread-5 has finished working
Thread-5 has terminated
Threads State: All are FINISHED
Main Thread Finished...
```

# 158. Python - Creating a Thread

Creating a thread in Python involves initiating a separate flow of execution within a program, allowing multiple operations to run concurrently. This is particularly useful for performing tasks simultaneously, such as handling various I/O operations in parallel.

Python provides multiple ways to create and manage threads.

- Creating a thread using the `threading` module is generally recommended due to its higher-level interface and additional functionalities.
- On the other hand, the `_thread` module offers a simpler, lower-level approach to create and manage threads, which can be useful for straightforward, low-overhead threading tasks.

In this tutorial, you will learn the basics of creating threads in Python using different approaches. We will cover creating threads using functions, extending the `Thread` class from the `threading` module, and utilizing the `_thread` module.

## Creating Threads with Functions

You can create threads by using the `Thread` class from the `threading` module. In this approach, you can create a thread by simply passing a function to the `Thread` object. Here are the steps to start a new thread –

- Define a function that you want the thread to execute.
- Create a `Thread` object using the `Thread` class, passing the target function and its arguments.
- Call the `start` method on the `Thread` object to begin execution.
- Optionally, call the `join` method to wait for the thread to complete before proceeding.

### Example

The following example demonstrates concurrent execution using threads in Python. It creates and starts multiple threads that execute different tasks concurrently by specifying user-defined functions as targets within the `Thread` class.

```
from threading import Thread

def addition_of_numbers(x, y):
    result = x + y
    print('Addition of {} + {} = {}'.format(x, y, result))

def cube_number(i):
    result = i ** 3
    print('Cube of {} = {}'.format(i, result))

def basic_function():
```

```

print("Basic function is running concurrently...")

Thread(target=addition_of_numbers, args=(2, 4)).start()
Thread(target=cube_number, args=(4,)).start()
Thread(target=basic_function).start()

```

On executing the above program, it will produce the following result –

```

Addition of 2 + 4 = 6
Cube of 4 = 64
Basic function is running concurrently...

```

## Creating Threads by Extending the Thread Class

Another approach to creating a thread is by extending the Thread class. This approach involves defining a new class that inherits from Thread and overriding its `__init__` and `run` methods. Here are the steps to start a new thread –

- Define a new subclass of the Thread class.
- Override the `__init__` method to add additional arguments.
- Override the `run` method to implement the thread's behavior.

### Example

This example demonstrates how to create and manage multiple threads using a custom `MyThread` class that extends the `threading.Thread` class in Python.

```

import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("Starting " + self.name)
        print_time(self.name, 5, self.counter)
        print ("Exiting " + self.name)

def print_time(threadName, counter, delay):

```

```

while counter:
    if exitFlag:
        threadName.exit()
    time.sleep(delay)
    print ("%s: %s" % (threadName, time.ctime(time())))
    counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()
print ("Exiting Main Thread")

```

When the above code is executed, it produces the following result –

```

Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Mon Jun 24 16:38:10 2024
Thread-2: Mon Jun 24 16:38:11 2024
Thread-1: Mon Jun 24 16:38:11 2024
Thread-1: Mon Jun 24 16:38:12 2024
Thread-2: Mon Jun 24 16:38:13 2024
Thread-1: Mon Jun 24 16:38:13 2024
Thread-1: Mon Jun 24 16:38:14 2024
Exiting Thread-1
Thread-2: Mon Jun 24 16:38:15 2024
Thread-2: Mon Jun 24 16:38:17 2024
Thread-2: Mon Jun 24 16:38:19 2024
Exiting Thread-2

```

## Creating Threads using start\_new\_thread() Function

The `start_new_thread()` function included in the `_thread` module is used to create a new thread in the running program. This module offers a low-level approach to threading. It is

simpler but does not have some of the advanced features provided by the threading module.

Here is the syntax of the `_thread.start_new_thread()` Function

```
_thread.start_new_thread ( function, args[, kwargs] )
```

This function starts a new thread and returns its identifier. The function parameter specifies the function that the new thread will execute. Any arguments required by this function can be passed using args and kwargs.

### Example

```
import _thread
import time

# Define a function for the thread
def thread_task( threadName, delay):

    for count in range(1, 6):
        time.sleep(delay)
        print ("Thread name: {} Count: {}".format ( threadName, count ))


# Create two threads as follows
try:
    _thread.start_new_thread( thread_task, ("Thread-1", 2, ) )
    _thread.start_new_thread( thread_task, ("Thread-2", 4, ) )
except:
    print ("Error: unable to start thread")

while True:
    pass

thread_task("test", 0.3)
```

It will produce the following output –

```
Thread name: Thread-1 Count: 1
Thread name: Thread-2 Count: 1
Thread name: Thread-1 Count: 2
Thread name: Thread-1 Count: 3
Thread name: Thread-2 Count: 2
Thread name: Thread-1 Count: 4
Thread name: Thread-1 Count: 5
Thread name: Thread-2 Count: 3
```

```
Thread name: Thread-2 Count: 4
Thread name: Thread-2 Count: 5
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 17, in <module>
    while True:
KeyboardInterrupt
```

The program goes in an infinite loop. You will have to press ctrl-c to stop.

# 159. Python - Starting a Thread

In Python, starting a thread involves using the `start()` method provided by the `Thread` class in the `threading` module. This method initiates the thread's activity and automatically calls its `run()` method in a separate thread of execution. Meaning that, when you call `start()` on each thread object (for example., `thread1`, `thread2`, `thread3`) to initiate their execution,

Python launches separate threads that concurrently execute the `run()` method defined in each `Thread` instance. The main thread continues its execution after starting the child threads.

In this tutorial, you will see a detailed explanation and example of how to use the `start()` method effectively in multi-threaded programming to understand its behavior in multi-thread applications.

## Starting a Thread in Python

The `start()` method is fundamental for beginning the execution of a thread. It sets up the thread's environment and schedules it to run. Importantly, it should only be called once per `Thread` object. If this method is called more than once on the same `Thread` object, it will raise a `RuntimeError`.

Here is the syntax for using the `start()` method on a `Thread` object –

```
threading.Thread.start()
```

### Example

Let's see the below example, that demonstrates how to start a new thread in Python using the `start()` method.

```
from threading import Thread  
  
from time import sleep  
  
  
def my_function(arg):  
    for i in range(arg):  
        print("child Thread running", i)  
        sleep(0.5)  
  
    thread = Thread(target = my_function, args = (10, ))  
    thread.start()  
  
    print("thread finished...exiting")
```

When the above code is executed, it produces the following result –

```
child Thread running 0  
child Thread running 1  
thread finished...exiting
```

```
child Thread running 2
child Thread running 3
child Thread running 4
child Thread running 5
child Thread running 6
child Thread running 7
child Thread running 8
child Thread running 9
```

**Example**

Here is another example demonstrating the working of the start() method. You can observe that, by not calling the start() method on thread2, it remains inactive and does not begin execution.

```
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter

    def run(self):
        print("Starting " + self.name)
        print_time(self.name, self.counter)

        print("Exiting " + self.name)

def print_time(threadName, counter):
    while counter:
        time.sleep(1)
        print("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# Create new threads
thread1 = MyThread(1, "Thread-1", 1)
thread2 = MyThread(2, "Thread-2", 2)
```

```
thread3 = MyThread(3, "Thread-3", 3)

# Start new Threads
thread1.start()
thread3.start()

print("Exiting Main Thread")
```

The above code will produce the following output –

```
Starting Thread-1
Starting Thread-3
Exiting Main Thread
Thread-1: Mon Jun 24 18:24:59 2024
Exiting Thread-1
Thread-3: Mon Jun 24 18:24:59 2024
Thread-3: Mon Jun 24 18:25:00 2024
Thread-3: Mon Jun 24 18:25:01 2024
Exiting Thread-3
```

# 160. Python - Joining the Threads

In Python, joining the threads means using the `join()` method to wait for one thread to finish before moving on to others. This is useful in multithreaded programming to make sure some threads are completed before starting or continuing with other threads. By using the `join()` method, you can make sure that one thread has finished running before another thread or the main program continues. In this tutorial you will get the detailed explain of the `join()` method with suitable examples.

## Joining the Threads in Python

To join the threads in Python, you can use the `Thread.join()` method from the `threading` module. Which generally is used to block the calling thread until the thread on which `join()` was called terminates. The termination may be either normal, because of an unhandled exception – or until the optional timeout occurs. You can call `join()` multiple times. However, if you try to join the current thread or attempts to join a thread before starting it with the `start()` method, will raise the `RuntimeError` exception.

Following is the syntax of the `Thread.join()` method –

```
thread.join(timeout)
```

Where, the `timeout` is an optional parameter that takes a floating-point number specifying the maximum wait time in seconds (or fractions thereof). If it is not provided or `None`, the method will block until the thread terminates.

This method always returns `None`. After calling `join()`, you can use `is_alive()` to check if the thread is still running. This is useful to determine if the `join()` call timed out.

### Example

The following example demonstrates the use of `join()` in a multithreaded program. It starts two threads (`thread1` and `thread2`). Initially, it blocks the main thread until `thread1` finishes executing the `my_function_1`. After `thread1` completes, `thread2.start()` is called, followed by `thread2.join()` to ensure that the main thread waits until `thread2` finishes executing `my_function_2()`.

```
from threading import Thread  
from time import sleep  
  
def my_function_1(arg):  
    for i in range(arg):  
        print("Child Thread 1 running", i)  
        sleep(0.5)  
  
def my_function_2(arg):  
    for i in range(arg):  
        print("Child Thread 2 running", i)
```

```

sleep(0.1)

# Create thread objects
thread1 = Thread(target=my_function_1, args=(5,))
thread2 = Thread(target=my_function_2, args=(3,))

# Start the first thread and wait for it to complete
thread1.start()
thread1.join()

# Start the second thread and wait for it to complete
thread2.start()
thread2.join()

print("Main thread finished...exiting")

```

When the above code is executed, it produces the following result –

```

Child Thread 1 running 0
Child Thread 1 running 1
Child Thread 1 running 2
Child Thread 1 running 3
Child Thread 1 running 4
Child Thread 2 running 0
Child Thread 2 running 1
Child Thread 2 running 2
Main thread finished...exiting

```

### Example

Here is another example that demonstrates how the `join()` method with a timeout allows waiting for a thread to complete for a specified period, then proceeding even if the thread hasn't finished.

```

from threading import Thread
from time import sleep

def my_function_1(arg):
    for i in range(arg):
        print("Child Thread 1 running", i)
        sleep(0.5)

```

```
def my_function_2(arg):
    for i in range(arg):
        print("Child Thread 2 running", i)
        sleep(0.1)

# Create thread objects
thread1 = Thread(target=my_function_1, args=(5,))
thread2 = Thread(target=my_function_2, args=(3,))

# Start the first thread and wait for 0.2 seconds
thread1.start()
thread1.join(timeout=0.2)

# Start the second thread and wait for it to complete
thread2.start()
thread2.join()

print("Main thread finished...exiting")
```

When you run the above code, you can see the following output –

```
Child Thread 1 running 0
Child Thread 2 running 0
Child Thread 2 running 1
Child Thread 2 running 2
Child Thread 1 running 1
Main thread finished...exiting
Child Thread 1 running 2
Child Thread 1 running 3
Child Thread 1 running 4
```

# 161. Python - Naming the Threads

In Python, naming a thread involves assigning a string as an identifier to the thread object. Thread names in Python are primarily used for identification purposes only and do not affect the thread's behavior or semantics. Multiple threads can share the same name, and names can be specified during the thread's initialization or changed dynamically.

Thread naming in Python provides a straightforward way to identify and manage threads within a concurrent program. By assigning meaningful names, users can enhance code clarity and easily debug the complex multi-threaded applications.

## Naming the Threads in Python

When you create a thread using `threading.Thread()` class, you can specify its name using the `name` parameter. If not provided, Python assigns a default name like the following pattern "Thread-N", where N is a small decimal number. Alternatively, if you specify a target function, the default name format becomes "Thread-N (target\_function\_name)".

### Example

Here is an example demonstrates assigning custom and default names to threads created using `threading.Thread()` class, and displays how names can reflect target functions.

```
from threading import Thread
import threading
from time import sleep

def my_function_1(arg):
    print("This tread name is", threading.current_thread().name)

# Create thread objects
thread1 = Thread(target=my_function_1, name='My_thread', args=(2,))
thread2 = Thread(target=my_function_1, args=(3,))

print("This tread name is", threading.current_thread().name)

# Start the first thread and wait for 0.2 seconds
thread1.start()
thread1.join()

# Start the second thread and wait for it to complete
thread2.start()
```

```
thread2.join()
```

On executing the above, it will produce the following results –

```
This tread name is MainThread
This tread name is My_thread
This tread name is Thread-1 (my_function_1)
```

## Dynamically Assigning Names to the Python Threads

You can assign or change a thread's name dynamically by directly modifying the name attribute of the thread object.

### Example

This example shows how to dynamically change thread names by modifying the name attribute of the thread object.

```
from threading import Thread
import threading
from time import sleep

def my_function_1(arg):
    threading.current_thread().name = "custom_name"
    print("This tread name is", threading.current_thread().name)

# Create thread objects
thread1 = Thread(target=my_function_1, name='My_thread', args=(2,))
thread2 = Thread(target=my_function_1, args=(3,))

print("This tread name is", threading.current_thread().name)

# Start the first thread and wait for 0.2 seconds
thread1.start()
thread1.join()

# Start the second thread and wait for it to complete
thread2.start()
thread2.join()
```

When you execute the above code, it will produce the following results –

```
This tread name is MainThread
This tread name is custom_name
```

This thread name is custom\_name

### Example

Threads can be initialized with custom names and even renamed after creation. This example demonstrates creating threads with custom names and modifying a thread's name after creation.

```
import threading

def addition_of_numbers(x, y):
    print("This Thread name is :", threading.current_thread().name)
    result = x + y

def cube_number(i):
    result = i ** 3
    print("This Thread name is :", threading.current_thread().name)

def basic_function():
    print("This Thread name is :", threading.current_thread().name)

# Create threads with custom names
t1 = threading.Thread(target=addition_of_numbers, name='My_thread', args=(2, 4))
t2 = threading.Thread(target=cube_number, args=(4,))
t3 = threading.Thread(target=basic_function)

# Start and join threads
t1.start()
t1.join()

t2.start()
t2.join()

t3.name = 'custom_name' # Assigning name after thread creation
t3.start()
t3.join()

print(threading.current_thread().name) # Print main thread's name
```

Upon execution, the above code will produce the following results –

```
This Thread name is : My_thread  
This Thread name is : Thread-1 (cube_number)  
This Thread name is : custom_name  
MainThread
```

# 162. Python - Thread Scheduling

Thread scheduling in Python is a process of deciding which thread runs at any given time. In a multi-threaded program, multiple threads are executed independently, allowing for parallel execution of tasks. However, Python does not have built-in support for controlling thread priorities or scheduling policies directly. Instead, it relies on the operating system's thread scheduler.

Python threads are mapped to native threads of the host operating system, such as POSIX threads (pthreads) on Unix-like systems or Windows threads. The operating system's scheduler manages the execution of these threads, including context switching, thread priorities, and scheduling policies. Python provides basic thread scheduling capabilities through the `threading.Timer` class and the `sched` module.

In this tutorial will learn the basics of thread scheduling in Python, including how to use the `sched` module for scheduling tasks and the `threading.Timer` class for delayed execution of functions.

## Scheduling Threads using the Timer Class

The `Timer` class of the Python `threading` module allows you to schedule a function to be called after a certain amount of time. This class is a subclass of `Thread` and serves as an example of creating custom threads.

You start a timer by calling its `start()` method, similar to `threads`. If needed, you can stop the timer before it begins by using the `cancel()` method. Note that the actual delay before the action is executed might not match the exact interval specified.

### Example

This example demonstrates how to use the `threading.Timer()` class to schedule and manage the execution of tasks (custom threads) in Python.

```
import threading  
import time  
  
# Define the event function  
def schedule_event(name, start):  
    now = time.time()  
    elapsed = int(now - start)  
    print('Elapsed:', elapsed, 'Name:', name)  
  
# Start time  
start = time.time()  
print('START:', time.ctime(start))
```

```
# Schedule events using Timer

t1 = threading.Timer(3, schedule_event, args=('EVENT_1', start))
t2 = threading.Timer(2, schedule_event, args=('EVENT_2', start))

# Start the timers
t1.start()
t2.start()

t1.join()
t2.join()

# End time
end = time.time()
print('End:', time.ctime(end))
```

On executing the above program, it will produce the following output –

```
START: Tue Jul  2 14:46:33 2024
Elapsed: 2 Name: EVENT_2
Elapsed: 3 Name: EVENT_1
End: Tue Jul  2 14:46:36 2024
```

## Scheduling Threads using the sched Module

The `sched` module in Python's standard library provides a way to schedule tasks. It implements a generic event scheduler for running tasks at specific times. It provides similar tools like task scheduler in windows or Linux.

### Key Classes and Methods of the sched Module

The `scheduler()` class is defined in the `sched` module is used to create a scheduler object. Here is the syntax of the class –

```
scheduler(timefunc=time.monotonic, delayfunc=time.sleep)
```

The methods defined in `scheduler` class include –

- **`scheduler.enter(delay, priority, action, argument=(), kwargs={})`** – Events can be scheduled to run after a delay, or at a specific time. To schedule them with a delay, `enter()` method is used.
- **`scheduler.cancel(event)`** – Remove the event from the queue. If the event is not an event currently in the queue, this method will raise a `ValueError`.
- **`scheduler.run(blocking=True)`** – Run all scheduled events.

Events can be scheduled to run after a delay, or at a specific time. To schedule them with a delay, use the `enter()` method, which takes four arguments.

- A number representing the delay
- A priority value
- The function to call
- A tuple of arguments for the function

### Example

This example demonstrates how to schedule events to run after a delay using the `sched` module. It schedules two different events –

```
import sched
import time

scheduler = sched.scheduler(time.time, time.sleep)

def schedule_event(name, start):
    now = time.time()
    elapsed = int(now - start)
    print('elapsed=', elapsed, 'name= ', name)

start = time.time()
print('START:', time.ctime(start))
scheduler.enter(2, 1, schedule_event, ('EVENT_1', start))
scheduler.enter(5, 1, schedule_event, ('EVENT_2', start))

scheduler.run()

# End time
end = time.time()
print('End:', time.ctime(end))
```

It will produce the following output –

```
START: Tue Jul  2 15:11:48 2024
elapsed= 2 name= EVENT_1
elapsed= 5 name= EVENT_2
End: Tue Jul  2 15:11:53 2024
```

### Example

Let's take another example to understand the concept better. This example schedules a function to perform an addition after a 4-second delay using the `sched` module in Python.

```
import sched
from datetime import datetime
```

```
import time

def addition(a,b):
    print("Performing Addition : ", datetime.now())
    print("Time : ", time.monotonic())
    print("Result {}+{} = ".format(a, b), a+b)

s = sched.scheduler()

print("Start Time : ", datetime.now())

event1 = s.enter(4, 1, addition, argument = (5,6))
print("Event Created : ", event1)
s.run()
print("End Time : ", datetime.now())
```

It will produce the following output –

```
Start Time : 2024-07-02 15:18:27.862524
Event Created : Event(time=2927111.05638099, priority=1, sequence=0,
action=<function addition at 0x7f31f902bd90>, argument=(5, 6), kwargs={})
Performing Addition : 2024-07-02 15:18:31.866381
Time : 2927111.060294749
Result 5+6 = 11
End Time : 2024-07-02 15:18:31.866545
```

# 163. Python - Thread Pools

A thread pool is a mechanism that automatically manages multiple threads efficiently, allowing tasks to be executed concurrently. Python does not provide thread pooling directly through the `threading` module.

Instead, it offers thread-based pooling through the `multiprocessing.dummy` module and the `concurrent.futures` module. These modules provide convenient interfaces for creating and managing thread pools, making it easier to perform concurrent task execution.

## What is a Thread Pool?

A thread pool is a collection of threads that are managed by a pool. Each thread in the pool is called a worker or a worker thread. These threads can be reused to perform multiple tasks, which reduces the burden of creating and destroying threads repeatedly.

Thread pools control the creation of threads and their life cycle, making them more efficient for handling large numbers of tasks.

We can implement thread-pools in Python using the following classes –

- Python `ThreadPool` Class
- Python `ThreadPoolExecutor` Class

## Using Python ThreadPool Class

The `multiprocessing.pool.ThreadPool` class provides a thread pool interface within the `multiprocessing` module. It manages a pool of worker threads to which jobs can be submitted for concurrent execution.

A `ThreadPool` object simplifies the management of multiple threads by handling the creation and distribution of tasks among the worker threads. It shares an interface with the `Pool` class, originally designed for processes, but has been adjusted to work with threads too.

`ThreadPool` instances are fully interface-compatible with `Pool` instances and should be managed either as a context manager or by calling `close()` and `terminate()` manually.

## Example

This example demonstrates the parallel execution of the `square` and `cube` functions on the list of numbers using the Python thread pool, where each function is applied to the numbers concurrently with up to 3 threads, each with a delay of 1 second between executions.

```
from multiprocessing.dummy import Pool as ThreadPool
import time

def square(number):
    sqr = number * number
    time.sleep(1)
```

```

print("Number:{} Square:{}".format(number, sqr))

def cube(number):
    cub = number*number*number
    time.sleep(1)
    print("Number:{} Cube:{}".format(number, cub))

numbers = [1, 2, 3, 4, 5]
pool = ThreadPool(3)
pool.map(square, numbers)
pool.map(cube, numbers)

pool.close()

```

## Output

On executing the above code you will get the following output –

```

Number:2 Square:4
Number:1 Square:1
Number:3 Square:9
Number:4 Square:16
Number:5 Square:25
Number:1 Cube:1
Number:2 Cube:8
Number:3 Cube:27
Number:4 Cube:64
Number:5 Cube:125

```

## Using Python ThreadPoolExecutor Class

The ThreadPoolExecutor class of the Python the concurrent.futures module provides a high-level interface for asynchronously executing functions using threads. The concurrent.futures module includes Future class and two Executor classes – ThreadPoolExecutor and ProcessPoolExecutor.

## The Future Class

The concurrent.futures.Future class is responsible for handling asynchronous execution of any callable such as a function. To obtain a Future object, you should call the submit() method on any Executor object. It should not be created directly by its constructor.

Important methods in the Future class are –

- **result(timeout=None):** This method returns the value returned by the call. If the call hasn't yet completed, then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a TimeoutError will be raised. If timeout is not specified, there is no limit to the wait time.
- **cancel():** This method, attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return a boolean value False. Otherwise the call will be cancelled and the method returns True.
- **cancelled():** Returns True if the call was successfully cancelled.
- **running():** Returns True if the call is currently being executed and cannot be cancelled.
- **done():** Returns True if the call was successfully cancelled or finished running.

## The ThreadPoolExecutor Class

This class represents a pool of specified number maximum worker threads to execute calls asynchronously.

```
concurrent.futures.ThreadPoolExecutor(max_threads)
```

### Example

Here is an example that uses the concurrent.futures.ThreadPoolExecutor class to manage and execute tasks asynchronously in Python. Specifically, it shows how to submit multiple tasks to a thread pool and how to check their execution status.

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def square(numbers):
    for val in numbers:
        ret = val*val
        sleep(1)
        print("Number:{} Square:{}".format(val, ret))

def cube(numbers):
    for val in numbers:
        ret = val*val*val
        sleep(1)
        print("Number:{} Cube:{}".format(val, ret))

if __name__ == '__main__':
    numbers = [1,2,3,4,5]
    executor = ThreadPoolExecutor(4)
    thread1 = executor.submit(square, (numbers))
    thread2 = executor.submit(cube, (numbers))
    print("Thread 1 executed ? :",thread1.done())
    print("Thread 2 executed ? :",thread2.done())
```

```
sleep(2)
print("Thread 1 executed ? :",thread1.done())
print("Thread 2 executed ? :",thread2.done())
```

It will produce the following output –

```
Thread 1 executed ? : False
Thread 2 executed ? : False
Number:1 Square:1
Number:1 Cube:1
Thread 1 executed ? : False
Thread 2 executed ? : False
Number:2 Square:4
Number:2 Cube:8
Number:3 Square:9
Number:3 Cube:27
Number:4 Square:16
Number:4 Cube:64
Number:5 Square:25
Number:5 Cube:125
```

# 164. Python - Main Thread

In Python, the main thread is the initial thread that starts when the Python interpreter is executed. It is the default thread within a Python process, responsible for managing the program and creating additional threads. Every Python program has at least one thread of execution called the main thread.

The main thread by default is a non-daemon thread. In this tutorial you will see the detailed explanation with relevant examples about main thread in Python programming.

## Accessing the Main Thread

The threading module in Python provides functions to access the threads. Here are the key functions –

- **threading.current\_thread()**: This function returns a `threading.Thread` instance representing the current thread.
- **threading.main\_thread()**: Returns a `threading.Thread` instance representing the main thread.

### Example

The `threading.current_thread()` function returns a `threading.Thread` instance representing the current thread. Here is an example.

```
import threading

name = 'Tutorialspoint'
print('Output:', name)
print(threading.current_thread())
```

It will produce the following output –

```
Output: Tutorialspoint
<_MainThread(MainThread, started 140260292161536)>
```

### Example

This example demonstrates how to use the `threading.main_thread()` function to get a reference to the main thread. And it also shows the difference between the main thread and other threads using `threading.current_thread()` function.

```
import threading
import time

def func(x):
    time.sleep(x)
    if not threading.current_thread() is threading.main_thread():
```

```

print('threading.current_thread() not threading.main_thread()')

t = threading.Thread(target=func, args=(0.5,))
t.start()

print(threading.main_thread())
print("Main thread finished")

```

When the above code is executed, it produces the following result –

```

<_MainThread(MainThread, started 140032182964224)>
Main thread finished
threading.current_thread() not threading.main_thread()

```

## Main Thread Behavior in Python

The main thread will exit whenever it has finished executing all the code in your script that is not started in a separate thread. For instance, when you start a new thread using `start()` method, the main thread will continue to execute the remaining code in the script until it reaches the end and then exit.

Since the other threads are started in a non-daemon mode by default, they will continue running until they are finished, even if the main thread has exited.

### Example

The following example shows the main thread behavior in a python multithreaded program.

```

import threading
import time

def func(x):
    print('Current Thread Details:',threading.current_thread())
    for n in range(x):
        print('Internal Thread Running', n)
    print('Internal Thread Finished...')

t = threading.Thread(target=func, args=(6,))
t.start()

for i in range(3):
    print('Main Thread Running',i)
print("Main Thread Finished...")

```

It will produce the following output –

```
Current Thread Details: Thread(Thread-1 (func), started 140562647860800)>
Main Thread Running 0
Internal Thread Running 0
Main Thread Running 1
Main Thread Running 2
Internal Thread Running 1
Main Thread Finished...
Internal Thread Running 2
Internal Thread Running 3
Internal Thread Running 4
Internal Thread Running 5
Internal Thread Finished...
```

*The above code can produce different outputs for different runs and different compilers.*

## Main Thread Waiting for Other Threads

To ensure that the main thread waits for all other threads to finish, you can join the threads using the `join()` method. By using the `join()` method, you can control the execution flow and ensure that the main thread properly waits for all other threads to complete their tasks before exiting. This helps in managing the lifecycle of threads in a multi-threaded Python program effectively.

### Example

This example demonstrates how to properly manage the main thread and ensure it does not exit before the worker threads have finished their tasks.

```
from threading import Thread
from time import sleep

def my_function_1():
    print("Worker 1 started")
    sleep(1)
    print("Worker 1 done")

def my_function_2(main_thread):
    print("Worker 2 waiting for Worker 1 to finish")
    main_thread.join()
    print("Worker 2 started")
```

```
sleep(1)
print("Worker 2 done")

worker1 = Thread(target=my_function_1)
worker2 = Thread(target=my_function_2, args=(worker1,))

worker1.start()
worker2.start()

for num in range(6):
    print("Main thread is still working on task", num)
    sleep(0.60)

worker1.join()
print("Main thread Completed")
```

When the above code is executed, it produces the following result –

```
Worker 1 started
Worker 2 waiting for Worker 1 to finish
Main thread is still working on task 0
Main thread is still working on task 1
Worker 1 done
Worker 2 started
Main thread is still working on task 2
Main thread is still working on task 3
Worker 2 done
Main thread is still working on task 4
Main thread is still working on task 5
Main thread Completed
```

# 165. Python - Thread Priority

In Python, currently thread priority is not directly supported by the threading module. unlike Java, Python does not support thread priorities, thread groups, or certain thread control mechanisms like destroying, stopping, suspending, resuming, or interrupting threads.

Python threads are designed simple and are loosely based on Java's threading model. This is because of Python's Global Interpreter Lock (GIL), which manages Python threads.

However, you can simulate priority-based behavior using techniques such as sleep durations, custom scheduling logic within threads or using the additional module which manages task priorities.

## Setting the Thread Priority Using Sleep()

You can simulate thread priority by introducing delays or using other mechanisms to control the execution order of threads. One common approach to simulate thread priority is by adjusting the sleep duration of your threads.

Threads with a lower priority sleep longer, and threads with a high priority sleep shorter.

### Example

Here's a simple example to demonstrate how to customize the thread priorities using the delays in Python threads. In this example, Thread-2 completes before Thread-1 because it has a lower priority value, resulting in a shorter sleep time.

```
import threading  
import time  
  
class DummyThread(threading.Thread):  
    def __init__(self, name, priority):  
        threading.Thread.__init__(self)  
        self.name = name  
        self.priority = priority  
  
    def run(self):  
        name = self.name  
        time.sleep(1.0 * self.priority)  
        print(f"{name} thread with priority {self.priority} is running")  
  
# Creating threads with different priorities  
t1 = DummyThread(name='Thread-1', priority=4)  
t2 = DummyThread(name='Thread-2', priority=1)
```

```
# Starting the threads
t1.start()
t2.start()

# Waiting for both threads to complete
t1.join()
t2.join()

print('All Threads are executed')
```

**Output**

On executing the above program, you will get the following results –

```
Thread-2 thread with priority 1 is running
Thread-1 thread with priority 4 is running
All Threads are executed
```

**Adjusting Python Thread Priority on Windows**

On Windows Operating system you can manipulate the thread priority using the `ctypes` module. This is one of the Python's standard module used for interacting with the Windows API.

**Example**

This example demonstrates how to manually set the priority of threads in Python on a Windows system using the `ctypes` module.

```
import threading
import ctypes
import time

# Constants for Windows API
w32 = ctypes.windll.kernel32
SET_THREAD = 0x20
PRIORITIZE_THE_THREAD = 1

class MyThread(threading.Thread):
    def __init__(self, start_event, name, iterations):
        super().__init__()
        self.start_event = start_event
```

```
self.thread_id = None
self.iterations = iterations
self.name = name

def set_priority(self, priority):
    if not self.is_alive():
        print('Cannot set priority for a non-active thread')
        return

    thread_handle = w32.OpenThread(SET_THREAD, False, self.thread_id)
    success = w32.SetThreadPriority(thread_handle, priority)
    w32.CloseHandle(thread_handle)
    if not success:
        print('Failed to set thread priority:', w32.GetLastError())

def run(self):
    self.thread_id = w32.GetCurrentThreadId()
    self.start_event.wait()
    while self.iterations:
        print(f'{self.name} running')
        start_time = time.time()
        while time.time() - start_time < 1:
            pass
        self.iterations -= 1

# Create an event to synchronize thread start
start_event = threading.Event()

# Create threads
thread_normal = MyThread(start_event, name='normal', iterations=4)
thread_high = MyThread(start_event, name='high', iterations=4)

# Start the threads
thread_normal.start()
thread_high.start()
```

```
# Adjusting priority of 'high' thread
thread_high.set_priority(PRIORITIZE_THE_THREAD)

# Trigger thread execution
start_event.set()
```

## Output

While executing this code in your Python interpreter, you will get the following results –

```
high running
normal running
high running
normal running
high running
normal running
high running
normal running
```

## Prioritizing Python Threads Using the Queue Module

The queue module in Python's standard library is useful in threaded programming when information must be exchanged safely between multiple threads. The Priority Queue class in this module implements all the required locking semantics.

With a priority queue, the entries are kept sorted (using the heapq module) and the lowest valued entry is retrieved first.

The Queue objects have following methods to control the Queue –

- **get()** – The get() removes and returns an item from the queue.
- **put()** – The put adds item to a queue.
- **qsize()** – The qsize() returns the number of items that are currently in the queue.
- **empty()** – The empty( ) returns True if queue is empty; otherwise, False.
- **full()** – the full() returns True if queue is full; otherwise, False.

```
queue.PriorityQueue(maxsize=0)
```

This is the Constructor for a priority queue. maxsize is an integer that sets the upper limit on the number of items that can be placed in the queue. If maxsize is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one that would be returned by min(entries)). A typical pattern for entries is a tuple in the form –

```
(priority_number, data)
```

## Example

This example demonstrates the use of the PriorityQueue class in the queue module to manage task priorities between the two threads.

```
from time import sleep
from random import random, randint
from threading import Thread
from queue import PriorityQueue

queue = PriorityQueue()

def producer(queue):
    print('Producer: Running')
    for i in range(5):

        # create item with priority
        value = random()
        priority = randint(0, 5)
        item = (priority, value)
        queue.put(item)

    # wait for all items to be processed
    queue.join()

    queue.put(None)
    print('Producer: Done')

def consumer(queue):
    print('Consumer: Running')

    while True:

        # get a unit of work
        item = queue.get()
        if item is None:
            break

        sleep(item[1])
        print(item)
        queue.task_done()

    print('Consumer: Done')
```

```
producer = Thread(target=producer, args=(queue,))
producer.start()

consumer = Thread(target=consumer, args=(queue,))
consumer.start()

producer.join()
consumer.join()
```

## Output

On execution, it will produce the following output –

```
Producer: Running
Consumer: Running
(0, 0.15332707626852804)
(2, 0.4730737391435892)
(2, 0.8679231358257962)
(3, 0.051924220435665025)
(4, 0.23945882716108446)
Producer: Done
Consumer: Done
```

# 166. Python - Daemon Threads

Daemon threads in Python are useful for running background tasks that are not critical to the program's operation. They allow you to run tasks in the background without worrying about keeping track of them.

Python provides two types of threads: non-daemon and daemon threads. By default, threads are non-daemon threads. This tutorial provides a detailed explanation with relevant examples about daemon threads in Python programming.

## Overview of Daemon Threads

Sometimes, it is necessary to execute a task in the background. A special type of thread is used for background tasks, called a daemon thread. In other words, daemon threads execute tasks in the background. These threads handle non-critical tasks that may be useful to the application but do not hamper it if they fail or are canceled while they are active and running.

Also, a daemon thread will not have control over when it is terminated. The program will terminate once all non-daemon threads finish, even if there are daemon threads still running at that point of time.

## Difference Between Daemon & Non-Daemon Threads

| Daemon                                                                                 | Non-daemon                                                            |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| A process will exit if only daemon threads are running (or if no threads are running). | A process will not exit if at least one non-daemon thread is running. |
| Daemon threads are used for background tasks.                                          | Non-daemon threads are used for critical tasks.                       |
| Daemon threads are terminated abruptly.                                                | Non-daemon threads run to completion.                                 |

Daemon threads can perform tasks such as –

- Create a file that stores Log information in the background.
- Perform web scraping in the background.
- Save the data automatically into a database in the background.

## Creating a Daemon Thread in Python

To create a daemon thread, you need to set the `daemon` property of the `Thread` constructor to `True`.

```
t1=threading.Thread(daemon=True)
```

By default, the `daemon` property is set to `None`, If you change it to not `None`, `daemon` explicitly sets whether the thread is `daemonic`.

### Example

725

Take a look at the following example to create a daemon thread and check whether the thread is using the daemon attribute.

```
import threading

from time import sleep


# function to be executed in a new thread
def run():

    # get the current thread
    thread = threading.current_thread()

    # is it a daemon thread?
    print(f'Daemon thread: {thread.daemon}')

    # Create a new thread and set it as daemon
    thread = threading.Thread(target=run, daemon=True)

    # start the thread
    thread.start()

    print('Is Main Thread is Daemon thread:', threading.current_thread().daemon)

    # Block for a short time to allow the daemon thread to run
    sleep(0.5)
```

It will produce the following output –

```
Daemon thread: True
Is Main Thread is Daemon thread: False
```

If a thread object is created in the main thread without any parameters, then the created thread will be a non-daemon thread because the main thread is not a daemon thread. Therefore, all threads created in the main thread default to non-daemon. However, we can change the daemon property to True by using the Thread.daemon attribute before starting the thread.

### **Example**

Here is an example –

```
import threading

from time import sleep


# function to be executed in a new thread
def run():
```

```

# get the current thread
thread = threading.current_thread()
# is it a daemon thread?
print(f'Daemon thread: {thread.daemon}')

# Create a new thread
thread = threading.Thread(target=run)

# Using the daemon property set the thread as daemon before starting the thread
thread.daemon = True

# start the thread
thread.start()

print('Is Main Thread is Daemon thread:', threading.current_thread().daemon)

# Block for a short time to allow the daemon thread to run
sleep(0.5)

```

On executing the above program, we will get the following output –

```

Daemon thread: True
Is Main Thread is Daemon thread: False

```

## Managing the Daemon Thread Attribute

If you attempt to set the daemon status of a thread after starting it, then a `RuntimeError` will be raised.

### Example

Here is another example that demonstrates getting the `RuntimeError` when you try to set the daemon status of a thread after starting it.

```

from time import sleep
from threading import current_thread
from threading import Thread

# function to be executed in a new thread
def run():
    # get the current thread
    thread = current_thread()

```

```
# is it a daemon thread?  
print(f'Daemon thread: {thread.daemon}')  
thread.daemon = True  
  
# create a new thread  
thread = Thread(target=run)  
  
# start the new thread  
thread.start()  
  
# block for a 0.5 sec for daemon thread to run  
sleep(0.5)
```

It will produce the following output –

```
Daemon thread: False  
Exception in thread Thread-1 (run):  
Traceback (most recent call last):  
    . . .  
    . . .  
    thread.daemon = True  
File "/usr/lib/python3.10/threading.py", line 1203, in daemon  
    raise RuntimeError("cannot set daemon status of active thread")  
RuntimeError: cannot set daemon status of active thread
```

# 167. Python - Synchronizing Threads

In Python, when multiple threads are working concurrently with shared resources, it's important to synchronize their access to maintain data integrity and program correctness. Synchronizing threads in python can be achieved using various synchronization primitives provided by the threading module, such as locks, conditions, semaphores, and barriers to control access to shared resources and coordinate the execution of multiple threads.

In this tutorial, we'll learn about various synchronization primitives provided by Python's threading module.

## Thread Synchronization using Locks

The lock object in the Python's threading module provide the simplest synchronization primitive. They allow threads to acquire and release locks around critical sections of code, ensuring that only one thread can execute the protected code at a time.

A new lock is created by calling the `Lock()` method, which returns a lock object. The lock can be acquired using the `acquire(blocking)` method, which force the threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock and released using the `release()` method.

### Example

The following example demonstrates how to use locks (the `threading.Lock()` method) to synchronize threads in Python, ensuring that multiple threads access shared resources safely and correctly.

```
import threading

counter = 10

def increment(theLock, N):
    global counter
    for i in range(N):
        theLock.acquire()
        counter += 1
        theLock.release()

lock = threading.Lock()
t1 = threading.Thread(target=increment, args=[lock, 2])
t2 = threading.Thread(target=increment, args=[lock, 10])
t3 = threading.Thread(target=increment, args=[lock, 4])
```

```
t1.start()
t2.start()
t3.start()

# Wait for all threads to complete
for thread in (t1, t2, t3):
    thread.join()

print("All threads have completed")
print("The Final Counter Value:", counter)
```

**Output**

When the above code is executed, it produces the following output –

```
All threads have completed
The Final Counter Value: 26
```

**Condition Objects for Synchronizing Python Threads**

Condition objects enable threads to wait until notified by another thread. They are useful for providing communication between the threads. The `wait()` method is used to block a thread until it is notified by another thread through `notify()` or `notify_all()`.

**Example**

This example demonstrates how Condition objects can synchronize threads using the `notify()` and `wait()` methods.

```
import threading

counter = 0

# Consumer function
def consumer(cv):
    global counter
    with cv:
        print("Consumer is waiting")
        cv.wait() # Wait until notified by increment
        print("Consumer has been notified. Current Counter value:", counter)

# increment function
def increment(cv, N):
```

```

global counter

with cv:
    print("increment is producing items")
    for i in range(1, N + 1):
        counter += i # Increment counter by i

    # Notify the consumer
    cv.notify()
    print("Increment has finished")

# Create a Condition object
cv = threading.Condition()

# Create and start threads
consumer_thread = threading.Thread(target=consumer, args=[cv])
increment_thread = threading.Thread(target=increment, args=[cv, 5])

consumer_thread.start()
increment_thread.start()

consumer_thread.join()
increment_thread.join()

print("The Final Counter Value:", counter)

```

## Output

On executing the above program, it will produce the following output –

```

Consumer is waiting
increment is producing items
Increment has finished
Consumer has been notified. Current Counter value: 15
The Final Counter Value: 15

```

## Synchronizing threads using the join() Method

The `join()` method in Python's `threading` module is used to wait until all threads have completed their execution. This is a straightforward way to synchronize the main thread with the completion of other threads.

## Example

This demonstrates synchronization of threads using the join() method to ensure that the main thread waits for all started threads to complete their work before proceeding.

```
import threading  
  
import time  
  
  
class MyThread(threading.Thread):  
    def __init__(self, threadID, name, counter):  
        threading.Thread.__init__(self)  
        self.threadID = threadID  
        self.name = name  
        self.counter = counter  
  
  
    def run(self):  
        print("Starting " + self.name)  
        print_time(self.name, self.counter, 3)  
  
  
def print_time(threadName, delay, counter):  
    while counter:  
        time.sleep(delay)  
        print("%s: %s" % (threadName, time.ctime(time.time())))  
        counter -= 1  
  
  
threads = []  
  
  
# Create new threads  
thread1 = MyThread(1, "Thread-1", 1)  
thread2 = MyThread(2, "Thread-2", 2)  
  
  
# Start the new Threads  
thread1.start()  
thread2.start()  
  
  
# Join the threads  
thread1.join()  
thread2.join()
```

```
print("Exiting Main Thread")
```

### Output

On executing the above program, it will produce the following output –

```
Starting Thread-1
Starting Thread-2
Thread-1: Mon Jul  1 16:05:14 2024
Thread-2: Mon Jul  1 16:05:15 2024
Thread-1: Mon Jul  1 16:05:15 2024
Thread-1: Mon Jul  1 16:05:16 2024
Thread-2: Mon Jul  1 16:05:17 2024
Thread-2: Mon Jul  1 16:05:19 2024
Exiting Main Thread
```

## Additional Synchronization Primitives

In addition to the above synchronization primitives, Python's threading module offers: –

- RLocks (Reentrant Locks): A variant of locks that allow a thread to acquire the same lock multiple times before releasing it, useful in recursive functions or nested function calls.
- Semaphores: Similar to locks but with a counter. Threads can acquire the semaphore up to a certain limit defined during initialization. Semaphores are useful for limiting access to resources with a fixed capacity.
- Barriers: Allows a fixed number of threads to synchronize at a barrier point and continue executing only when all threads have reached that point. Barriers are useful for coordinating a group of threads that must all complete a certain phase of execution before any of them can proceed further.

# Python Synchronization

# 168. Python - Inter-Thread Communication

Inter-Thread Communication refers to the process of enabling communication and synchronization between threads within a Python multi-threaded program.

Generally, threads in Python share the same memory space within a process, which allows them to exchange data and coordinate their activities through shared variables, objects, and specialized synchronization mechanisms provided by the `threading` module.

To facilitate inter-thread communication, the `threading` module provides various synchronization primitives like, Locks, Events, Conditions, and Semaphores objects. In this tutorial you will learn how to use the Event and Condition object for providing the communication between threads in a multi-threaded program.

## The Event Object

An Event object manages the state of an internal flag so that threads can wait or set. Event object provides methods to control the state of this flag, allowing threads to synchronize their activities based on shared conditions.

The flag is initially false and becomes true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

Following are the key methods of the Event object –

- **`is_set()`:** Return True if and only if the internal flag is true.
- **`set()`:** Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.
- **`clear()`:** Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.
- **`wait(timeout=None)`:** Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs. When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds.

## Example

The following code attempts to simulate the traffic flow being controlled by the state of traffic signal either GREEN or RED.

There are two threads in the program, targeting two different functions. The `signal_state()` function periodically sets and resets the event indicating change of signal from GREEN to RED.

The `traffic_flow()` function waits for the event to be set, and runs a loop till it remains set.

```
from threading import Event, Thread
import time

terminate = False
```

```
def signal_state():
    global terminate
    while not terminate:
        time.sleep(0.5)
        print("Traffic Police Giving GREEN Signal")
        event.set()
        time.sleep(1)
        print("Traffic Police Giving RED Signal")
        event.clear()

def traffic_flow():
    global terminate
    num = 0
    while num < 10 and not terminate:
        print("Waiting for GREEN Signal")
        event.wait()
        print("GREEN Signal ... Traffic can move")
        while event.is_set() and not terminate:
            num += 1
            print("Vehicle No:", num, " Crossing the Signal")
            time.sleep(1)
        print("RED Signal ... Traffic has to wait")

    event = Event()
    t1 = Thread(target=signal_state)
    t2 = Thread(target=traffic_flow)
    t1.start()
    t2.start()

# Terminate the threads after some time
time.sleep(5)
terminate = True

# join all threads to complete
t1.join()
t2.join()
```

```
print("Exiting Main Thread")
```

### Output

On executing the above code you will get the following output –

```
Waiting for GREEN Signal
Traffic Police Giving GREEN Signal
GREEN Signal ... Traffic can move
Vehicle No: 1 Crossing the Signal
Traffic Police Giving RED Signal
RED Signal ... Traffic has to wait
Waiting for GREEN Signal
Traffic Police Giving GREEN Signal
GREEN Signal ... Traffic can move
Vehicle No: 2 Crossing the Signal
Vehicle No: 3 Crossing the Signal
Traffic Police Giving RED Signal
Traffic Police Giving GREEN Signal
Vehicle No: 4 Crossing the Signal
Traffic Police Giving RED Signal
RED Signal ... Traffic has to wait
Traffic Police Giving GREEN Signal
Traffic Police Giving RED Signal
Exiting Main Thread
```

## The Condition Object

The Condition object in Python's threading module provides a more advanced synchronization mechanism. It allows threads to wait for a notification from another thread before proceeding. The Condition object are always associated with a lock and provide mechanisms for signaling between threads.

Following is the syntax of the `threading.Condition()` class –

```
threading.Condition(lock=None)
```

Below are the key methods of the Condition object –

- **`acquire(*args)`:** Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.
- **`release()`:** Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.
- **`wait(timeout=None)`:** This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable

in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

- **wait\_for(predicate, timeout=None):** This utility method may call wait() repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to False if the method timed out.
- **notify(n=1):** This method wakes up at most n of the threads waiting for the condition variable; it is a no-op if no threads are waiting.
- **notify\_all():** Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

### Example

This example demonstrates a simple form of inter-thread communication using the Condition object of the Python's threading module. Here thread\_a and thread\_b are communicated using a Condition object, the thread\_a waits until it receives a notification from thread\_b. the thread\_b sleeps for 2 seconds before notifying thread\_a and then finishes.

```
from threading import Condition, Thread
import time

c = Condition()

def thread_a():
    print("Thread A started")
    with c:
        print("Thread A waiting for permission...")
        c.wait()
        print("Thread A got permission!")
    print("Thread A finished")

def thread_b():
    print("Thread B started")
    with c:
        time.sleep(2)
        print("Notifying Thread A...")
        c.notify()
    print("Thread B finished")

Thread(target=thread_a).start()
Thread(target=thread_b).start()
```

## Output

On executing the above code you will get the following output –

```
Thread A started
Thread A waiting for permission...
Thread B started
Notifying Thread A...
Thread B finished
Thread A got permission!
Thread A finished
```

## Example

Here is another code demonstrating how the Condition object is used for providing the communication between threads. The thread t2 runs the taskB() function, and the thread t1 runs the taskA() function. The t1 thread acquires the condition and notifies it.

By that time, the t2 thread is in a waiting state. After the condition is released, the waiting thread proceeds to consume the random number generated by the notifying function.

```
from threading import Condition, Thread
import time
import random

numbers = []

def taskA(c):
    for _ in range(5):
        with c:
            num = random.randint(1, 10)
            print("Generated random number:", num)
            numbers.append(num)
            print("Notification issued")
            c.notify()
            time.sleep(0.3)

def taskB(c):
    for i in range(5):
        with c:
            print("waiting for update")
            while not numbers:
```

```
c.wait()  
    print("Obtained random number", numbers.pop())  
    time.sleep(0.3)  
  
c = Condition()  
t1 = Thread(target=taskB, args=(c,))  
t2 = Thread(target=taskA, args=(c,))  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
print("Done")
```

When you execute this code, it will produce the following output –

```
waiting for update  
Generated random number: 2  
Notification issued  
Obtained random number 2  
Generated random number: 5  
Notification issued  
waiting for update  
Obtained random number 5  
Generated random number: 1  
Notification issued  
waiting for update  
Obtained random number 1  
Generated random number: 9  
Notification issued  
waiting for update  
Obtained random number 9  
Generated random number: 2  
Notification issued  
waiting for update  
Obtained random number 2  
Done
```

# 169. Python - Thread Deadlock

A deadlock may be described as a concurrency failure mode. It is a situation in a program where one or more threads wait for a condition that never occurs. As a result, the threads are unable to progress and the program is stuck or frozen and must be terminated manually.

Deadlock situation may arise in many ways in your concurrent program. Deadlocks are not developed intentionally, instead, they are a side effect or bug in the code.

Common causes of thread deadlocks are listed below –

- A thread that attempts to acquire the same mutex lock twice.
- Threads that wait on each other (e.g. A waits on B, B waits on A).
- When a thread that fails to release a resource such as lock, semaphore, condition, event, etc.
- Threads that acquire mutex locks in different orders (e.g. fail to perform lock ordering).

## How to Avoid Deadlocks in Python Threads?

When multiple threads in a multi-threaded application attempt to access the same resource, such as performing read/write operations on the same file, it can lead to data inconsistency. Therefore, it is important to synchronize concurrent access to resources by using locking mechanisms.

The Python threading module provides a simple-to-implement locking mechanism to synchronize threads. You can create a new lock object by calling the Lock() class, which initializes the lock in an unlocked state.

## Locking Mechanism with the Lock Object

An object of the Lock class has two possible states – locked or unlocked, initially in unlocked state when first created. A lock doesn't belong to any particular thread.

The Lock class defines acquire() and release() methods.

### The acquire() Method

The acquire() method of the Lock class changes the lock's state from unlocked to locked. It returns immediately unless the optional blocking argument is set to True, in which case it waits until the lock is acquired.

Here is the Syntax of this method –

```
Lock.acquire(blocking, timeout)
```

Where,

- **blocking** – If blocking is set to false, the thread will not be blocked. To block a thread, set the variable to true.
- **timeout** – Specifies a timeout period for acquiring the lock.

The return value of this method is True if the lock is acquired successfully; False if not.

## The release() Method

When the state is locked, this method in another thread changes it to unlocked. This can be called from any thread, not only the thread which has acquired the lock

Following is the Syntax of the release() method –

```
Lock.release()
```

The release() method should only be called in the locked state. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked and waiting for the lock to become unlocked, it allows exactly one of them to proceed. There is no return value of this method.

### Example

In the following program, two threads try to call the synchronized() method. One of them acquires the lock and gains the access while the other waits. When the run() method is completed for the first thread, the lock is released and the synchronized method is available for second thread.

When both the threads join, the program comes to an end.

```
from threading import Thread, Lock
import time

lock=Lock()
threads=[]

class myThread(Thread):
    def __init__(self,name):
        Thread.__init__(self)
        self.name=name
    def run(self):
        lock.acquire()
        synchronized(self.name)
        lock.release()

def synchronized(threadName):
    print ("{} has acquired lock and is running synchronized
method".format(threadName))
    counter=5
    while counter:
        print ('**', end='')
        time.sleep(2)
```

```

        counter=counter-1

        print('\nlock released for', threadName)

t1=myThread('Thread1')
t2=myThread('Thread2')

t1.start()
threads.append(t1)

t2.start()
threads.append(t2)

for t in threads:
    t.join()

print ("end of main thread")

```

It will produce the following output –

```

Thread1 has acquired lock and is running synchronized method
*****
lock released for Thread1

Thread2 has acquired lock and is running synchronized method
*****
lock released for Thread2

end of main thread

```

## Semaphore Object for Synchronization

In addition to locks, Python threading module supports semaphores, which offering another synchronization technique. It is one of the oldest synchronization techniques invented by a well-known computer scientist, Edsger W. Dijkstra.

The basic concept of semaphore is to use an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

The Semaphore class in threading module defines acquire() and release() methods.

### The acquire() Method

If the internal counter is larger than zero on entry, decrement it by one and return True immediately.

If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return True. Exactly one thread will be awoken by each call to `release()`. The order in which threads awake is arbitrary.

If blocking parameter is set to False, do not block. If a call without an argument would block, return False immediately; otherwise, do the same thing as when called without arguments, and return True.

### The `release()` Method

Release a semaphore, incrementing the internal counter by 1. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up n of those threads.

#### Example

This example demonstrates how to use a `Semaphore` object in Python to control access to a shared resource among multiple threads, for avoiding deadlock in Python's multi-threaded program.

```
from threading import *
import time

# creating thread instance where count = 3
lock = Semaphore(4)

# creating instance
def synchronized(name):

    # calling acquire method
    lock.acquire()

    for n in range(3):
        print('Hello! ', end = '')
        time.sleep(1)
        print( name)

    # calling release method
    lock.release()

# creating multiple thread
thread_1 = Thread(target = synchronized , args = ('Thread 1',))
thread_2 = Thread(target = synchronized , args = ('Thread 2',))
```

```
thread_3 = Thread(target = synchronized , args = ('Thread 3',))

# calling the threads
thread_1.start()
thread_2.start()
thread_3.start()
```

It will produce the following output –

```
Hello! Hello! Hello! Thread 1
Hello! Thread 2
Thread 3
Hello! Hello! Thread 1
Hello! Thread 3
Thread 2
Hello! Hello! Thread 1
Thread 3
Thread 2
```

# 170. Python - Interrupting a Thread

Interrupting a thread in Python is a common requirement in multi-threaded programming, where a thread's execution needs to be terminated under certain conditions. In a multi-threaded program, a task in a new thread, may be required to be stopped. This may be for many reasons, such as – task completion, application shutdown, or other external conditions.

In Python, interrupting threads can be achieved using `threading.Event` or by setting a termination flag within the thread itself. These methods allow you to interrupt the threads effectively, ensuring that resources are properly released and threads exit cleanly.

## Thread Interruption using Event Object

One of the straightforward ways to interrupt a thread is by using the `threading.Event` class. This class allows one thread to signal to another that a particular event has occurred. Here's how you can implement thread interruption using `threading.Event`.

### Example

In this example, we have a `MyThread` class. Its object starts executing the `run()` method. The main thread sleeps for a certain period and then sets an event. Till the event is detected, loop in the `run()` method continues. As soon as the event is detected, the loop terminates.

```
from time import sleep
from threading import Thread
from threading import Event

class MyThread(Thread):
    def __init__(self, event):
        super(MyThread, self).__init__()
        self.event = event

    def run(self):
        i=0
        while True:
            i+=1
            print ('Child thread running...',i)
            sleep(0.5)
            if self.event.is_set():
                break
            print()
```

```

print('Child Thread Interrupted')

event = Event()
thread1 = MyThread(event)
thread1.start()

sleep(3)
print('Main thread stopping child thread')
event.set()
thread1.join()

```

When you execute this code, it will produce the following output –

```

Child thread running... 1
Child thread running... 2
Child thread running... 3
Child thread running... 4
Child thread running... 5
Child thread running... 6
Main thread stopping child thread
Child Thread Interrupted

```

## Thread Interruption using a Flag

Another approach to interrupting threads is by using a flag that the thread checks at regular intervals. This method involves setting a flag attribute in the thread object and regularly checking its value in the thread's execution loop.

### Example

This example demonstrates how to use a flag to control and stop a running thread in Python multithreaded program.

```

import threading
import time

def foo():
    t = threading.current_thread()
    while getattr(t, "do_run", True):
        print("working on a task")
        time.sleep(1)
    print("Stopping the Thread after some time.")

```

```
# Create a thread
t = threading.Thread(target=foo)
t.start()

# Allow the thread to run for 5 seconds
time.sleep(5)

# Set the termination flag to stop the thread
t.do_run = False
```

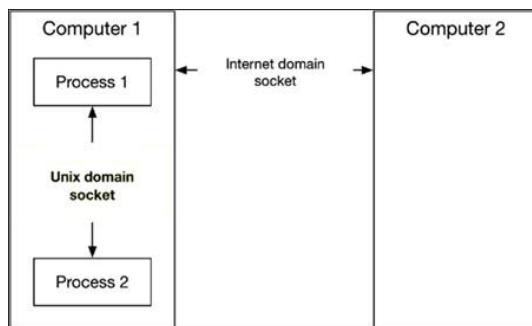
When you execute this code, it will produce the following output –

```
working on a task
Stopping the Thread after some time.
```

# Python Networking

# 171. Python - Network Programming

The threading module in Python's standard library is capable of handling multiple threads and their interaction within a single process. Communication between two processes running on the same machine is handled by Unix domain sockets, whereas for the processes running on different machines connected with TCP (Transmission control protocol), Internet domain sockets are used.



Python's standard library consists of various built-in modules that support interprocess communication and networking. Python provides two levels of access to the network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

| Protocol | Common function    | Port No | Python module              |
|----------|--------------------|---------|----------------------------|
| HTTP     | Web pages          | 80      | httplib, urllib, xmlrpclib |
| NNTP     | Usenet news        | 119     | nntplib                    |
| FTP      | File transfers     | 20      | ftplib, urllib             |
| SMTP     | Sending email      | 25      | smtplib                    |
| POP3     | Fetching email     | 110     | poplib                     |
| IMAP4    | Fetching email     | 143     | imaplib                    |
| Telnet   | Command lines      | 23      | telnetlib                  |
| Gopher   | Document transfers | 70      | gopherlib, urllib          |

# 172. Python - Socket Programming

## Python Socket Programming

Socket programming is a technique in which we communicate between two nodes connected in a network where the server node listens to the incoming requests from the client nodes.

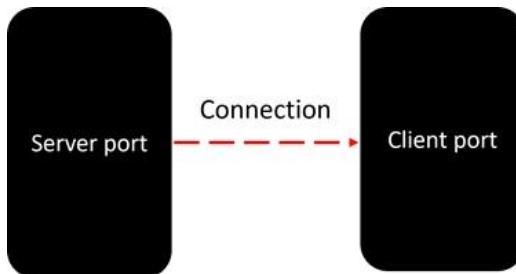
In Python, the socket module is used for socket programming. The socket module in the standard library included functionality required for communication between server and client at hardware level.

This module provides access to the BSD socket interface. It is available on all operating systems such as Linux, Windows, MacOS.

### What are Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different machines.

A socket is identified by the combination of IP address and the port number. It should be properly configured at both ends to begin communication.



Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

The term socket programming implies programmatically setting up sockets to be able to send and receive data.

There are two types of communication protocols –

- connection-oriented protocol
- connection-less protocol

TCP or Transmission Control Protocol is a connection-oriented protocol. The data is transmitted in packets by the server, and assembled in the same order of transmission by the receiver. Since the sockets at either end of the communication need to be set before starting, this method is more reliable.

UDP or User Datagram Protocol is connectionless. The method is not reliable because the sockets does not require establishing any connection and termination process for transferring the data.

## Python socket Module

The socket module is used for creating and managing socket programming for the connected nodes in a network. The socket module provides a socket class. You need to create a socket using the `socket.socket()` constructor.

An object of the socket class represents the pair of host name and the port numbers.

### Syntax

The following is the syntax of `socket.socket()` constructor –

```
socket.socket (socket_family, socket_type, protocol=0)
```

### Parameters

- **family** – `AF_INET` by default. Other values - `AF_INET6` (eight groups of four hexadecimal digits), `AF_UNIX`, `AF_CAN` (Controller Area Network) or `AF_RDS` (Reliable Datagram Sockets).
- **socket\_type** – should be `SOCK_STREAM` (the default), `SOCK_DGRAM`, `SOCK_RAW` or perhaps one of the other `SOCK_` constants.
- **protocol** – number is usually zero and may be omitted.

### Return Type

This method returns a socket object.

Once you have the socket object, then you can use the required methods to create your client or server program.

## Server Socket Methods

The socket instantiated on server is called server socket. Following methods are available to the socket object on the server –

- **bind() method** – This method binds the socket to specified IP address and port number.
- **listen() method** – This method starts server and runs into a listen loop looking for connection request from client.
- **accept() method** – When connection request is intercepted by server, this method accepts it and identifies the client socket with its address.

To create a socket on a server, use the following snippet –

```
import socket
server = socket.socket()
server.bind(('localhost',12345))
server.listen()
client, addr = server.accept()
print ("connection request from: " + str(addr))
```

By default, the server is bound to local machine's IP address 'localhost' listening at arbitrary empty port number.

## Client Socket Methods

Similar socket is set up on the client end. It mainly sends connection request to server socket listening at its IP address and port number

### **connect() method**

This method takes a two-item tuple object as argument. The two items are IP address and port number of the server.

```
obj=socket.socket()
obj.connect((host,port))
```

Once the connection is accepted by the server, both the socket objects can send and/or receive data.

### **send() method**

The server sends data to client by using the address it has intercepted.

```
client.send(bytes)
```

Client socket sends data to socket it has established connection with.

### **sendall() method**

Similar to send(). However, unlike send(), this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success.

### **sendto() method**

This method is to be used in case of UDP protocol only.

### **recv() method**

This method is used to retrieve data sent to the client. In case of server, it uses the remote socket whose request has been accepted.

```
client.recv(bytes)
```

### **recvfrom() method**

This method is used in case of UDP protocol.

## Python - Socket Server

To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call the bind(hostname, port) function to specify a port for your service on the given host.

Next, call the `accept` method of the returned object. This method waits until a client connects to the port you specified, and then returns a connection object that represents the connection to that client.

### **Example of Server Socket**

```
import socket
host = "127.0.0.1"
port = 5001
server = socket.socket()
server.bind((host, port))
server.listen()
conn, addr = server.accept()
print ("Connection from: " + str(addr))
while True:
    data = conn.recv(1024).decode()
    if not data:
        break
    data = str(data).upper()
    print (" from client: " + str(data))
    data = input("type message: ")
    conn.send(data.encode())
conn.close()
```

### **Python - Socket Client**

Let us write a very simple client program, which opens a connection to a given port 5001 and a given localhost. It is very simple to create a socket client using the Python's `socket` module function.

The `socket.connect(hostname, port)` opens a TCP connection to hostname on the port. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

### **Example of Client Socket**

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits when 'q' is entered.

```
import socket
host = '127.0.0.1'
port = 5001
obj = socket.socket()
obj.connect((host, port))
message = input("type message: ")
```

```

while message != 'q':
    obj.send(message.encode())
    data = obj.recv(1024).decode()
    print ('Received from server: ' + data)
    message = input("type message: ")
obj.close()

```

- Run Server code first. It starts listening.
- Then start client code. It sends request.
- Request accepted. Client address identified
- Type some text and press Enter.
- Data received is printed. Send data to client.
- Data from server is received.
- Loop terminates when 'q' is input.

Server-client interaction is shown below –

```

D:\socketsvr>python socketsvr.py
Connection from: ('127.0.0.1', 52324)
from client: HELLO PYTHON
type message: Thank you
from client: GOOD BYE
type message: See you later

D:\socketsvr>

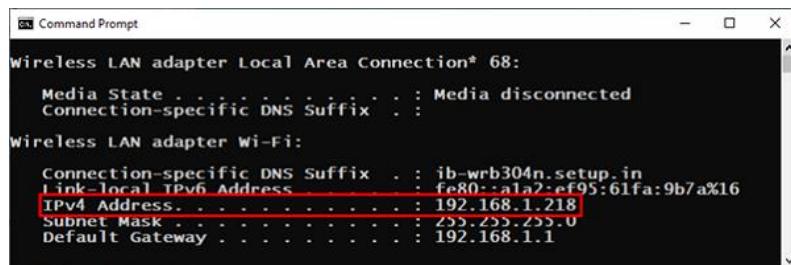
D:\socketsvr>python socketclient.py
type message: Hello Python
Received from server: Thank you
type message: Good Bye
Received from server: See you later
type message: q

D:\socketsvr>

```

We have implemented client-server communication with socket module on the local machine. To put server and client codes on two different machines on a network, we need to find the IP address of the server machine.

On Windows, you can find the IP address by running ipconfig command. The ifconfig command is the equivalent command on Ubuntu.



Change host string in both the server and client codes with IPv4 Address value and run them as before.

## Python File Transfer with Socket Module

The following program demonstrates how socket communication can be used to transfer a file from server to the client

### Server Code

The code for establishing connection is same as before. After the connection request is accepted, a file on server is opened in binary mode for reading, and bytes are successively read and sent to the client stream till end of file is reached.

```
import socket
host = "127.0.0.1"
port = 5001
server = socket.socket()
server.bind((host, port))
server.listen()
conn, addr = server.accept()
data = conn.recv(1024).decode()
filename='test.txt'
f = open(filename,'rb')
while True:
    l = f.read(1024)
    if not l:
        break
    conn.send(l)
    print('Sent ',repr(l))
f.close()
print('File transferred')
conn.close()
```

### Client Code

On the client side, a new file is opened in wb mode. The stream of data received from server is written to the file. As the stream ends, the output file is closed. A new file will be created on the client machine.

```
import socket

s = socket.socket()
host = "127.0.0.1"
port = 5001
```

```

s.connect((host, port))

s.send("Hello server!".encode())


with open('recv.txt', 'wb') as f:

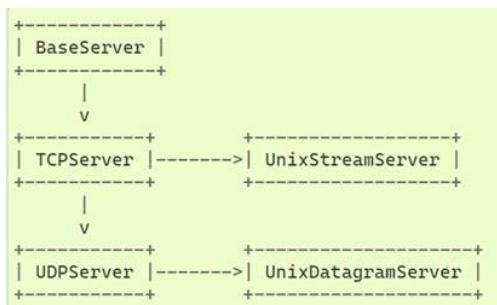
    while True:
        print('receiving data...')
        data = s.recv(1024)
        if not data:
            break
        f.write(data)

f.close()
print('Successfully received')
s.close()
print('connection closed')

```

## The Python socketserver Module

The socketserver module in Python's standard library is a framework for simplifying task of writing network servers. There are following classes in module, which represent synchronous servers –



These classes work with corresponding RequestHandler classes for implementing the service. BaseServer is the superclass of all Server objects in the module.

TCPServer class uses the internet TCP protocol, to provide continuous streams of data between the client and server. The constructor automatically attempts to invoke `server_bind()` and `server_activate()`. The other parameters are passed to the BaseServer base class.

You must also create a subclass of StreamRequestHandler class. It provides `self.rfile` and `self.wfile` attributes to read or write to get the request data or return data to the client.

- UDPServer and DatagramRequestHandler – These classes are meant to be used for UDP protocol.
- DatagramRequestHandler and UnixDatagramServer – These classes use Unix domain sockets; they're not available on non-Unix platforms.

## Server Code

You must write a RequestHandler class. It is instantiated once per connection to the server, and must override the handle() method to implement communication to the client.

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):

    def handle(self):
        self.data = self.request.recv(1024).strip()
        host, port = self.client_address
        print("{}:{} wrote:".format(host, port))
        print(self.data.decode())
        msg = input("enter text .. ")
        self.request.sendall(msg.encode())
```

On the server's assigned port number, an object of TCPServer class calls the forever() method to put the server in the listening mode and accepts incoming requests from clients.

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        server.serve_forever()
```

## Client Code

When working with socketserver, the client code is more or less similar with the socket client application.

```
import socket
import sys

HOST, PORT = "localhost", 9999

while True:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        # Connect to server and send data
        sock.connect((HOST, PORT))
        data = input("enter text .. .")
        sock.sendall(bytes(data + "\n", "utf-8"))

        # Receive data from the server and shut down
        received = str(sock.recv(1024), "utf-8")
```

```

print("Sent: {}".format(data))
print("Received: {}".format(received))

```

Run the server code in one command prompt terminal. Open multiple terminals for client instances. You can simulate a concurrent communication between the server and more than one clients.

| <b>Server</b>                                                                                                                                                                                                                                                                                                                                                                                | <b>Client-1</b>                                                                                                                                                                                                                                                                       | <b>Client-2</b>                                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| D:\socketsrvr>python<br>myserver.py<br>127.0.0.1:54518 wrote:<br>from client-1<br>enter text ..<br>hello<br>127.0.0.1:54522 wrote:<br>how are you<br>enter text ..<br>fine<br>127.0.0.1:54523 wrote:<br>from client-2<br>enter text ..<br>hi client-2<br>127.0.0.1:54526 wrote:<br>good bye<br>enter text ..<br>bye bye<br>127.0.0.1:54530 wrote:<br>thanks<br>enter text ..<br>bye client-2 | D:\socketsrvr>python<br>myclient.py<br>enter text ...<br>from client-1<br>Sent:<br>from client-1<br>Received: hello<br>enter text ...<br>how are you<br>Sent:<br>how are you<br>Received: fine<br>enter text ...<br>good bye<br>Sent: good bye<br>Received: bye bye<br>enter text ... | D:\socketsrvr>python<br>myclient.py<br>enter text ...<br>from client-2<br>Sent:<br>from client-2<br>Received: hi client-2<br>enter text ...<br>thanks<br>Sent: thanks<br>Received:<br>bye client-2<br>enter text ... |

# 173. Python - URL Processing

In the world of Internet, different resources are identified by URLs (Uniform Resource Locators). Python's standard library includes the `urllib` package, which has modules for working with URLs. It helps you parse URLs, fetch web content, and manage errors.

This tutorial introduces `urllib` basics to help you start using it. Improve your skills in web scraping, fetching data, and managing URLs with Python using `urllib`.

- The `urllib` package contains the following modules for processing URLs –
- `urllib.parse` module is used for parsing a URL into its parts.
- `urllib.request` module contains functions for opening and reading URLs
- `urllib.error` module carries definitions of the exceptions raised by `urllib.request`
- `urllib.robotparser` module parses the robots.txt files

## The `urllib.parse` Module

This module serves as a standard interface to obtain various parts from a URL string. The module contains following functions –

### `urlparse(urlstring)`

Parse a URL into six components, returning a 6-item named tuple. Each tuple item is a string corresponding to following attributes –

| Attribute | Index | Value                              |
|-----------|-------|------------------------------------|
| scheme    | 0     | URL scheme specifier               |
| netloc    | 1     | Network location part              |
| path      | 2     | Hierarchical path                  |
| params    | 3     | Parameters for last path element   |
| query     | 4     | Query component                    |
| fragment  | 5     | Fragment identifier                |
| username  |       | User name                          |
| password  |       | Password                           |
| hostname  |       | Host name (lower case)             |
| Port      |       | Port number as integer, if present |

### Example

```
from urllib.parse import urlparse
url = "https://example.com/employees/name/?salary>=25000"
parsed_url = urlparse(url)
print (type(parsed_url))
```

```

print ("Scheme:",parsed_url.scheme)
print ("netloc:", parsed_url.netloc)
print ("path:", parsed_url.path)
print ("params:", parsed_url.params)
print ("Query string:", parsed_url.query)
print ("Frgment:", parsed_url.fragment)

```

It will produce the following output –

```

<class 'urllib.parse.ParseResult'>
Scheme: https
netloc: example.com
path: /employees/name/
params:
Query string: salary>=25000
Frgment:

```

### **parse\_qs(qs))**

This function Parse a query string given as a string argument. Data is returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

To further fetch the query parameters from the query string into a dictionary, use `parse_qs()` function of the `query` attribute of `ParseResult` object as follows –

#### **Example**

```

from urllib.parse import urlparse, parse_qs
url = "https://example.com/employees?name=Anand&salary=25000"
parsed_url = urlparse(url)
dct = parse_qs(parsed_url.query)
print ("Query parameters:", dct)

```

It will produce the following output –

```
Query parameters: {'name': ['Anand'], 'salary': ['25000']}
```

### **urlsplit(urlstring)**

This is similar to `urlparse()`, but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the path portion of the URL is needed.

### **urlunparse(parts)**

This function is the opposite of urlparse() function. It constructs a URL from a tuple as returned by urlparse(). The parts argument can be any six-item iterable. This returns an equivalent URL.

### Example

```
from urllib.parse import urlunparse

lst = ['https', 'example.com', '/employees/name/', '', 'salary>=25000', '']
new_url = urlunparse(lst)
print ("URL:", new_url)
```

It will produce the following output –

```
URL: https://example.com/employees/name/?salary>=25000
```

### urlunsplit(parts)

Combine the elements of a tuple as returned by urlsplit() into a complete URL as a string. The parts argument can be any five-item iterable.

## The urllib.request Module

This module offers the functions and classes for handling the URL's opening and reading operations by using the urlopen() function.

### urlopen() function

This function opens the given URL, which can be either a string or a Request object. The optional timeout parameter specifies a timeout in seconds for blocking operations. This actually only works for HTTP, HTTPS and FTP connections.

This function always returns an object which can work as a context manager and has the properties url, headers, and status. For HTTP and HTTPS URLs, this function returns a http.client.HTTPResponse object slightly modified.

### Example

The following code uses urlopen() function to read the binary data from an image file, and writes it to local file. You can open the image file on your computer using any image viewer.

```
from urllib.request import urlopen
obj = urlopen("https://www.tutorialspoint.com/images/logo.png")
data = obj.read()
img = open("img.jpg", "wb")
img.write(data)
img.close()
```

It will produce the following output –



## The Request Object

The `urllib.request` module includes `Request` class. This class is an abstraction of a URL request. The constructor requires a mandatory string argument a valid URL.

### Syntax

```
urllib.request.Request(url, data, headers, origin_req_host, method=None)
```

### Parameters

`url` – A string that is a valid URL

- **data** – An object specifying additional data to send to the server. This parameter can only be used with HTTP requests. Data may be bytes, file-like objects, and iterables of bytes-like objects.
- **headers** – Should be a dictionary of headers and their associated values.
- **origin\_req\_host** – Should be the request-host of the origin transaction
- **method** – should be a string that indicates the HTTP request method. One of GET, POST, PUT, DELETE and other HTTP verbs. Default is GET.

### Example

```
from urllib.request import Request
obj = Request("https://www.tutorialspoint.com/")
```

This Request object can now be used as an argument to `urlopen()` method.

```
from urllib.request import Request, urlopen
obj = Request("https://www.tutorialspoint.com/")
resp = urlopen(obj)
```

The `urlopen()` function returns a `HttpServletResponse` object. Calling its `read()` method fetches the resource at the given URL.

```
from urllib.request import Request, urlopen
obj = Request("https://www.tutorialspoint.com/")
resp = urlopen(obj)
data = resp.read()
print (data)
```

## Sending Data

If you define `data` argument to the `Request` constructor, a POST request will be sent to the server. The data should be any object represented in bytes.

```
from urllib.request import Request, urlopen
from urllib.parse import urlencode
```

```

values = {'name': 'Madhu',
          'location': 'India',
          'language': 'Hindi' }
data = urlencode(values).encode('utf-8')
obj = Request("https://example.com", data)

```

## Sending Headers

The Request constructor also accepts header argument to push header information into the request. It should be in a dictionary object.

```

headers = {'User-Agent': user_agent}
obj = Request("https://example.com", data, headers)

```

## The urllib.error Module

Following exceptions are defined in urllib.error module –

### URLError

URLError is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist. In this case, the exception raised will have a 'reason' attribute.

### Example

```

from urllib.request import Request, urlopen
import urllib.error as err

obj = Request("http://www.nosuchserver.com")
try:
    urlopen(obj)
except err.URLError as e:
    print(e)

```

It will produce the following output –

```
HTTP Error 403: Forbidden
```

### HTTPError

Every time the server sends a HTTP response it is associated with a numeric "status code". It code indicates why the server is unable to fulfill the request. The default handlers will handle some of these responses for you. For those it can't handle, urlopen() function raises an HTTPError. Typical examples of HTTPErrors are '404' (page not found), '403' (request forbidden), and '401' (authentication required).

**Example**

```
from urllib.request import Request, urlopen  
import urllib.error as err  
  
obj = Request("http://www.python.org/fish.html")  
try:  
    urlopen(obj)  
except err.HTTPError as e:  
    print(e.code)
```

It will produce the following output –

```
404
```

# 174. Python - Generics

In Python, generics is a mechanism with which you can define functions, classes, or methods that can operate on multiple types while maintaining type safety. With the implementation of Generics, it is possible to write reusable code that can be used with different data types. It ensures promoting code flexibility and type correctness.

Normally, in Python programming, you don't need to declare a variable type. The type is determined dynamically by the value assigned to it. Python's interpreter doesn't perform type checks and hence it may raise runtime exceptions.

Python introduced generics with type hints in version 3.5, allowing you to specify the expected types of variables, function arguments, and return values. This feature helps in reducing runtime errors and improving code readability.

Generics extend the concept of type hints by introducing type variables, which represent generic types that can be replaced with specific types when using the generic function or class.

## Defining a Generic Function

Let us have a look at the following example that defines a generic function –

```
from typing import List, TypeVar, Generic
T = TypeVar('T')
def reverse(items: List[T]) -> List[T]:
    return items[::-1]
```

Here, we define a generic function called 'reverse'. The function takes a list ('List[T]') as an argument and returns a list of the same type. The type variable 'T' represents the generic type, which will be replaced with a specific type when the function is used.

## Calling the Generic Function with Different Data Types

The function reverse() function is called with different data types –

```
numbers = [1, 2, 3, 4, 5]
reversed_numbers = reverse(numbers)
print(reversed_numbers)

fruits = ['apple', 'banana', 'cherry']
reversed_fruits = reverse(fruits)
print(reversed_fruits)
```

It will produce the following output –

```
[5, 4, 3, 2, 1]
```

```
['cherry', 'banana', 'apple']
```

## Defining a Generic Class

A generic type is typically declared by adding a list of type parameters after the class name. The following example uses generics with a generic class –

```
from typing import List, TypeVar, Generic
T = TypeVar('T')
class Box(Generic[T]):
    def __init__(self, item: T):
        self.item = item
    def get_item(self) -> T:
        return self.item
Let us create objects of the above generic class with int and str type
box1 = Box(42)
print(box1.get_item())

box2 = Box('Hello')
print(box2.get_item())
```

It will produce the following output –

```
42
Hello
```

# Python Libraries

# 175. NumPy Tutorial

**NumPy**, which stands for **Numerical Python**, is an open-source Python library consisting of multidimensional and single-dimensional array elements. It's a standard that computes numerical data in Python. NumPy is most widely used in almost every domain where numerical computation is required, like science and engineering; hence, the NumPy API functionalities are highly utilized in data science and scientific Python packages, including Pandas, SciPy, Matplotlib, scikit-learn, scikit-image, and many more.

This NumPy tutorial explains the basics of NumPy, such as its architecture and environment. It also discusses array functions, types of indexing, etc., and then extends to learn Matplotlib, Pandas, SciPy, and other important Python libraries. All this is explained with the help of examples for better understanding.

## Why NumPy - Need of NumPy

NumPy is a fundamental package for numerical computation in Python. It provides mathematical functions to compute data as well as functions to operate multi-dimensional arrays and matrices efficiently. Here are some reasons why NumPy is essential:

- NumPy includes a wide range of mathematical functions for basic arithmetic, linear algebra, Fourier analysis, and more.
- NumPy performs numerical operations on large datasets efficiently.
- NumPy supports multi-dimensional arrays, allowing for the representation of complex data structures such as images, sound waves, and tensors in machine learning models.
- It supports the writing of concise and readable code for complex mathematical computations.
- NumPy integrates with other libraries to do scientific computation; these are SciPy (for scientific computing), Pandas (for data manipulation and analysis), and scikit-learn (for machine learning).
- Many scientific and numerical computing libraries and tools are built on top of NumPy.
- Its widespread adoption and stability make it a standard choice for numerical computing tasks.

Overall, NumPy plays a crucial role in the Python ecosystem for scientific computing, data analysis, machine learning, and more. Its efficient array operations and extensive mathematical functions make it an indispensable tool for working with numerical data in Python.

## NumPy Applications - Uses of NumPy

The NumPy API in Python is used primarily for numerical computing. It provides support for a wide range of mathematical functions to operate on data efficiently. The following are some common application areas where NumPy is extensively used:

- **Data Analysis:** NumPy offers rapid and effective array operations, rendering it well-suited for tasks like data cleansing, filtering, and transformation. It is predominantly used in the analysis and scientific handling of data, particularly when working with extensive, large datasets.

- **Machine Learning and Artificial Intelligence:** Different machine learning and deep learning frameworks in Python, such as TensorFlow, and PyTorch, rely on NumPy arrays for handling input data, model parameters, and outputs.
- **Scientific Computing:** NumPy is widely used in scientific computing applications such as physics, chemistry, biology, and astronomy for data manipulation, numerical simulations, and analysis. NumPy is often used in numerical simulations and computational modelling for solving differential equations, optimisation problems, and other mathematical problems.
- **Array manipulation:** NumPy provides an assortment of methods for manipulating arrays, such as resizing, slicing, indexing, stacking, splitting, and concatenating arrays. These techniques are essential for preparing and manipulating data in diverse scientific computing jobs.
- **Finance and Economics:** The NumPy API is also widely used in financial data analysis and economics to do portfolio optimization, risk analysis, time series analysis, and statistical modelling.
- **Engineering and Robotics:** NumPy is used in engineering disciplines such as mechanical, civil, and electrical engineering for tasks like finite element analysis, control system design, and robotics simulations.
- **Image and Signal Processing:** NumPy is extensively used in processing and analyzing images and signals.
- **Data Visualization:** NumPy doesn't provide data visualization but supports Matplotlib and Seaborn libraries to generate plots and visualizations from numerical data.

Overall, NumPy's versatility and efficiency make it an essential Python package across a wide range of application areas in scientific computing, data analysis, and beyond.

## NumPy Example

The following is an example of Python NumPy:

```
# Importing NumPy Array
import numpy as np

# Creating an array using np.array() method
arr = np.array([10, 20, 30, 40, 50])

# Printing
print(arr) # Prints [10 20 30 40 50]
```

## NumPy Compiler

To practice the NumPy example, we provided an online compiler. Practice your NumPy programs here:

[Online NumPy Compiler](#)

## Audience

This NumPy tutorial has been prepared for those who want to learn about the basics and functions of NumPy. It is specifically useful in data science, engineering, agriculture science, management, statistics, research, and other related domains where numerical computation is required. After completing this tutorial, you will find yourself at a moderate level of expertise from where you can take yourself to higher levels of expertise.

## Prerequisites

You should have a basic understanding of computer programming terminologies. A basic understanding of Python and any of the programming languages is a plus.

## NumPy Codebase

NumPy's source code can be found at this github repository:

<https://github.com/numpy/numpy>

# 176. Python Pandas Tutorial

Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. This Pandas tutorial has been prepared for those who want to learn about the foundations and advanced features of the Pandas Python package. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc. In this tutorial, we will learn the various features of Python Pandas and how to use them in practice.

## What is Pandas?

Pandas is a powerful Python library that is specifically designed to work on data frames that have "relational" or "labeled" data. Its aim aligns with doing real-world data analysis using Python. Its flexibility and functionality make it indispensable for various data-related tasks. Hence, this Python package works well for data manipulation, operating a dataset, exploring a data frame, data analysis, and machine learning-related tasks. To work on it we should first install it using a pip command like "pip install pandas" and then import it like "import pandas as pd". After successfully installing and importing, we can enjoy the innovative functions of pandas to work on datasets or data frames. Pandas versatility and ease of use make it a go-to tool for working with structured data in Python.

Generally, Pandas operates a data frame using Series and DataFrame; where Series works on a one-dimensional labeled array holding data of any type like integers, strings, and objects, while a DataFrame is a two-dimensional data structure that manages and operates data in tabular form (using rows and columns).

## Why Pandas?

The beauty of Pandas is that it simplifies the task related to data frames and makes it simple to do many of the time-consuming, repetitive tasks involved in working with data frames, such as:

- **Import datasets** - available in the form of spreadsheets, comma-separated values (CSV) files, and more.
- **Data cleansing** - dealing with missing values and representing them as NaN, NA, or NaT.
- **Size mutability** - columns can be added and removed from DataFrame and higher-dimensional objects.
- **Data normalization** – normalize the data into a suitable format for analysis.
- **Data alignment** - objects can be explicitly aligned to a set of labels.
- **Intuitive merging and joining data sets** – we can merge and join datasets.
- **Reshaping and pivoting of datasets** – datasets can be reshaped and pivoted as per the need.
- **Efficient manipulation and extraction** - manipulation and extraction of specific parts of extensive datasets using intelligent label-based slicing, indexing, and subsetting techniques.
- **Statistical analysis** - to perform statistical operations on datasets.
- **Data visualization** - Visualize datasets and uncover insights.

## Applications of Pandas

The most common applications of Pandas are as follows:

- **Data Cleaning:** Pandas provides functionalities to clean messy data, deal with incomplete or inconsistent data, handle missing values, remove duplicates, and standardize formats to do effective data analysis.
- **Data Exploration:** Pandas easily summarize statistics, find trends, and visualize data using built-in plotting functions, Matplotlib, or Seaborn integration.
- **Data Preparation:** Pandas may pivot, melt, convert variables, and merge datasets based on common columns to prepare data for analysis.
- **Data Analysis:** Pandas supports descriptive statistics, time series analysis, group-by operations, and custom functions.
- **Data Visualization:** Pandas itself has basic plotting capabilities; it integrates and supports data visualization libraries like Matplotlib, Seaborn, and Plotly to create innovative visualizations.
- **Time Series Analysis:** Pandas supports date/time indexing, resampling, frequency conversion, and rolling statistics for time series data.
- **Data Aggregation and Grouping:** Pandas groupby() function lets you aggregate data and compute group-wise summary statistics or apply functions to groups.
- **Data Input/Output:** Pandas makes data input and export easy by reading and writing CSV, Excel, JSON, SQL databases, and more.
- **Machine Learning:** Pandas works well with Scikit-learn for data preparation, feature engineering, and model input data.
- **Web Scraping:** Pandas may be used with BeautifulSoup or Scrapy to parse and analyze structured web data for web scraping and data extraction.
- **Financial Analysis:** Pandas is commonly used in finance for stock market data analysis, financial indicator calculation, and portfolio optimization.
- **Text Data Analysis:** Pandas' string manipulation, regular expressions, and text mining functions help analyze textual data.
- **Experimental Data Analysis:** Pandas makes manipulating and analyzing large datasets, performing statistical tests, and visualizing results easy.

## Audience: Who Should Learn Pandas

This Pandas tutorial has been prepared for those who want to learn about the foundations and advanced features of the Pandas Python package. It is most widely used in the domain of data science, engineering, research, agriculture science, management, statistics, and other related fields where computation on a data set requires or explores the data frames to find out the data insights that are required to make fruitful decisions.

After completing this tutorial, you will find yourself skilled in pandas Python package from where you can take yourself to the next levels of expertise on other Python packages like Matplotlib, SciPy, scikit-learn, scikit-image, and many more to keep mastering Python language.

Pandas library uses most of the functionalities of NumPy. It is suggested to you to go through our tutorial on NumPy.

## Prerequisites to Learn Pandas

You should have a basic understanding of computer programming. A basic understanding of Python and any of the programming languages is a plus. Basic knowledge of statistics and mathematics is helpful for data analysis and interpretation. Pandas provide functions for descriptive statistics, aggregation, and computation of summary metrics. By having a

strong foundation of above mentioned, you'll be well-equipped to leverage the power of Pandas for data manipulation and analysis tasks.

## Pandas Codebase

You can find the source for the Pandas at <https://github.com/jvns/pandas-cookbook>

## Frequently Asked Questions about Python Pandas

There are some very Frequently Asked Questions(FAQ) about Python Pandas, this section tries to answer them briefly.

### What is Python pandas used for?

Pandas is a Python library used for data manipulation and analysis. It is widely used in the domain of data science, engineering, research, agriculture science, management, statistics, and other related fields where you need to work with datasets.

### List Key Features of Pandas.

The key features of Pandas as follows –

- Fast and Efficient DataFrame Object.
- Pandas supports various data loading tools for in-memory data objects.
- Data alignment and handling of missing data.
- Pandas allows you to reshaping and pivoting of datasets.
- Label-based slicing, indexing and subsetting of large data sets.
- Insert or delete columns from a data structure.
- Group by data for aggregation and transformations.
- High performance merging and joining.
- Time Series functionality.

### Define Series in Pandas.

A Series in Pandas is a one-dimensional labeled array capable of holding data of any type (integer, string, float, Python objects, etc.).

### What are the two main data types in pandas?

The two primary data structures of pandas are –

- Series (1-dimensional)
- DataFrame (2-dimensional)

### Why do we need pandas in Python?

Pandas is the best tool for handling real-world messy data. It is built on top of NumPy and is open-source. Pandas allows for fast and effective data manipulation using its data structures, Series and DataFrame. It handles missing data, supports multiple file formats, and facilitates data cleaning and analysis.

### Is Python pandas free for commercial use?

Yes, Python pandas is free for commercial use. It is accessible to everyone and free for users to use and modify.

## Who developed Python pandas?

Pandas development began in 2008 at AQR Capital Management. By the end of 2009, it became open-source, and it is now actively supported by a community of contributors worldwide.

## What is the structure of pandas?

The two primary data structures of pandas are:

- **Series** – 1-dimensional labeled array.
- **DataFrame** – 2-dimensional table of data with labeled axes.

## How to Install Pandas in Python?

The easiest way to install pandas is to install it as part of the Anaconda distribution, a cross-platform distribution for data analysis and scientific computing. The Conda package manager is the recommended installation method for most users. For further details, refer to our Environment Setup page.

## What is the difference between pandas and NumPy?

Pandas provides high-level data manipulation tools built on top of NumPy. The Pandas module mainly works with tabular data, whereas the NumPy module works with numerical data.

## What can you do using Pandas?

Pandas is a Python package that provides fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It is a fundamental high-level building block for performing practical, real-world data analysis in Python, aiming to be the most powerful and flexible open-source data analysis/manipulation tool available in any language.

## Which is the best place to learn Python pandas?

The best place to learn Python pandas is through our comprehensive and user-friendly tutorial. Our Python Pandas tutorial provides an excellent starting point for understanding data analysis programming with Python pandas. You can explore our simple and effective learning materials at your own pace.

## How to Learn Python pandas?

Following are some tips to learn Python Pandas –

- Decide to learn Python Pandas and stay committed to your goal.
- Install the necessary tools like Anaconda or Miniconda on your computer.
- Start with our Python Pandas tutorial and progress step by step from the basics.
- Read more articles, watch online courses, or buy a book on Python Pandas to deepen your understanding.
- Apply what you've learned by developing small projects that incorporate Python Pandas and other technologies.

## How do I handle missing values in a DataFrame?

You can handle missing values in a DataFrame by –

- Inserting missing data
- Performing calculations with missing data
- Dropping missing data
- Filling missing data

# 177. SciPy Tutorial

SciPy, a scientific library for Python is an open source, BSD-licensed library for mathematics, science and engineering. The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The main reason for building the SciPy library is that, it should work with NumPy arrays. It provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization. This is an introductory tutorial, which covers the fundamentals of SciPy and describes how to deal with its various modules.

## Audience

This tutorial is prepared for the readers, who want to learn the basic features along with the various functions of SciPy. After completing this tutorial, the readers will find themselves at a moderate level of expertise, from where they can take themselves to higher levels of expertise.

## Prerequisites

Before proceeding with the various concepts given in this tutorial, it is being expected that the readers have a basic understanding of Python. In addition to this, it will be very helpful, if the readers have some basic knowledge of other programming languages. SciPy library depends on the NumPy library, hence learning the basics of NumPy makes the understanding easy.

# 178. Matplotlib Tutorial

## What Is Matplotlib?

Matplotlib is one of the most popular Python packages used for data visualization. It is a cross-platform library for making 2D plots from data in arrays. It provides an object-oriented API that helps in embedding plots in applications using Python GUI toolkits such as PyQt, WxPython, or Tkinter. It can be used in Python and IPython shells, Jupyter notebook and web application servers also.

Matplotlib is a Python library that is specifically designed to do effective data visualization. It's a cornerstone of plotting libraries in Python which empowers beginners to dive into the world of attractive data visualization. Matplotlib is an open-source Python library that offers various data visualization (like Line plots, histograms, scatter plots, bar charts, Scatter plots, Pie Charts, and Area Plot etc). A beauty of the Python matplotlib library is its Python code. Its script is structured which denotes that a few lines of code are all that are required in most instances to generate a visual data plot.

## Matplotlib and Pyplot

Matplotlib is a versatile toolkit that allows for the creation of static, animated, and interactive visualizations in the Python programming language.

Generally, matplotlib overlays two APIs:

- **The pyplot API:** to make plot using matplotlib.pyplot.
- **Object-Oriented API:** A group of objects assembled with greater flexibility than pyplot. It provides direct access to Matplotlib's backend layers.

Matplotlib simplifies simple tasks and enables complex tasks to be accomplished. Following are the key aspects of matplotlib:

- Matplotlib offers to create quality plots.
- Matplotlib offers interactive figures and customizes their visual style that can be manipulated as per need.
- Matplotlib offers export to many file formats.

## Online Editor

We have provided an Online Python Compiler/Interpreter. Which helps you to Edit and Execute the Python code directly from your browser. You can also execute the Matplotlib programs using this.

Try to click the icon run button to run the following matplotlib code to display a basic line plot.

```
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.linspace(0, 2 * np.pi, 200)
```

```

y = np.sin(x)

fig, ax = plt.subplots(figsize=(7, 4))
ax.set_title('Sin Wave')
ax.plot(x, y)
plt.show()

```

## Applications of Matplotlib

The most common applications of matplotlib include:

- **Data Visualization:** Many scientific researches, data analytics, and machine learning applications use Matplotlib to visualize data.
- **Scientific Research:** Matplotlib helps scientists visualize experimental data, simulation findings, and statistical analysis. It improves data comprehension and communication for researchers.
- **Engineering:** Matplotlib helps engineers to visualize sensor readings, simulation findings, and design parameters. It excels at graphing in mechanical, civil, aeronautical, and electrical engineering.
- **Finance:** Finance professionals use Matplotlib to visualize stock prices, market trends, portfolio performance, and risk assessments. It helps analysts and traders make decisions by visualizing complicated financial data in simple graphics.
- **Geospatial Analysis:** Matplotlib, Basemap, and Cartopy are used to visualize geographical data such as maps, satellite images, climate data, and GIS data. Users may generate interactive maps, plot geographical characteristics, and overlay data for spatial analysis.
- **Biology and Bioinformatics:** Matplotlib helps biologists and bioinformaticians visualize DNA sequences, protein structures, phylogenetic trees, and gene expression patterns. It helps researchers to visualize complicated biological processes.
- **Education:** Educational institutions use Matplotlib to teach data visualization, programming, and scientific computing. Its easy-to-use visualization interface makes it suited for high school and university students and teachers.
- **Web Development:** Flask, Django, and Plotly Dash can incorporate Matplotlib into online apps. It lets developers build dynamic, interactive visualizations for web pages and dashboards.
- **Machine Learning:** Machine learning projects visualize data distributions, model performance metrics, decision boundaries, and training progress with Matplotlib. It helps machine learning practitioners analyze algorithm behavior and troubleshoot model-building concerns.
- **Presentation and Publication:** Matplotlib creates high-quality figures for scientific research, reports, presentations, and posters. It offers many customization options to optimize the plot look for publishing and presentation.

Matplotlib lets users produce informative and attractive visualizations for analysis, communication, and decision-making.

## Why to Learn Matplotlib?

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It has become one of the most widely used plotting libraries in the Python ecosystem. Some of the reasons are as to make Matplotlib popular:

- **Plotting Capabilities:** Matplotlib provides extensive functionality for creating a variety of plots like line plots, scatter plots, bar charts, histograms, pie charts, 3D plots, etc.
- **Quality Graphics:** It allows its users to control every aspect of their plots, including colors, line styles, markers, fonts, and annotations.
- **Integration with NumPy and Pandas:** Matplotlib works with NumPy and Pandas to visualize arrays, data frames, and other data structures.
- **Cross-Platform Compatibility:** Matplotlib operates on Windows, macOS, and Linux, making it accessible to many people.
- **Extensive Documentation and Tutorials:** Beginners and experts may easily get started with Matplotlib thanks to its extensive documentation and online training.

Matplotlib is a robust and versatile Python toolkit used for visualizing data which makes it indispensable for data analysts, scientists, engineers, and other professionals working with data.

## Who Should Learn Matplotlib?

This Matplotlib tutorial has been prepared for those who want to learn about the foundations and advances of the Matplotlib Python package. It is most widely used in the domains of data science, engineering, research, agriculture science, management, statistics, and other related fields where data visualization primarily requires finding data insights using charts and graphs to understand the data patterns. It really helps the companies in strategic decision-making.

This Matplotlib tutorial is designed for beginners and professionals to cover matplotlib concepts, including the process of installing matplotlib and making different plots. It offers a detailed description, valuable insights, and the fundamental principles of constructing attractive visualizations. Whether you are a student embarking in the field of data science or a professional, this tutorial provides a strong foundation to explore data analysis using data visualization through Matplotlib to present the data. Hence, this tutorial aims to explain the different functions of Matplotlib for data analysis.

## Prerequisites to Learn Matplotlib

You should have a basic understanding of computer programming. A basic understanding of Python and any of the programming languages is a plus. Basic knowledge of statistics and mathematics is helpful for data analysis and interpretation. Matplotlib offers functions for data visualization. By having a strong foundation of above mentioned, you'll be well-equipped to leverage the power of matplotlib for data visualization.

## Frequently Asked Questions about Matplotlib

There are some Frequently Asked Questions(FAQ) about Matplotlib, this section tries to answer them briefly.

### What is Matplotlib used for?

Matplotlib is used for creating static, animated, and interactive visualizations in Python. It's a powerful library widely used for data visualization tasks, offering various functionalities to generate plots such as line plots, scatter plots, bar charts, histograms, and 3D plots.

### **Why can Matplotlib be Confusing?**

Because of its nomenclature for plots and the two plotting interfaces: the pyplot approach and the object-oriented style. These aspects may initially challenge the users who are trying to understand the library.

### **Why is it called Matplotlib?**

The name Matplotlib originated from the library's early goal of emulating the MATLAB graphics commands. However, it's important to note that Matplotlib is independent of MATLAB and can be used in a Pythonic, object-oriented manner.

### **Why is Matplotlib so Popular?**

It offers a wide range of functionalities to create plots like line plots, scatter plots, bar charts, histograms, 3D plots, and much more. Due to its accessibility, Matplotlib is recognized as one of the most popular data visualization tools.

### **Why is Matplotlib helpful?**

Matplotlib is helpful because it simplifies the process of creating plots and visualizing data. It allows users to generate plots with just a few commands, making it accessible for both beginners and experienced programmers.

### **What are the advantages of Matplotlib?**

Matplotlib offers several advantages few of them are listed below –

- Efficient Data Access
- Robust Data Handling
- Creates publication-quality plots
- Support for Multiple Outputs
- It also provides graphical user interfaces
- Flexible Data Representation
- Advanced Visualization Capabilities
- Open-Source Nature
- Simplifies data analysis by providing a user-friendly interface and powerful tools
- It provides high-quality images

### **Who uses Matplotlib?**

Matplotlib is used by persons in various fields, including data science, finance, engineering, and research. Particularly used within the data science industries. Its flexibility and capability to handle complex data visualization tasks make it a popular choice among individuals working with data.

### **Who invented Matplotlib?**

Matplotlib was originally written by John D. Hunter, a neurobiologist, with the initial goal of emulating MATLAB's plotting capabilities to work with EEG data. Then it has had an active development community and is distributed under a BSD-style license.

## How to Learn Matplotlib?

Learning Matplotlib involves exploring its simple and advanced commands. You can start by following tutorials and examples, gradually building confidence in creating plots for data visualization. Our comprehensive learning materials provide a solid foundation for mastering Matplotlib. Also, it is good to follow the Official Documentation.

## Is Matplotlib a data visualization?

Yes, Matplotlib is a powerful library for data visualization in Python. It allows users to create a variety of plots, charts, and graphs to effectively represent and analyze data.

## What are the two approaches to Matplotlib?

There are two main approaches to using Matplotlib: the pyplot approach (also known as An implicit or functional interface) and the object-oriented style (called An explicit or Axes interface).

## What are the benefits of Matplotlib?

Matplotlib provides benefits such as the ability to create high-quality plots, compatibility with various output formats, ease of integration into graphical user interfaces, and support LaTeX and math text, allowing users to display mathematical equations and symbols in their plots, such as axis labels, titles, and annotations.

## What type of histogram does Matplotlib support?

Matplotlib supports various types of histograms, including bar charts, stacked bar charts, and 3D histograms.

## Which is the best place to learn Matplotlib?

You can use our simple and the best Matplotlib tutorial to learn Matplotlib. Our tutorial offers an excellent starting point for learning Matplotlib. You can explore our simple and effective learning materials at your own pace.

## What are the 3 layers of Matplotlib architecture?

Matplotlib's architecture consists of three layers –

- **Backend Layer** – This layer is responsible for handling the display of Matplotlib figures.
- **Artist Layer** – The Artist layer is crucial in the creation and manipulation of visual elements within Matplotlib. Each visual component, such as lines, text, and shapes, is represented as an "artist" in this layer.
- **Scripting Layer** – The Scripting layer is where users interact with Matplotlib to generate visualizations using Python scripts or code.

## What is Matplotlib font?

The Matplotlib font refers to the text appearance in plots generated using Matplotlib. The library provides robust support for customizing text properties in plots. By default, Matplotlib uses the DejaVu Sans font. However, users have the flexibility to configure default fonts and even use their custom fonts.

# 179. Django Tutorial

Django is a web development framework that assists in building and maintaining quality web applications. Django helps eliminate repetitive tasks making the development process an easy and time saving experience. This tutorial gives a complete understanding of Django.

## Audience

This tutorial is designed for developers who want to learn how to develop quality web applications using the smart techniques and tools offered by Django.

## Prerequisites

Before you proceed, make sure that you understand the basics of procedural and object-oriented programming: control structures, data structures and variables, classes, objects, etc.

## Frequently Asked Questions about Django

There are some very Frequently Asked Questions(FAQ) about Django, this section tries to answer them briefly.

### What Is the Django Framework?

Django is a web development framework for building websites and web applications using Python. It provides a set of tools and features that make it easier to create web applications quickly and efficiently. Django handles common web development tasks like database management, URL routing, form handling, and user authentication, allowing developers to focus on building the unique features of application instead of reinventing the wheel. It follows the "Don't Repeat Yourself" (DRY) principle, promoting code reuse and maintainability.

### What is Django used for?

Django is used to build websites and web applications. It provides tools and features that help developers create these digital platforms more easily and efficiently. With Django, developers can handle tasks like managing databases, routing URLs, handling user input through forms, and managing user authentication. Essentially, Django simplifies the process of building complex web applications by providing a structured framework that handles many common tasks, allowing developers to focus on creating the unique features and functionalities of their websites or web apps.

### Who invented django?

Django was invented by Adrian Holovaty and Simon Willison in 2003 while working at a newspaper company called World Online. They created Django to help them build web applications quickly and efficiently for the newsroom. Django was later released as an open-source project in 2005, allowing developers worldwide to use and contribute to its

development. Since then, it has grown into a popular web development framework known for its simplicity, flexibility, and productivity-enhancing features.

## How to check Django version?

To check the Django version, you can use a command prompt or terminal –

- Open a command prompt or terminal window on your computer.
- Type the following command and press Enter: `python -m django --version`.
- You will see the Django version displayed on the screen.

## How to Learn Django?

To learn Django, start by setting up your environment with Python and Django installed. Then, explore the basics of Django through tutorials or official documentation, learning about its architecture and key components like views, models, and templates. Practice building simple projects to reinforce your understanding, gradually adding more advanced features as you progress. Finally, stay updated with the latest Django updates and best practices, and regularly practice by working on real-world projects or contributing to open-source projects to sharpen your skills.

## What are the disadvantages of Django?

The disadvantages of Django include its complexity for beginners, as it has a steep learning curve compared to simpler frameworks. Additionally, Django can be seen as too opinionated, i.e. it imposes certain conventions and patterns which may not always align with the preferences of developers. It is also known for being heavyweight, i.e. it includes many built-in features that may not be needed for all projects, potentially leading to unnecessary complexity and overhead. Lastly, Django's ORM (Object-Relational Mapping) can sometimes result in performance issues when dealing with large datasets or complex queries, requiring optimization efforts to overcome.

## What are the skills required for becoming a Django Developer?

To become a Django developer, you will need strong Python programming skills, a solid understanding of web development concepts, proficiency in Django framework including models, views, templates, and forms, familiarity with databases and SQL, and knowledge of version control systems like Git. Problem-solving abilities, effective communication, and collaboration skills are also essential for success in this role.

## Which Companies Use Django?

Many companies use Django for building their websites and web applications. Some notable examples include Instagram, which initially used Django to handle its massive photo-sharing platform, Pinterest, which relies on Django for its content-sharing site, and Spotify, which uses Django for various aspects of its music streaming service.

Additionally, companies like Disqus, Eventbrite, and Dropbox have also adopted Django for their web development needs.

## What is the best Database for Django?

The best database for Django depends on the specific requirements of your project. However, Django officially supports several databases, including SQLite, PostgreSQL, MySQL, and Oracle. For small projects or development purposes, SQLite, which is included

with Python, is often used due to its simplicity and ease of setup. For larger projects requiring scalability and advanced features, PostgreSQL is a popular choice because of its robustness, performance, and support for complex data types and operations.

## How to write Hello World in Django?

To write "Hello World" in Django, start by setting up Django on your computer. Create a new Django project and app using command line tools. Define a URL route in the urls.py file of the project and create a view function in the app's views.py file that returns an HTTP response with the "Hello World" message. Map the URL route to the view function. Finally, run the Django development server and navigate to the specified URL in a web browser to see the "Hello World" message displayed on the webpage.

# 180. OpenCV Python Tutorial

OpenCV stands for Open Source Computer Vision and is a library of functions which is useful in real time computer vision application programming. OpenCV-Python is a Python wrapper around C++ implementation of OpenCV library. It is a rapid prototyping tool for computer vision problems. This tutorial is designed to give fluency in OpenCV-Python and to explain how you can use it in your applications.

## Audience

This tutorial is designed for the computer science students and professionals who wish to gain expertise in the field of computer vision applications. After completing this tutorial, you will find yourself at a moderate level of expertise in using OpenCV-Python from where you can take yourself to next levels.

## Prerequisites

We assume you have prior knowledge of Python and NumPy libraries. Moreover, it would be beneficial, if you are well acquainted with JAVA programming language. You can go through our tutorials on Python, JAVA and NumPy, if required, before beginning with this tutorial.

## Python Miscellaneous

# 181. Python - Date and Time

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Following modules in Python's standard library handle date and time related processing –

- `DateTime` module
- `Time` module
- `Calendar` module

## What are Tick Intervals

Tick intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular time module available in Python, which provides functions for working with times, and for converting between representations. The function `time.time()` returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

### Example

```
import time # This is required to include time module.  
ticks = time.time()  
print ("Number of ticks since 12:00am, January 1, 1970:", ticks)
```

This would produce a result something as follows –

```
Number of ticks since 12:00am, January 1, 1970: 1681928297.5316436
```

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

## What is TimeTuple?

Many of the Python's time functions handle time as a tuple of 9 numbers, as shown below –

| Index | Field        | Values                              |
|-------|--------------|-------------------------------------|
| 0     | 4-digit year | 2016                                |
| 1     | Month        | 1 to 12                             |
| 2     | Day          | 1 to 31                             |
| 3     | Hour         | 0 to 23                             |
| 4     | Minute       | 0 to 59                             |
| 5     | Second       | 0 to 61 (60 or 61 are leap-seconds) |
| 6     | Day of Week  | 0 to 6 (0 is Monday)                |

|   |                  |                                           |
|---|------------------|-------------------------------------------|
| 7 | Day of year      | 1 to 366 (Julian day)                     |
| 8 | Daylight savings | -1, 0, 1, -1 means library determines DST |

**For example,**

```
>>>import time
>>> print (time.localtime())
```

This would produce an output as follows –

```
time.struct_time(tm_year=2023, tm_mon=4, tm_mday=19, tm_hour=23, tm_min=49,
tm_sec=8, tm_wday=2, tm_yday=109, tm_isdst=0)
```

The above tuple is equivalent to struct\_time structure. This structure has the following attributes –

| Index | Attributes | Values                                    |
|-------|------------|-------------------------------------------|
| 0     | tm_year    | 2016                                      |
| 1     | tm_mon     | 1 to 12                                   |
| 2     | tm_mday    | 1 to 31                                   |
| 3     | tm_hour    | 0 to 23                                   |
| 4     | tm_min     | 0 to 59                                   |
| 5     | tm_sec     | 0 to 61 (60 or 61 are leap-seconds)       |
| 6     | tm_wday    | 0 to 6 (0 is Monday)                      |
| 7     | tm_yday    | 1 to 366 (Julian day)                     |
| 8     | tm_isdst   | -1, 0, 1, -1 means library determines DST |

## Getting the Current Time

To translate a time instance from seconds since the epoch floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all valid nine items.

```
import time
localtime = time.localtime(time.time())
print ("Local current time :", localtime)
```

This would produce the following result, which could be formatted in any other presentable form –

```
Local current time : time.struct_time(tm_year=2023, tm_mon=4, tm_mday=19,
tm_hour=23, tm_min=42, tm_sec=41, tm_wday=2, tm_yday=109, tm_isdst=0)
```

## Getting the Formatted Time

You can format any time as per your requirement, but a simple method to get time in a readable format is `asctime()` –

```
import time

localtime = time.asctime( time.localtime(time.time()) )
print ("Local current time : ", localtime)
```

This would produce the following output –

```
Local current time : Wed Apr 19 23:45:27 2023
```

## Getting the Calendar for a Month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008).

```
import calendar
cal = calendar.month(2023, 4)
print ("Here is the calendar:")
print (cal)
```

This would produce the following output –

```
Here is the calendar:
```

```
April 2023
```

```
Mo Tu We Th Fr Sa Su
```

```
    1  2
```

```
 3  4  5  6  7  8  9
```

```
10 11 12 13 14 15 16
```

```
17 18 19 20 21 22 23
```

```
24 25 26 27 28 29 30
```

## The time Module

There is a popular time module available in Python, which provides functions for working with times and for converting between representations. Here is the list of all available methods.

| Sr.No. | Function with Description                                                                                                                                                                                                  |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | <a href="#">time.altzone</a>                                                                                                                                                                                               |
| 1      | The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero. |
| 2      | <a href="#">time.asctime([tupletime])</a>                                                                                                                                                                                  |

|    |                                                                                                                                                                                                                                           |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.                                                                                                                                       |
| 3  | <a href="#">time.clock()</a><br>Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().              |
| 4  | <a href="#">time.ctime([secs])</a><br>Like asctime(localtime(secs)) and without arguments is like asctime( )                                                                                                                              |
| 5  | <a href="#">time.gmtime([secs])</a><br>Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0                                                                |
| 6  | <a href="#">time.localtime([secs])</a><br>Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules). |
| 7  | <a href="#">time.mktime(tupletime)</a><br>Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.                                            |
| 8  | <a href="#">time.sleep(secs)</a><br>Suspends the calling thread for secs seconds.                                                                                                                                                         |
| 9  | <a href="#">time.strftime(fmt[,tupletime])</a><br>Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.                                                    |
| 10 | <a href="#">time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')</a><br>Parses str according to format string fmt and returns the instant in time-tuple format.                                                                                  |
| 11 | <a href="#">time.time()</a><br>Returns the current time instant, a floating-point number of seconds since the epoch.                                                                                                                      |
| 12 | <a href="#">time.tzset()</a><br>Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.                                                                                    |

Let us go through the functions briefly.

There are two important attributes available with time module. They are –

| Sr.No. | Attribute with Description                                                                                                                                |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | <a href="#">time.timezone</a>                                                                                                                             |
| 1      | Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa). |

| <b>time.tzname</b> |                                                                                                                                             |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| 2                  | Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively. |

## The calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call the calendar.setfirstweekday() function.

Here is a list of functions available with the calendar module –

| Sr.No. | Function with Description                                                                                                                                                                                                                                                             |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>calendar.calendar(year,w=2,l=1,c=6)</b><br>Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.       |
| 2      | <b>calendar.firstweekday( )</b><br>Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.                                                                                                         |
| 3      | <b>calendar.isleap(year)</b><br>Returns True if year is a leap year; otherwise, False.                                                                                                                                                                                                |
| 4      | <b>calendar.leapdays(y1,y2)</b><br>Returns the total number of leap days in the years within range(y1,y2).                                                                                                                                                                            |
| 5      | <b>calendar.month(year,month,w=2,l=1)</b><br>Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.         |
| 6      | <b>calendar.monthcalendar(year,month)</b><br>Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.                                                         |
| 7      | <b>calendar.monthrange(year,month)</b><br>Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12. |
| 8      | <b>calendar.prcal(year,w=2,l=1,c=6)</b>                                                                                                                                                                                                                                               |

|    |                                                                                                                                                           |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | Like print calendar.calendar(year,w,l,c).                                                                                                                 |
| 9  | <b>calendar.pmonth(year,month,w=2,l=1)</b>                                                                                                                |
|    | Like print calendar.month(year,month,w,l).                                                                                                                |
| 10 | <b>calendar.setfirstweekday(weekday)</b>                                                                                                                  |
|    | Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).                                                      |
| 11 | <b>calendar.timegm(tupletime)</b>                                                                                                                         |
|    | The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch. |
| 12 | <b>calendar.weekday(year,month,day)</b>                                                                                                                   |
|    | Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).                  |

## Python datetime Module

Python's datetime module is included in the standard library. It consists of classes that help manipulate data and time data and perform date time arithmetic.

Objects of datetime classes are either aware or naïve. If the object includes timezone information, it is aware, and if not it is classified as naïve. An object of date class is naïve, whereas time and datetime objects are aware.

### Python date Object

A date object represents a date with year, month, and day. The current Gregorian calendar is indefinitely extended in both directions.

#### Syntax

```
datetime.date(year, month, day)
```

Arguments must be integers, in the following ranges –

- **year** – MINYEAR <= year <= MAXYEAR
- **month** – 1 <= month <= 12
- **day** – 1 <= day <= number of days in the given month and year

If the value of any argument outside those ranges is given, ValueError is raised.

#### Example

```
from datetime import date
date1 = date(2023, 4, 19)
print("Date:", date1)
date2 = date(2023, 4, 31)
```

It will produce the following output –

```
Date: 2023-04-19
```

```

Traceback (most recent call last):
  File "C:\Python311\hello.py", line 8, in <module>
    date2 = date(2023, 4, 31)
ValueError: day is out of range for month

```

## Date class attributes

- **date.min** – The earliest representable date, date(MINYEAR, 1, 1).
- **date.max** – The latest representable date, date(MAXYEAR, 12, 31).
- **date.resolution** – The smallest possible difference between non-equal date objects.
- **date.year** – Between MINYEAR and MAXYEAR inclusive.
- **date.month** – Between 1 and 12 inclusive.
- **date.day** – Between 1 and the number of days in the given month of the given year.

## Example

```

from datetime import date

# Getting min date
mindate = date.min
print("Minimum Date:", mindate)

# Getting max date
maxdate = date.max
print("Maximum Date:", maxdate)

Date1 = date(2023, 4, 20)
print("Year:", Date1.year)
print("Month:", Date1.month)
print("Day:", Date1.day)

```

It will produce the following output –

```

Minimum Date: 0001-01-01
Maximum Date: 9999-12-31
Year: 2023
Month: 4
Day: 20

```

## Class Methods in Date Class

- **today()** – Return the current local date.

- **fromtimestamp(timestamp)** – Return the local date corresponding to the POSIX timestamp, such as output generated by time.time().
- **fromordinal(ordinal)** – Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.
- **fromisoformat(date\_string)** – Return a date corresponding to a date\_string given in any valid ISO 8601 format, except ordinal dates

### Example

```
from datetime import date
print (date.today())
d1=date.fromisoformat('2023-04-20')
print (d1)
d2=date.fromisoformat('20230420')
print (d2)
d3=date.fromisoformat('2023-W16-4')
print (d3)
```

It will produce the following output –

```
2023-04-20
2023-04-20
2023-04-20
2023-04-20
```

### Instance Methods in Date Class

- **replace()** – Return a date by replacing specified attributes with new values by keyword arguments are specified.
- **timetuple()** – Return a time.struct\_time such as returned by time.localtime().
- **toordinal()** – Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any date object d, date.fromordinal(d.toordinal()) == d.
- **weekday()** – Return the day of the week as an integer, where Monday is 0 and Sunday is 6.
- **isoweekday()** – Return the day of the week as an integer, where Monday is 1 and Sunday is 7.
- **isocalendar()** – Return a named tuple object with three components: year, week and weekday.
- **isoformat()** – Return a string representing the date in ISO 8601 format, YYYY-MM-DD:
- **\_\_str\_\_()** – For a date d, str(d) is equivalent to d.isoformat()
- **ctime()** – Return a string representing the date:
- **strftime(format)** – Return a string representing the date, controlled by an explicit format string.
- **\_\_format\_\_(format)** – Same as date.strftime().

### Example

```
from datetime import date
```

```

d = date.fromordinal(738630) # 738630th day after 1. 1. 0001
print (d)
print (d.timetuple())
# Methods related to formatting string output
print (d.isoformat())
print (d.strftime("%d/%m/%y"))
print (d.strftime("%A %d. %B %Y"))
print (d.ctime())

print ('The {1} is {0:%d}, the {2} is {0:%B}'.format(d, "day", "month"))

# Methods for extracting 'components' under different calendars
t = d.timetuple()
for i in t:
    print(i)

ic = d.isocalendar()
for i in ic:
    print(i)

# A date object is immutable; all operations produce a new object
print (d.replace(month=5))

```

It will produce the following output –

```

2023-04-20
time.struct_time(tm_year=2023, tm_mon=4, tm_mday=20, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=3, tm_yday=110, tm_isdst=-1)
2023-04-20
20/04/23
Thursday 20. April 2023
Thu Apr 20 00:00:00 2023
The day is 20, the month is April.
2023
4
20
0

```

```

0
0
3
110
-1
2023
16
4
2023-05-20

```

## Python time Module

An object time class represents the local time of the day. It is independent of any particular day. If the object contains the tzinfo details, it is the aware object. If it is None then the time object is the naive object.

### Syntax

```
datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
```

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments must be integers in the following ranges –

- **hour** –  $0 \leq \text{hour} < 24$ ,
- **minute** –  $0 \leq \text{minute} < 60$ ,
- **second** –  $0 \leq \text{second} < 60$ ,
- **microsecond** –  $0 \leq \text{microsecond} < 1000000$

If any of the arguments are outside those ranges is given, ValueError is raised.

### Example

```

from datetime import time

time1 = time(8, 14, 36)
print("Time:", time1)

time2 = time(minute = 12)
print("time", time2)

time3 = time()
print("time", time3)

time4 = time(hour = 26)

```

It will produce the following output –

```
Time: 08:14:36
time 00:12:00
time 00:00:00
Traceback (most recent call last):
  File "/home/cg/root/64b912f27faef/main.py", line 12, in
    time4 = time(hour = 26)
ValueError: hour must be in 0..23
```

## Class attributes

- **time.min** – The earliest representable time, time(0, 0, 0, 0).
- **time.max** – The latest representable time, time(23, 59, 59, 999999).
- **time.resolution** – The smallest possible difference between non-equal time objects.

## Example

```
from datetime import time
print(time.min)
print(time.max)
print (time.resolution)
```

It will produce the following output –

```
00:00:00
23:59:59.999999
0:00:00.000001
```

## Instance attributes

- **time.hour** – In range(24)
- **time.minute** – In range(60)
- **time.second** – In range(60)
- **time.microsecond** – In range(1000000)
- **time.tzinfo** – the tzinfo argument to the time constructor, or None.

## Example

```
from datetime import time
t = time(8,23,45,5000)
print(t.hour)
print(t.minute)
print (t.second)
print (t.microsecond)
```

It will produce the following output –

```
8
23
455000
```

## Instance Methods of time Object

- **replace()** – Return a time with the same value, except for those attributes given new values by whichever keyword arguments are specified.
- **isoformat()** – Return a string representing the time in ISO 8601 format
- **\_\_str\_\_()** – For a time t, str(t) is equivalent to t.isoformat().
- strftime(format) – Return a string representing the time, controlled by an explicit format string.
- **\_\_format\_\_(format)** – Same as time.strftime().
- **utcoffset()** – If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(None),
- **dst()** – If tzinfo is None, returns None, else returns self.tzinfo.dst(None),
- **tzname()** – If tzinfo is None, returns None, else returns self.tzinfo.tzname(None), or raises an exception

## Python datetime object

An object of datetime class contains the information of date and time together. It assumes the current Gregorian calendar extended in both directions; like a time object, and there are exactly 3600\*24 seconds in every day.

### Syntax

```
datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0,
tzinfo=None, *, fold=0)
```

The year, month and day arguments are required.

- **year** – MINYEAR <= year <= MAXYEAR,
- **month** – 1 <= month <= 12,
- **day** – 1 <= day <= number of days in the given month and year,
- **hour** – 0 <= hour < 24,
- **minute** – 0 <= minute < 60,
- **second** – 0 <= second < 60,
- **microsecond** – 0 <= microsecond < 1000000,
- **fold** – in [0, 1].

If any argument in outside ranges is given, ValueError is raised.

### Example

```
from datetime import datetime
dt = datetime(2023, 4, 20)
print(dt)

dt = datetime(2023, 4, 20, 11, 6, 32, 5000)
print(dt)
```

It will produce the following output –

```
2023-04-20 00:00:00
2023-04-20 11:06:32.005000
```

### Class attributes

- **datetime.min** – The earliest representable datetime, datetime(MINYEAR, 1, 1, tzinfo=None).
- **datetime.max** – The latest representable datetime, datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None).
- **datetime.resolution** – The smallest possible difference between non-equal datetime objects, timedelta(microseconds=1).

### Example

```
from datetime import datetime
min = datetime.min
print("Min DateTime ", min)

max = datetime.max
print("Max DateTime ", max)
```

It will produce the following output –

```
Min DateTime 0001-01-01 00:00:00
Max DateTime 9999-12-31 23:59:59.999999
```

### Instance Attributes of datetime Object

- **datetime.year** – Between MINYEAR and MAXYEAR inclusive.
- **datetime.month** – Between 1 and 12 inclusive.
- **datetime.day** – Between 1 and the number of days in the given month of the given year.
- **datetime.hour** – In range(24)
- **datetime.minute** – In range(60)
- **datetime.second** – In range(60)
- **datetime.microsecond** – In range(1000000).
- **datetime.tzinfo** – The object passed as the tzinfo argument to the datetime constructor, or None if none was passed.
- **datetime.fold** – In [0, 1]. Used to disambiguate wall times during a repeated interval.

### Example

```
from datetime import datetime
dt = datetime.now()

print("Day: ", dt.day)
print("Month: ", dt.month)
```

```
print("Year: ", dt.year)
print("Hour: ", dt.hour)
print("Minute: ", dt.minute)
print("Second: ", dt.second)
```

It will produce the following output –

```
Day: 20
Month: 4
Year: 2023
Hour: 15
Minute: 5
Second: 52
```

## Class Methods of datetime Object

- **today()** – Return the current local datetime, with tzinfo None.
- **now(*tz=None*)** – Return the current local date and time.
- **utcnow()** – Return the current UTC date and time, with tzinfo None.
- **utcfromtimestamp(*timestamp*)** – Return the UTC datetime corresponding to the POSIX timestamp, with tzinfo None
- **fromtimestamp(*timestamp*, *timezone.utc*)** – On the POSIX compliant platforms, it is equivalent todatetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
- **fromordinal(*ordinal*)** – Return the datetime corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.
- **fromisoformat(*date\_string*)** – Return a datetime corresponding to a date\_string in any valid ISO 8601 format.

## Instance Methods of datetime Object

- **date()** – Return date object with same year, month and day.
- **time()** – Return time object with same hour, minute, second, microsecond and fold.
- **timetz()** – Return time object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method time().
- **replace()** – Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified.
- **astimezone(*tz=None*)** – Return a datetime object with new tzinfo attribute *tz*
- **utcoffset()** – If tzinfo is None, returns None, else returns self.tzinfo.utcoffset(self)
- **dst()** – If tzinfo is None, returns None, else returns self.tzinfo.dst(self)
- **tzname()** – If tzinfo is None, returns None, else returns self.tzinfo.tzname(self)
- **timetuple()** – Return a time.struct\_time such as returned by time.localtime().
- **atime.toordinal()** – Return the proleptic Gregorian ordinal of the date.
- **timestamp()** – Return POSIX timestamp corresponding to the datetime instance.
- **isoweekday()** – Return day of the week as an integer, where Monday is 1, Sunday is 7.
- **isocalendar()** – Return a named tuple with three components: year, week and weekday.

- **`isoformat(sep='T', timespec='auto')`** – Return a string representing the date and time in ISO 8601 format
- **`__str__()`** – For a datetime instance d, str(d) is equivalent to d.isoformat(' ').
- **`ctime()`** – Return a string representing the date and time:
- **`strftime(format)`** – Return a string representing the date and time, controlled by an explicit format string.
- **`__format__(format)`** – Same as strftime().

### Example

```
from datetime import datetime, date, time, timezone

# Using datetime.combine()
d = date(2022, 4, 20)
t = time(12, 30)
datetime.combine(d, t)

# Using datetime.now()
d = datetime.now()
print (d)

# Using datetime.strptime()
dt = datetime.strptime("23/04/20 16:30", "%d/%m/%y %H:%M")

# Using datetime.timetuple() to get tuple of all attributes
tt = dt.timetuple()
for it in tt:
    print(it)

# Date in ISO format
ic = dt.isocalendar()
for it in ic:
    print(it)
```

It will produce the following output –

```
2023-04-20 15:12:49.816343
2020
4
23
16
```

```
30
0
3
114
-1
2020
17
4
```

## Python timedelta Object

The timedelta object represents the duration between two dates or two time objects.

### Syntax

```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0,
hours=0, weeks=0)
```

Internally, the attributes are stored in days, seconds and microseconds. Other arguments are converted to those units –

- A millisecond is converted to 1000 microseconds.
- A minute is converted to 60 seconds.
- An hour is converted to 3600 seconds.
- A week is converted to 7 days.

While days, seconds and microseconds are then normalized so that the representation is unique.

### Example

The following example shows that Python internally stores days, seconds and microseconds only.

```
from datetime import timedelta

delta = timedelta(
    days=100,
    seconds=27,
    microseconds=10,
    milliseconds=29000,
    minutes=5,
    hours=12,
    weeks=2
)
# Only days, seconds, and microseconds remain
print (delta)
```

It will produce the following output –

```
114 days, 12:05:56.000010
```

### Example

The following example shows how to add timedelta object to a datetime object.

```
from datetime import datetime, timedelta

date1 = datetime.now()

date2= date1+timedelta(days = 4)
print("Date after 4 days:", date2)

date3 = date1-timedelta(15)
print("Date before 15 days:", date3)
```

It will produce the following output –

```
Date after 4 days: 2023-04-24 18:05:39.509905
Date before 15 days: 2023-04-05 18:05:39.509905
```

### Class Attributes of timedelta Object

- **timedelta.min** – The most negative timedelta object, timedelta(-999999999).
- **timedelta.max** – The most positive timedelta object, timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999).
- **timedelta.resolution** – The smallest possible difference between non-equal timedelta objects, timedelta(microseconds=1)

### Example

```
from datetime import timedelta

# Getting minimum value
min = timedelta.min
print("Minimum value:", min)

max = timedelta.max
print("Maximum value", max)
```

It will produce the following output –

```
Minimum value: -999999999 days, 0:00:00
```

```
Maximum value 999999999 days, 23:59:59.999999
```

## Instance Attributes of timedelta Object

Since only day, second and microseconds are stored internally, those are the only instance attributes for a timedelta object.

- **days** – Between -999999999 and 999999999 inclusive
- **seconds** – Between 0 and 86399 inclusive
- **microseconds** – Between 0 and 999999 inclusive

## Instance Methods of timedelta Object

- **timedelta.total\_seconds()** – Return the total number of seconds contained in the duration.

### Example

```
from datetime import timedelta  
  
year = timedelta(days=365)  
  
years = 5 * year  
  
print (years)  
  
print (years.days // 365)  
  
646  
  
year_1 = years // 5  
  
print(year_1.days)
```

It will produce the following output –

```
1825 days, 0:00:00  
5  
365
```

# 182. Python - math Module

## Python math Module

The math module is a built-in module in Python that is used for performing mathematical operations. This module provides various built-in methods for performing different mathematical tasks.

**Note:** The math module's methods do not work with complex numbers. For that, you can use the cmath module.

## Importing math Module

Before using the methods of the math module, you need to import the math module into your code. The following is the syntax:

```
import math
```

## Methods of Python math Module

The following is the list of math module methods that we have categorized based on their functionality and usage.

### Math Module - Theoretic and Representation Methods

Python includes following theoretic and representation Functions in the math module –

| Sr.No. | Function & Description                                                                                                                                                  |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.ceil(x)</a><br>The ceiling of x: the smallest integer not less than x                                                                                  |
| 2      | <a href="#">math.comb(n,k)</a><br>This function is used to find the returns the number of ways to choose "x" items from "y" items without repetition and without order. |
| 3      | <a href="#">math.copysign(x, y)</a><br>This function returns a float with the magnitude (absolute value) of x but the sign of y.                                        |
| 4      | <a href="#">math.cmp(x, y)</a><br>This function is used to compare the values of two objects. This function is deprecated in Python3.                                   |
| 5      | <a href="#">math.fabs(x)</a><br>This function is used to calculate the absolute value of a given integer.                                                               |
| 6      | <a href="#">math.factorial(n)</a><br>This function is used to find the factorial of a given integer.                                                                    |
| 7      | <a href="#">math.floor(x)</a>                                                                                                                                           |

|    |                                                                                                                                                                                                          |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | This function calculates the floor value of a given integer.                                                                                                                                             |
| 8  | <a href="#"><u>math.fmod(x, y)</u></a><br>The fmod() function in math module returns same result as the "%" operator. However fmod() gives more accurate result of modulo division than modulo operator. |
| 9  | <a href="#"><u>math.frexp(x)</u></a><br>This function is used to calculate the mantissa and exponent of a given number.                                                                                  |
| 10 | <a href="#"><u>math.fsum(iterable)</u></a><br>This function returns the floating point sum of all numeric items in an iterable i.e. list, tuple, array.                                                  |
| 11 | <a href="#"><u>math.gcd(*integers)</u></a><br>This function is used to calculate the greatest common divisor of all the given integers.                                                                  |
| 12 | <a href="#"><u>math.isclose()</u></a><br>This function is used to determine whether two given numeric values are close to each other.                                                                    |
| 13 | <a href="#"><u>math.isfinite(x)</u></a><br>This function is used to determine whether the given number is a finite number.                                                                               |
| 14 | <a href="#"><u>math.isinf(x)</u></a><br>This function is used to determine whether the given value is infinity (+ve or, -ve).                                                                            |
| 15 | <a href="#"><u>math.isnan(x)</u></a><br>This function is used to determine whether the given number is "NaN".                                                                                            |
| 16 | <a href="#"><u>math.isqrt(n)</u></a><br>This function calculates the integer square-root of the given non negative integer.                                                                              |
| 17 | <a href="#"><u>math.lcm(*integers)</u></a><br>This function is used to calculate the least common factor of the given integer arguments.                                                                 |
| 18 | <a href="#"><u>math.ldexp(x, i)</u></a><br>This function returns product of first number with exponent of second number. So, ldexp(x,y) returns $x*2**y$ . This is inverse of frexp() function.          |
| 19 | <a href="#"><u>math.modf(x)</u></a><br>This returns the fractional and integer parts of x in a two-item tuple.                                                                                           |
| 20 | <a href="#"><u>math.nextafter(x, y, steps)</u></a><br>This function returns the next floating-point value after x towards y.                                                                             |
| 21 | <a href="#"><u>math.perm(n, k)</u></a><br>This function is used to calculate the permutation. It returns the number of ways to choose x items from y items without repetition and with order.            |
| 22 | <a href="#"><u>math.prod(iterable, *, start)</u></a>                                                                                                                                                     |

|    |                                                                                                                                                                                                                                                    |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | This function is used to calculate the product of all numeric items in the iterable (list, tuple) given as argument.                                                                                                                               |
| 23 | <a href="#">math.remainder(x,y)</a><br>This function returns the remainder of x with respect to y. This is the difference $x - n*y$ , where n is the integer closest to the quotient $x / y$ .                                                     |
| 24 | <a href="#">math.trunc(x)</a><br>This function returns integral part of the number, removing the fractional part. <code>trunc()</code> is equivalent to <code>floor()</code> for positive x, and equivalent to <code>ceil()</code> for negative x. |
| 25 | <a href="#">math.ulp(x)</a><br>This function returns the value of the least significant bit of the float x. <code>trunc()</code> is equivalent to <code>floor()</code> for positive x, and equivalent to <code>ceil()</code> for negative x.       |

## Math Module - Power and Logarithmic Methods

| Sr.No. | Function & Description                                                                                                                                               |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.cbrt(x)</a><br>This function is used to calculate the cube root of a number.                                                                        |
| 2      | <a href="#">math.exp(x)</a><br>This function calculate the exponential of x: $e^x$                                                                                   |
| 3      | <a href="#">math.exp2(x)</a><br>This function returns 2 raised to power x. It is equivalent to $2^{**x}$ .                                                           |
| 4      | <a href="#">math.expm1(x)</a><br>This function returns e raised to the power x, minus 1. Here e is the base of natural logarithms.                                   |
| 5      | <a href="#">math.log(x)</a><br>This function calculates the natural logarithm of x, for $x > 0$ .                                                                    |
| 6      | <a href="#">math.log1p(x)</a><br>This function returns the natural logarithm of $1+x$ (base e). The result is calculated in a way which is accurate for x near zero. |
| 7      | <a href="#">math.log2(x)</a><br>This function returns the base-2 logarithm of x. This is usually more accurate than $\log(x, 2)$ .                                   |
| 8      | <a href="#">math.log10(x)</a><br>The base-10 logarithm of x for $x > 0$ .                                                                                            |
| 9      | <a href="#">math.pow(x, y)</a>                                                                                                                                       |

|    |                                    |
|----|------------------------------------|
|    | The value of $x^{**}y$ .           |
| 10 | <a href="#">math.sqrt(x)</a>       |
|    | The square root of $x$ for $x > 0$ |

## Math Module - Trigonometric Methods

Python includes following functions that perform trigonometric calculations in the math module –

| Sr.No. | Function & Description                                                                               |
|--------|------------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.acos(x)</a><br>This function returns the arc cosine of $x$ , in radians.            |
| 2      | <a href="#">math.asin(x)</a><br>This function returns the arc sine of $x$ , in radians.              |
| 3      | <a href="#">math.atan(x)</a><br>This function returns the arc tangent of $x$ , in radians.           |
| 4      | <a href="#">math.atan2(y, x)</a><br>This function returns $\text{atan}(y / x)$ , in radians.         |
| 5      | <a href="#">math.cos(x)</a><br>This function returns the cosine of $x$ radians.                      |
| 6      | <a href="#">math.sin(x)</a><br>This function returns the sine of $x$ radians.                        |
| 7      | <a href="#">math.tan(x)</a><br>This function returns the tangent of $x$ radians.                     |
| 8      | <a href="#">math.hypot(x, y)</a><br>This function returns the Euclidean norm, $\sqrt{x^*x + y^*y}$ . |

## Math Module - Angular conversion Methods

Following are the angular conversion function provided by Python math module –

| Sr.No. | Function & Description                                                                             |
|--------|----------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.degrees(x)</a><br>This function converts the given angle from radians to degrees. |
| 2      | <a href="#">math.radians(x)</a><br>This function converts the given angle from degrees to radians. |

## Math Module - Mathematical Constants

The Python math module defines the following mathematical constants –

| Sr.No. | Constants & Description                                                                                                        |
|--------|--------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.pi</a><br>This represents the mathematical constant pi, which equals to "3.141592..." to available precision. |
| 2      | <a href="#">math.e</a>                                                                                                         |

|   |                                                                                                                                                                                       |
|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|   | This represents the mathematical constant e, which is equal to " <b>2.718281...</b> " to available precision.                                                                         |
| 3 | <a href="#">math.tau</a><br>This represents the mathematical constant Tau (denoted by $\tau$ ). It is equivalent to the ratio of circumference to radius, and is equal to <b>2π</b> . |
| 4 | <a href="#">math.inf</a><br>This represents positive infinity. For negative infinity use " <b>-math.inf</b> ".                                                                        |
| 5 | <a href="#">math.nan</a><br>This constant is a floating-point "not a number" (NaN) value. Its value is equivalent to the output of <code>float('nan')</code> .                        |

## Math Module - Hyperbolic Methods

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles. Following are the hyperbolic functions of the Python math module –

| Sr.No. | Function & Description                                                                                                |
|--------|-----------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.acosh(x)</a><br>This function is used to calculate the inverse hyperbolic cosine of the given value. |
| 2      | <a href="#">math.asinh(x)</a><br>This function is used to calculate the inverse hyperbolic sine of a given number.    |
| 3      | <a href="#">math.atanh(x)</a><br>This function is used to calculate the inverse hyperbolic tangent of a number.       |
| 4      | <a href="#">math.cosh(x)</a><br>This function is used to calculate the hyperbolic cosine of the given value.          |
| 5      | <a href="#">math.sinh(x)</a><br>This function is used to calculate the hyperbolic sine of a given number.             |
| 6      | <a href="#">math.tanh(x)</a><br>This function is used to calculate the hyperbolic tangent of a number.                |

## Math Module - Special Methods

Following are the special functions provided by the Python math module –

| Sr.No. | Function & Description                                                                                                                                 |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <a href="#">math.erf(x)</a><br>This function returns the value of the Gauss error function for the given parameter.                                    |
| 2      | <a href="#">math.erfc(x)</a><br>This function is the complementary for the error function. Value of $\text{erf}(x)$ is equivalent to <b>1-erf(x)</b> . |

|   |                                                                                                                                             |
|---|---------------------------------------------------------------------------------------------------------------------------------------------|
|   | <u><a href="#">math.gamma(x)</a></u>                                                                                                        |
| 3 | This is used to calculate the factorial of the complex numbers. It is defined for all the complex numbers except the non-positive integers. |
|   | <u><a href="#">math.lgamma(x)</a></u>                                                                                                       |
| 4 | This function is used to calculate the natural logarithm of the absolute value of the Gamma function at x.                                  |

## Example Usage

The following example demonstrates the use of math module and its methods:

```
# Importing math Module
import math

# Using methods of math module
print(math.sqrt(9))
print(math.pow(3, 3))
print(math.exp(1))
print(math.log(100, 10))

print(math.factorial(4))
print(math.gcd(12, 3))
```

## Output

```
3.0
27.0
2.718281828459045
2.0
24
3
```

# 183. Python - Iterators

## Python Iterators

An iterator in Python is an object that enables traversal through a collection such as a list or a tuple, one element at a time. It follows the iterator protocol by using the implementation of two methods `__iter__()` and `__next__()`.

The `__iter__()` method returns the iterator object itself and the `__next__()` method returns the next element in the sequence by raising a `StopIteration` exception when no more elements are available.

Iterators provide a memory-efficient way to iterate over data, especially useful for large datasets. They can be created from iterable objects using the `iter()` function or implemented using custom classes and generators.

## Iterables vs Iterators

Before going deep into the iterator working, we should know the difference between the Iterables and Iterators.

- **Iterable:** An object capable of returning its members one at a time (e.g., lists, tuples).
- **Iterator:** An object representing a stream of data, returned one element at a time.

We normally use for loop to iterate through an iterable as follows –

```
for element in sequence:  
    print (element)
```

Python's built-in method `iter()` implements `__iter__()` method. It receives an iterable and returns iterator object.

## Example of Python Iterator

Following code obtains iterator object from sequence types such as list, string and tuple. The `iter()` function also returns keyiterator from dictionary.

```
print (iter("aa"))  
print (iter([1,2,3]))  
print (iter((1,2,3)))  
print (iter({}))
```

It will produce the following output –

```
<str_iterator object at 0x7fd0416b42e0>  
<list_iterator object at 0x7fd0416b42e0>  
<tuple_iterator object at 0x7fd0416b42e0>  
<dict_keyiterator object at 0x7fd041707560>
```

However, int is not iterable, hence it produces `TypeError`.

```
iterator = iter(100)
print (iterator)
```

It will produce the following output –

```
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 5, in <module>
    print (iter(100))
               ^
TypeError: 'int' object is not iterable
```

## Error Handling in Iterators

Iterator object has a method named `__next__()`. Every time it is called, it returns next element in iterator stream. Call to `next()` function is equivalent to calling `__next__()` method of iterator object.

This method which raises a `StopIteration` exception when there are no more items to return.

### Example

In the following is an example the iterator object we have created have only 3 elements and we are iterating through it more than thrice –

```
it = iter([1,2,3])
print (next(it))
print (it.__next__())
print (it.__next__())
print (next(it))
```

It will produce the following output –

```
1
2
3
Traceback (most recent call last):
  File "C:\Users\user\example.py", line 5, in <module>
    print (next(it))
               ^
StopIteration
```

This exception can be caught in the code that consumes the iterator using `try` and `except` blocks, though it's more common to handle it implicitly by using constructs like for loops which manage the `StopIteration` exception internally.

```
it = iter([1,2,3, 4, 5])
```

```

print (next(it))
while True:
    try:
        no = next(it)
        print (no)
    except StopIteration:
        break

```

It will produce the following output –

```

1
2
3
4
5

```

## Custom Iterator

A custom iterator in Python is a user-defined class that implements the iterator protocol which consists of two methods `__iter__()` and `__next__()`. This allows the class to behave like an iterator, enabling traversal through its elements one at a time.

To define a custom iterator class in Python, the class must define these methods.

### Example

In the following example, the `Oddnumbers` is a class implementing `__iter__()` and `__next__()` methods. On every call to `__next__()`, the number increments by 2 thereby streaming odd numbers in the range 1 to 10.

```

class Oddnumbers:

    def __init__(self, end_range):
        self.start = -1
        self.end = end_range

    def __iter__(self):
        return self

    def __next__(self):
        if self.start < self.end-1:
            self.start += 2
            return self.start

```

```

else:
    raise StopIteration

countiter = Oddnumbers(10)
while True:
    try:
        no = next(countiter)
        print (no)
    except StopIteration:
        break

```

It will produce the following output –

```

1
3
5
7
9

```

### Example

Let's create another iterator that generates the first n Fibonacci numbers with the following code –

```

class Fibonacci:
    def __init__(self, max_count):
        self.max_count = max_count
        self.count = 0
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_count:
            raise StopIteration

        fib_value = self.a
        self.a, self.b = self.b, self.a + self.b
        self.count += 1
        return fib_value

```

```

    return fib_value

# Using the Fibonacci iterator
fib_iterator = Fibonacci(10)

for number in fib_iterator:
    print(number)

```

It will produce the following output –

```

0
1
1
2
3
5
8
13
21
34

```

## Asynchronous Iterator

Asynchronous iterators in Python allow us to iterate over asynchronous sequences, enabling the handling of `async` operations within a loop.

They follow the asynchronous iterator protocol which consists of the methods `__aiter__()` and `__anext__()` (added in Python 3.10 version onwards.). These methods are used in conjunction with the `async for` loop to iterate over asynchronous data sources.

The `aiter()` function returns an asynchronous iterator object. It is an asynchronous counter part of the classical iterator. Any asynchronous iterator must support `__aiter__()` and `__anext__()` methods. These methods are internally called by the two built-in functions.

*Asynchronous functions are called co-routines and are executed with `asyncio.run()` method. The `main()` co-routine contains a while Loop that successively obtains odd numbers and raises `StopAsyncIteration` if the number exceeds 9.*

Like the classical iterator the asynchronous iterator gives a stream of objects. When the stream is exhausted, the `StopAsyncIteration` exception is raised.

### Example

In the example give below, an asynchronous iterator class `Oddnumbers` is declared. It implements `__aiter__()` and `__anext__()` method. On each iteration, a next odd number is returned and the program waits for one second, so that it can perform any other process asynchronously.

```
import asyncio

class Oddnumbers():
    def __init__(self):
        self.start = -1

    def __aiter__(self):
        return self

    async def __anext__(self):
        if self.start >= 9:
            raise StopAsyncIteration
        self.start += 2
        await asyncio.sleep(1)
        return self.start

async def main():
    it = Oddnumbers()
    while True:
        try:
            awaitable = anext(it)
            result = await awaitable
            print(result)
        except StopAsyncIteration:
            break

asyncio.run(main())
```

## Output

It will produce the following output –

```
1
3
5
7
9
```

# 184. Python - Generators

## Python Generators

Generators in Python are a convenient way to create iterators. They allow us to iterate through a sequence of values which means, values are generated on the fly and not stored in memory, which is especially useful for large datasets or infinite sequences.

The generator in Python is a special type of function that returns an iterator object. It appears similar to a normal Python function in that its definition also starts with def keyword. However, instead of return statement at the end, generator uses the yield keyword.

### Syntax

The following is the syntax of the generator() function –

```
def generator():
    ...
    ...
    yield obj
it = generator()
next(it)
...
```

## Creating Generators

There are two primary ways to create generators in python –

- Using Generator Functions
- Using Generator Expressions

### Using Generator Functions

The generator function uses 'yield' statement for returning the values all at a time. Each time the generator's `__next__()` method is called the generator resumes where it left off i.e. from right after the last yield statement. Here's the example of creating the generator function.

```
def count_up_to(max_value):
    current = 1
    while current <= max_value:
        yield current
        current += 1

# Using the generator
```

```
counter = count_up_to(5)
for number in counter:
    print(number)
```

**Output**

```
1
2
3
4
5
```

**Using Generator Expressions**

Generator expressions provide a compact way to create generators. They use a syntax similar to list comprehensions but used parentheses i.e. "{}" instead of square brackets i.e. "[]"

```
gen_expr = (x * x for x in range(1, 6))

for value in gen_expr:
    print(value)
```

**Output**

```
1
4
9
16
25
```

**Exception Handling in Generators**

We can create a generator and iterate it using a 'while' loop with exception handling for 'StopIteration' exception. The function in the below code is a generator that successively yield integers from 1 to 5.

When this function is called, it returns an iterator. Every call to next() method transfers the control back to the generator and fetches next integer.

```
def generator(num):
    for x in range(1, num+1):
        yield x
    return

it = generator(5)
```

```

while True:
    try:
        print (next(it))
    except StopIteration:
        break

```

**Output**

```

1
2
3
4
5

```

**Normal function vs Generator function**

Normal functions and generator functions in Python serve different purposes and exhibit distinct behaviors. Understanding their differences is essential for leveraging them effectively in our code.

A normal function computes and returns a single value or a set of values whether in a list or tuple, when called. Once it returns, the function's execution is complete and all local variables are discarded whereas a generator function yields values one at a time by suspending and resuming its state between each yield. It uses the `yield` statement instead of `return`.

**Example**

In this example we are creating a normal function and build a list of Fibonacci numbers and then iterate the list using a loop –

```

def fibonacci(n):
    fibo = []
    a, b = 0, 1
    while True:
        c=a+b
        if c>=n:
            break
        fibo.append(c)
        a, b = b, c
    return fibo
f = fibonacci(10)
for i in f:
    print (i)

```

**Output**

```
1
2
3
5
8
```

**Example**

In the above example we created a fibonacci series using the normal function and When we want to collect all Fibonacci series numbers in a list and then the list is traversed using a loop. Imagine that we want Fibonacci series going upto a large number.

In such cases, all the numbers must be collected in a list requiring huge memory. This is where generator is useful as it generates a single number in the list and gives it for consumption. Following code is the generator-based solution for list of Fibonacci numbers —

```
def fibonacci(n):
    a, b = 0, 1
    while True:
        c=a+b
        if c>=n:
            break
        yield c
        a, b = b, c
    return

f = fibonacci(10)
while True:
    try:
        print (next(f))
    except StopIteration:
        break
```

**Output**

```
1
2
3
5
8
```

## Asynchronous Generator

An asynchronous generator is a co-routine that returns an asynchronous iterator. A co-routine is a Python function defined with `async` keyword, and it can schedule and await other co-routines and tasks.

Just like a normal generator, the asynchronous generator yields incremental item in the iterator for every call to `anext()` function, instead of `next()` function.

### Syntax

The following is the syntax of the Asynchronous Generator –

```
async def generator():
    ...
    ...
    yield obj
it = generator()
anext(it)
...
```

### Example

Following code demonstrates a coroutine generator that yields incrementing integers on every iteration of an `async for` loop.

```
import asyncio

async def async_generator(x):
    for i in range(1, x+1):
        await asyncio.sleep(1)
        yield i

async def main():
    async for item in async_generator(5):
        print(item)

asyncio.run(main())
```

### Output

```
1
2
3
4
5
```

**Example**

Let us now write an asynchronous generator for Fibonacci numbers. To simulate some asynchronous task inside the co-routine, the program calls `sleep()` method for a duration of 1 second before yielding the next number. As a result, we will get the numbers printed on the screen after a delay of one second.

```
import asyncio

async def fibonacci(n):
    a, b = 0, 1
    while True:
        c=a+b
        if c>=n:
            break
        await asyncio.sleep(1)
        yield c
        a, b = b, c
    return

async def main():
    f = fibonacci(10)
    async for num in f:
        print (num)

asyncio.run(main())
```

**Output**

```
1
2
3
5
8
```

# 185. Python - Closures

## What is a Closure?

A Python closure is a nested function which has access to a variable from an enclosing function that has finished its execution. Such a variable is not bound in the local scope. To use immutable variables (number or string), we have to use the non-local keyword.

The main advantage of Python closures is that we can help avoid the using global values and provide some form of data hiding. They are used in Python decorators.

Closures are closely related to nested functions and allow inner functions to capture and retain the enclosing function's local state, even after the outer function has finished execution. Understanding closures requires familiarity with nested functions, variable scope and how Python handles function objects.

- **Nested Functions:** In Python functions can be defined inside other functions. These are called nested functions or inner functions.
- **Accessing Enclosing Scope:** Inner functions can access variables from the enclosing i.e. outer scope. This is where closures come into play.
- **Retention of State:** When an inner function i.e. closure captures and retains variables from its enclosing scope, even if the outer function has completed execution or the scope is no longer available.

## Nested Functions

Nested functions in Python refer to the practice of defining one function inside another function. This concept allows us to organize code more effectively, encapsulate functionality and manage variable scope.

Following is the example of nested functions where functionB is defined inside functionA. Inner function is then called from inside the outer function's scope.

### Example

```
def functionA():
    print ("Outer function")
    def functionB():
        print ("Inner function")
        functionB()

    functionA()
```

### Output

```
Outer function
Inner function
```

If the outer function receives any argument, it can be passed to the inner function as in the below example.

```
def functionA(name):
    print ("Outer function")
    def functionB():
        print ("Inner function")
        print ("Hi {}".format(name))
    functionB()

functionA("Python")
```

### Output

```
Outer function
Inner function
Hi Python
```

## Variable Scope

When a closure is created i.e. an inner function that captures variables from its enclosing scope, it retains access to those variables even after the outer function has finished executing. This behavior allows closures to "remember" and manipulate the values of variables from the enclosing scope.

### Example

Following is the example of the closure with the variable scope –

```
def outer_function(x):
    y = 10

    def inner_function(z):
        return x + y + z # x and y are captured from the enclosing scope

    return inner_function

closure = outer_function(5)
result = closure(3)
print(result)
```

### Output

```
18
```

## Creating a closure

Creating a closure in Python involves defining a nested function within an outer function and returning the inner function. Closures are useful for capturing and retaining the state of variables from the enclosing scope.

### Example

In the below example, we have a `functionA` function which creates and returns another function `functionB`. The nested `functionB` function is the closure.

The outer `functionA` function returns a `functionB` function and assigns it to the `myfunction` variable. Even if it has finished its execution. However, the printer closure still has access to the `name` variable.

Following is the example of creating the closure in python –

```
def functionA(name):
    name ="New name"
    def functionB():
        print (name)
    return functionB

myfunction = functionA("My name")
myfunction()
```

### Output

```
New name
```

## nonlocal Keyword

In Python, `nonlocal` keyword allows a variable outside the local scope to be accessed. This is used in a closure to modify an immutable variable present in the scope of outer variable. Here is the example of the closure with the `nonlocal` keyword.

```
def functionA():
    counter =0
    def functionB():
        nonlocal counter
        counter+=1
        return counter
    return functionB

myfunction = functionA()

retval = myfunction()
```

```
print ("Counter:", retval)

retval = myfunction()
print ("Counter:", retval)

retval = myfunction()
print ("Counter:", retval)
```

### Output

```
Counter: 1
Counter: 2
Counter: 3
```

# 186. Python - Decorators

A Decorator in Python is a function that receives another function as argument. The argument function is the one to be decorated by decorator. The behavior of argument function is extended by the decorator without actually modifying it.

In this chapter, we will learn how to use Python decorator.

## Defining Function Decorator

Function in Python is a first order object. It means that it can be passed as argument to another function just as other data types such as number, string or list etc. It is also possible to define a function inside another function. Such a function is called nested function. Moreover, a function can return other functions as well.

The typical definition of a decorator function is as under –

```
def decorator(arg_function): #arg_function to be decorated  
    def nested_function():  
        #this wraps the arg_function and extends its behaviour  
        #call arg_function  
        arg_function()  
    return nested_function
```

Here a normal Python function –

```
def function():  
    print ("hello")
```

You can now decorate this function to extend its behaviour by passing it to decorator –

```
function=decorator(function)
```

If this function is now executed, it will show output extended by decorator.

## Examples of Python Decorators

Practice the following examples to understand the concept of Python decorators –

### Example 1

Following code is a simple example of decorator –

```
def my_function(x):  
    print("The number is=",x)  
  
def my_decorator(some_function,num):  
    def wrapper(num):
```

```

print("Inside wrapper to check odd/even")

if num%2 == 0:
    ret= "Even"
else:
    ret= "Odd!"
some_function(num)
return ret

print ("wrapper function is called")
return wrapper

no=10

my_function = my_decorator(my_function, no)
print ("It is ",my_function(no))

```

The `my_function()` just prints out the received number. However, its behaviour is modified by passing it to a `my_decorator`. The inner function receives the number and returns whether it is odd/even. Output of above code is –

```

wrapper function is called
Inside wrapper to check odd/even
The number is= 10
It is Even

```

## Example 2

An elegant way to decorate a function is to mention just before its definition, the name of decorator prepended by @ symbol. The above example is re-written using this notation –

```

def my_decorator(some_function):

    def wrapper(num):
        print("Inside wrapper to check odd/even")
        if num%2 == 0:
            ret= "Even"
        else:
            ret= "Odd!"
        some_function(num)
        return ret

    print ("wrapper function is called")
    return wrapper

@my_decorator

```

```
def my_function(x):
    print("The number is=",x)
no=10
print ("It is ",my_function(no))
```

Python's standard library defines following built-in decorators –

### **@classmethod Decorator**

The `classmethod` is a built-in function. It transforms a method into a class method. A class method is different from an instance method. Instance method defined in a class is called by its object. The method received an implicit object referred to by `self`. A class method on the other hand implicitly receives the class itself as first argument.

#### **Syntax**

In order to declare a class method, the following notation of decorator is used –

```
class Myclass:
    @classmethod
    def mymethod(cls):
        #....
```

The `@classmethod` form is that of function decorator as described earlier. The `mymethod` receives reference to the class. It can be called by the class as well as its object. That means `Myclass.mymethod` as well as `Myclass().mymethod` both are valid calls.

#### **Example of @classmethod Decorator**

Let us understand the behaviour of class method with the help of following example –

```
class counter:
    count=0
    def __init__(self):
        print ("init called by ", self)
        counter.count=counter.count+1
        print ("count=",counter.count)
    @classmethod
    def showcount(cls):
        print ("called by ",cls)
        print ("count=",cls.count)

c1=counter()
c2=counter()
print ("class method called by object")
c1.showcount()
```

```
print ("class method called by class")
counter.showcount()
```

In the class definition count is a class attribute. The `__init__()` method is the constructor and is obviously an instance method as it received `self` as object reference. Every object declared calls this method and increments count by 1.

The `@classmethod` decorator transforms `showcount()` method into a class method which receives reference to the class as argument even if it is called by its object. It can be seen even when `c1` object calls `showcount`, it displays reference of `counter` class.

It will display the following output –

```
init called by <__main__.counter object at 0x000001D32DB4F0F0>
count= 1

init called by <__main__.counter object at 0x000001D32DAC8710>
count= 2

class method called by object
called by <class '__main__.counter'>
count= 2

class method called by class
called by <class '__main__.counter'>
```

## **@staticmethod Decorator**

The `staticmethod` is also a built-in function in Python standard library. It transforms a method into a static method. Static method doesn't receive any reference argument whether it is called by instance of class or class itself. Following notation used to declare a static method in a class –

### **Syntax**

```
class Myclass:
    @staticmethod
    def mymethod():
        #....
```

Even though `Myclass.mymethod` as well as `Myclass().mymethod` both are valid calls, the static method receives reference of neither.

### **Example of @staticmethod Decorator**

The counter class is modified as under –

```
class counter:
    count=0
    def __init__(self):
        print ("init called by ", self)
        counter.count=counter.count+1
```

```

        print ("count=",counter.count)

    @staticmethod
    def showcount():
        print ("count=",counter.count)

c1=counter()
c2=counter()
print ("class method called by object")
c1.showcount()
print ("class method called by class")
counter.showcount()

```

As before, the class attribute count is increment on declaration of each object inside the `__init__()` method. However, since `mymethod()`, being a static method doesn't receive either self or cls parameter. Hence value of class attribute count is displayed with explicit reference to counter.

The output of the above code is as below –

```

init called by <__main__.counter object at 0x000002512EDCF0B8>
count= 1
init called by <__main__.counter object at 0x000002512ED48668>
count= 2
class method called by object
count= 2
class method called by class
count= 2

```

## @property Decorator

Python's `property()` built-in function is an interface for accessing instance variables of a class. The `@property` decorator turns an instance method into a "getter" for a read-only attribute with the same name, and it sets the docstring for the property to "Get the current value of the instance variable."

You can use the following three decorators to define a property –

- **`@property`** – Declares the method as a property.
- **`@<property-name>.setter`** – Specifies the setter method for a property that sets the value to a property.
- **`@<property-name>.deleter`** – Specifies the delete method as a property that deletes a property.

A property object returned by `property()` function has `getter`, `setter`, and `delete` methods.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

The fget argument is the getter method, fset is setter method. It optionally can have fdel as method to delete the object and doc is the documentation string.

### Syntax

The property() object's setter and getter may also be assigned with the following syntax also.

```
speed = property()
speed=f.getter(speed, get_speed)
speed=f.setter(speed, set_speed)
```

Where get\_speed() and set\_speeds() are the instance methods that retrieve and set the value to an instance variable speed in Car class.

The above statements can be implemented by @property decorator. Using the decorator car class is re-written as –

### Example of @property Decorator

```
class car:
    def __init__(self, speed=40):
        self._speed=speed
        return
    @property
    def speed(self):
        return self._speed
    @speed.setter
    def speed(self, speed):
        if speed<0 or speed>100:
            print ("speed limit 0 to 100")
        return
        self._speed=speed
        return

c1=car()
print (c1.speed) #calls getter
c1.speed=60 #calls setter
```

Property decorator is very convenient and recommended method of handling instance attributes.

# 187. Python - Recursion

Recursion is a fundamental programming concept where a function calls itself in order to solve a problem. This technique breaks down a complex problem into smaller and more manageable sub-problems of the same type. In Python, recursion is implemented by defining a function that makes one or more calls to itself within its own body.

## Components of Recursion

As we discussed before Recursion is a technique where a function calls itself. Here for understanding recursion, it's required to know its key components. Following are the primary components of the recursion –

- Base Case
- Recursive Case

### Base Case

The Base case is a fundamental concept in recursion, if serving as the condition under which a recursive function stops calling itself. It is essential for preventing infinite recursion and subsequent stack overflow errors.

The base case provides a direct solution to the simplest instance of the problem ensuring that each recursive call gets closer to this terminating condition.

The most popular example of recursion is calculation of factorial. Mathematically factorial is defined as –

$$n! = n \times (n-1)!$$

It can be seen that we use factorial itself to define factorial. Hence this is a fit case to write a recursive function. Let us expand above definition for calculation of factorial value of 5.

$$\begin{aligned} 5! &= 5 \times 4! \\ &= 5 \times 4 \times 3! \\ &= 5 \times 4 \times 3 \times 2! \\ &= 5 \times 4 \times 3 \times 2 \times 1! \\ &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 120 \end{aligned}$$

While we can perform this calculation using a loop, its recursive function involves successively calling it by decrementing the number till it reaches 1.

### Example

The following example shows how you can use a recursive function to calculate factorial –

```
def factorial(n):  
  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n-1)
```

```

print (n)
return 1 #base case
else:
    print (n,'*', end=' ')
    return n * factorial(n-1) #Recursive case

print ('factorial of 5=', factorial(5))

```

The above programs generates the following output –

```

5 * 4 * 3 * 2 * 1
factorial of 5= 120

```

## Recursive Case

The recursive case is the part of a recursive function where the function calls itself to solve a smaller or simpler instance of the same problem. This mechanism allows a complex problem to be broken down into more manageable sub-problems where each of them is a smaller version of the original problem.

The recursive case is essential for progressing towards the base case, ensuring that the recursion will eventually terminate.

## Example

Following is the example of the Recursive case. In this example we are generating the Fibonacci sequence in which the recursive case sums the results of the two preceding Fibonacci numbers –

```

def fibonacci(n):
    if n <= 0:
        return 0 # Base case for n = 0
    elif n == 1:
        return 1 # Base case for n = 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2) # Recursive case

fib_series = [fibonacci(i) for i in range(6)]
print(fib_series)

```

The above program generates the following output –

```
[0, 1, 1, 2, 3, 5]
```

## Binary Search using Recursion

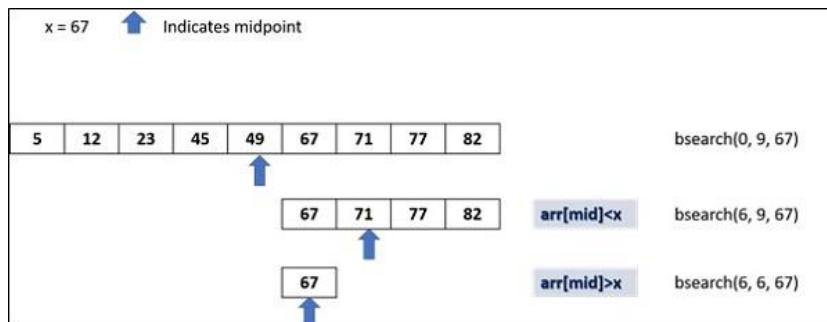
Binary search is a powerful algorithm for quickly finding elements in sorted lists, with logarithmic time complexity making it highly efficient.

Let us have a look at another example to understand how recursion works. The problem at hand is to check whether a given number is present in a list.

While we can perform a sequential search for a certain number in the list using a for loop and comparing each number, the sequential search is not efficient especially if the list is too large. The binary search algorithm that checks if the index 'high' is greater than index 'low'. Based on value present at 'mid' variable, the function is called again to search for the element.

We have a list of numbers, arranged in ascending order. The we find the midpoint of the list and restrict the checking to either left or right of midpoint depending on whether the desired number is less than or greater than the number at midpoint.

The following diagram shows how binary search works –



### Example

The following code implements the recursive binary searching technique –

```
def bsearch(my_list, low, high, elem):
    if high >= low:
        mid = (high + low) // 2
        if my_list[mid] == elem:
            return mid
        elif my_list[mid] > elem:
            return bsearch(my_list, low, mid - 1, elem)
        else:
            return bsearch(my_list, mid + 1, high, elem)
    else:
        return -1

my_list = [5,12,23, 45, 49, 67, 71, 77, 82]
num = 67
print("The list is")
print(my_list)
```

```
print ("Check for number:", num)
my_result = bsearch(my_list,0,len(my_list)-1,num)

if my_result != -1:
    print("Element found at index ", str(my_result))
else:
    print("Element not found!")
```

### Output

```
The list is
[5, 12, 23, 45, 49, 67, 71, 77, 82]
Check for number: 67
Element found at index 5
```

# 188. Python - Regular Expressions

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expression are popularly known as regex or regexp.

Usually, such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

Large scale text processing in data science projects requires manipulation of textual data. The regular expressions processing is supported by many programming languages including Python. Python's standard library has **re** module for this purpose.

Since most of the functions defined in **re** module work with raw strings, let us first understand what the raw strings are.

## Raw Strings

Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning. Python on the other hand uses the same character as escape character. Hence Python uses the raw string notation.

A string become a raw string if it is prefixed with r or R before the quotation symbols. Hence 'Hello' is a normal string were are r'Hello' is a raw string.

```
>>> normal="Hello"  
>>> print (normal)  
Hello  
>>> raw=r"Hello"  
>>> print (raw)  
Hello
```

In normal circumstances, there is no difference between the two. However, when the escape character is embedded in the string, the normal string actually interprets the escape sequence, whereas the raw string doesn't process the escape character.

```
>>> normal="Hello\nWorld"  
>>> print (normal)  
Hello  
World  
>>> raw=r"Hello\nWorld"  
>>> print (raw)  
Hello\nWorld
```

In the above example, when a normal string is printed the escape character '\n' is processed to introduce a newline. However, because of the raw string operator 'r' the effect of escape character is not translated as per its meaning.

## Metacharacters

Most letters and characters will simply match themselves. However, some characters are special metacharacters, and don't match themselves. Meta characters are characters having a special meaning, similar to \* in wild card.

Here's a complete list of the metacharacters –

|   |          |    |   |   |   |   |   |   |   |   |  |   |   |
|---|----------|----|---|---|---|---|---|---|---|---|--|---|---|
| . | $\wedge$ | \$ | * | + | ? | { | } | [ | ] | \ |  | ( | ) |
|---|----------|----|---|---|---|---|---|---|---|---|--|---|---|

The square bracket symbols [ and ] indicate a set of characters that you wish to match. Characters can be listed individually, or as a range of characters separating them by a '-'.

| Sr.No. | Metacharacters & Description                                                         |
|--------|--------------------------------------------------------------------------------------|
| 1      | [abc]<br>match any of the characters a, b, or c                                      |
|        | [a-c]<br>which uses a range to express the same set of characters.                   |
| 3      | [a-z]<br>match only lowercase letters.                                               |
|        | [0-9]<br>match only digits.                                                          |
| 5      | '^'<br>complements the character set in [].[^5] will match any character except '5'. |

'\ is an escaping metacharacter. When followed by various characters it forms various special sequences. If you need to match a [ or \, you can precede them with a backslash to remove their special meaning: \[ or \\.

Predefined sets of characters represented by such special sequences beginning with '\' are listed below –

| Sr.No. | Metacharacters & Description                                                               |
|--------|--------------------------------------------------------------------------------------------|
| 1      | \d<br>Matches any decimal digit; this is equivalent to the class [0-9].                    |
|        | \D<br>Matches any non-digit character; this is equivalent to the class [^0-9].             |
| 3      | \s<br>Matches any whitespace character; this is equivalent to the class [\t\n\r\f\v].      |
|        | \S<br>Matches any non-whitespace character; this is equivalent to the class [^\t\n\r\f\v]. |

|    |                                                                                                                |
|----|----------------------------------------------------------------------------------------------------------------|
|    | \w                                                                                                             |
| 5  | Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_].                              |
| 6  | \W                                                                                                             |
| 6  | Matches any non-alphanumeric character. equivalent to the class [^a-zA-Z0-9_].                                 |
| 7  | .                                                                                                              |
| 7  | Matches with any single character except newline '\n'.                                                         |
| 8  | ?                                                                                                              |
| 8  | match 0 or 1 occurrence of the pattern to its left                                                             |
| 9  | +                                                                                                              |
| 9  | 1 or more occurrences of the pattern to its left                                                               |
| 10 | *                                                                                                              |
| 10 | 0 or more occurrences of the pattern to its left                                                               |
| 11 | \b                                                                                                             |
| 11 | boundary between word and non-word and /B is opposite of /b                                                    |
| 12 | [..]                                                                                                           |
| 12 | Matches any single character in a square bracket and [^..] matches any single character not in square bracket. |
| 13 | \                                                                                                              |
| 13 | It is used for special meaning characters like \. to match a period or \+ for plus sign.                       |
| 14 | {n,m}                                                                                                          |
| 14 | Matches at least n and at most m occurrences of preceding                                                      |
| 15 | a  b                                                                                                           |
| 15 | Matches either a or b                                                                                          |

Python's re module provides useful functions for finding a match, searching for a pattern, and substitute a matched string with other string etc.

## The `re.match()` Function

This function attempts to match RE pattern at the start of string with optional flags. Following is the syntax for this function –

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters –

| Sr.No. | Parameter & Description                       |
|--------|-----------------------------------------------|
| 1      | <b>pattern</b>                                |
| 1      | This is the regular expression to be matched. |
| 2      | <b>String</b>                                 |

|   |                                                                                                                                 |
|---|---------------------------------------------------------------------------------------------------------------------------------|
|   | This is the string, which would be searched to match the pattern at the beginning of string.                                    |
| 3 | <b>Flags</b><br>You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below. |

The `re.match()` function returns a match object on success, `None` on failure. A match object instance contains information about the match: where it starts and ends, the substring it matched, etc.

The match object's `start()` method returns the starting position of pattern in the string, and `end()` returns the endpoint.

If the pattern is not found, the match object is `None`.

We use `group(num)` or `groups()` function of match object to get matched expression.

| Sr.No. | Match Object Methods & Description                                                                 |
|--------|----------------------------------------------------------------------------------------------------|
| 1      | <b>group(num=0)</b> This method returns entire match (or specific subgroup num)                    |
| 2      | <b>groups()</b> This method returns all matching subgroups in a tuple (empty if there weren't any) |

### Example

```
import re

line = "Cats are smarter than dogs"
matchObj = re.match( r'Cats', line)

print (matchObj.start(), matchObj.end())
print ("matchObj.group() : ", matchObj.group())
```

It will produce the following output –

```
0 4
matchObj.group() : Cats
```

### The `re.search()` Function

This function searches for first occurrence of RE pattern within the string, with optional flags. Following is the syntax for this function –

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters –

| Sr.No. | Parameter & Description                                                                                   |
|--------|-----------------------------------------------------------------------------------------------------------|
| 1      | <b>Pattern</b><br>This is the regular expression to be matched.                                           |
| 2      | <b>String</b><br>This is the string, which would be searched to match the pattern anywhere in the string. |

| <b>Flags</b> |                                                                                                                 |
|--------------|-----------------------------------------------------------------------------------------------------------------|
| 3            | You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below. |

The `re.search` function returns a match object on success, none on failure. We use `group(num)` or `groups()` function of match object to get the matched expression.

| Sr.No. | Match Object Methods & Description                                                                 |
|--------|----------------------------------------------------------------------------------------------------|
| 1      | <b>group(num=0)</b> This method returns entire match (or specific subgroup num)                    |
| 2      | <b>groups()</b> This method returns all matching subgroups in a tuple (empty if there weren't any) |

### Example

```
import re

line = "Cats are smarter than dogs"
matchObj = re.search( r'than', line)
print (matchObj.start(), matchObj.end())
print ("matchObj.group() : ", matchObj.group())
```

It will produce the following output –

```
17 21
matchObj.group() : than
```

### Matching Vs Searching

Python offers two different primitive operations based on regular expressions, `match` checks for a match only at the beginning of the string, while `search` checks for a match anywhere in the string (this is what Perl does by default).

### Example

```
import re

line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print ("match --> matchObj.group() : ", matchObj.group())
else:
    print ("No match!!")
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
    print ("search --> searchObj.group() : ", searchObj.group())
else:
```

```
print ("Nothing found!!")
```

When the above code is executed, it produces the following output –

```
No match!
search --> matchObj.group() : dogs
```

## The `re.findall()` Function

The `findall()` function returns all non-overlapping matches of pattern in string, as a list of strings or tuples. The string is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

### Syntax

```
re.findall(pattern, string, flags=0)
```

### Parameters

| Sr.No. | Parameter & Description                                                                                                         |
|--------|---------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>Pattern</b><br>This is the regular expression to be matched.                                                                 |
| 2      | <b>String</b><br>This is the string, which would be searched to match the pattern anywhere in the string.                       |
| 3      | <b>Flags</b><br>You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below. |

### Example

```
import re
string="Simple is better than complex."
obj=re.findall(r"ple", string)
print (obj)
```

It will produce the following output –

```
['ple', 'ple']
```

Following code obtains the list of words in a sentence with the help of `findall()` function.

```
import re
string="Simple is better than complex."
obj=re.findall(r"\w*", string)
print (obj)
```

It will produce the following output –

```
['Simple', '', 'is', '', 'better', '', 'than', '', 'complex', '', '']
```

## The `re.sub()` Function

One of the most important `re` methods that use regular expressions is `sub`.

### Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE pattern in `string` with `repl`, substituting all occurrences unless `max` is provided. This method returns modified string.

### Example

```
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print ("Phone Num : ", num)

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print ("Phone Num : ", num)
```

It will produce the following output –

```
Phone Num : 2004-959-559
Phone Num : 2004959559
```

### Example

The following example uses `sub()` function to substitute all occurrences of `is` with `was` word –

```
import re

string="Simple is better than complex. Complex is better than complicated."
obj=re.sub(r'is', r'was',string)
print (obj)
```

It will produce the following output –

```
Simple was better than complex. Complex was better than complicated.
```

## The `re.compile()` Function

The `compile()` function compiles a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods.

### Syntax

```
re.compile(pattern, flags=0)
```

## Flags

| Sr.No. | Modifier & Description                                                                                                                                                              |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>re.I</b><br>Performs case-insensitive matching.                                                                                                                                  |
| 2      | <b>re.L</b><br>Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).       |
| 3      | <b>re.</b><br>M Makes \$ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).                       |
| 4      | <b>re.S</b><br>Makes a period (dot) match any character, including a newline.                                                                                                       |
| 5      | <b>re.U</b><br>Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.                                                         |
| 6      | <b>re.X</b><br>Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker. |

The sequence –

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to –

```
result = re.match(pattern, string)
```

But using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

### Example

```
import re

string="Simple is better than complex. Complex is better than complicated."
pattern=re.compile(r'is')
obj=pattern.match(string)
obj=pattern.search(string)
print (obj.start(), obj.end())

obj=pattern.findall(string)
```

```
print (obj)

obj=pattern.sub(r'was', string)
print (obj)
```

It will produce the following output –

```
7 9
['is', 'is']
Simple was better than complex. Complex was better than complicated.
```

## The `re.finditer()` Function

This function returns an iterator yielding match objects over all non-overlapping matches for the RE pattern in string.

### Syntax

```
re.finditer(pattern, string, flags=0)
```

### Example

```
import re

string="Simple is better than complex. Complex is better than complicated."
pattern=re.compile(r'is')
iterator = pattern.finditer(string)
print (iterator )

for match in iterator:
    print(match.span())
```

It will produce the following output –

```
(7, 9)
(39, 41)
```

## Use Cases of Python Regex

### Finding all Adverbs

The `findall()` function matches all occurrences of a pattern, not just the first one as `search()` does. For example, if a writer wanted to find all of the adverbs in some text, they might use `findall()` in the following manner –

```
import re

text = "He was carefully disguised but captured quickly by police."
obj = re.findall(r"\w+ly\b", text)
```

```
print (obj)
```

It will produce the following output –

```
['carefully', 'quickly']
```

### Finding words starting with vowels

```
import re
text = 'Errors should never pass silently. Unless explicitly silenced.'
obj=re.findall(r'\b[aeiouAEIOU]\w+', text)
print (obj)
```

It will produce the following output –

```
['Errors', 'Unless', 'explicitly']
```

### Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these –

| Sr.No. | Modifier & Description                                                                                                                                                               |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>re.I</b><br>Performs case-insensitive matching.                                                                                                                                   |
| 2      | <b>re.L</b><br>Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).         |
| 3      | <b>re.M</b><br>Makes \$ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).                         |
| 4      | <b>re.S</b><br>Makes a period (dot) match any character, including a newline.                                                                                                        |
| 5      | <b>re.U</b><br>Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.                                                          |
| 6      | <b>re.X</b><br>Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set []) or when escaped by a backslash) and treats unescaped # as a comment marker. |

## Regular Expression Patterns

Except for control characters, (+ ? . \* ^ \$ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

| Sr.No. | Pattern & Description                                                                                                                    |
|--------|------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>^</b><br>Matches beginning of line.                                                                                                   |
| 2      | <b>\$</b><br>Matches end of line.                                                                                                        |
| 3      | <b>.</b><br>Matches any single character except newline. Using m option allows it to match newline as well.                              |
| 4      | <b>[...]</b><br>Matches any single character in brackets.                                                                                |
| 5      | <b>[^...]</b><br>Matches any single character not in brackets                                                                            |
| 6      | <b>re*</b><br>Matches 0 or more occurrences of preceding expression.                                                                     |
| 7      | <b>re+</b><br>Matches 1 or more occurrence of preceding expression.                                                                      |
| 8      | <b>re?</b><br>Matches 0 or 1 occurrence of preceding expression.                                                                         |
| 9      | <b>re{ n }</b><br>Matches exactly n number of occurrences of preceding expression.                                                       |
| 10     | <b>re{ n, }</b><br>Matches n or more occurrences of preceding expression.                                                                |
| 11     | <b>re{ n, m }</b><br>Matches at least n and at most m occurrences of preceding expression.                                               |
| 12     | <b>a  b</b><br>Matches either a or b.                                                                                                    |
| 13     | <b>(re)</b><br>Groups regular expressions and remembers matched text.                                                                    |
| 14     | <b>(?imx)</b><br>Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.   |
| 15     | <b>(?-imx)</b><br>Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| 16     | <b>(?: re)</b><br>Groups regular expressions without remembering matched text.                                                           |
| 17     | <b>(?imx: re)</b><br>Temporarily toggles on i, m, or x options within parentheses.                                                       |
| 18     | <b>(?-imx: re)</b><br>Temporarily toggles off i, m, or x options within parentheses.                                                     |
| 19     | <b>(?#...)</b>                                                                                                                           |

|    |                                                                                                                                          |
|----|------------------------------------------------------------------------------------------------------------------------------------------|
|    | Comment.                                                                                                                                 |
| 20 | <b>(?= re)</b><br>Specifies position using a pattern. Doesn't have a range.                                                              |
| 21 | <b>(?! re)</b><br>Specifies position using pattern negation. Doesn't have a range.                                                       |
| 22 | <b>(?&gt; re)</b><br>Matches independent pattern without backtracking.                                                                   |
| 23 | <b>\w</b><br>Matches word characters.                                                                                                    |
| 24 | <b>\W</b><br>Matches nonword characters.                                                                                                 |
| 25 | <b>\s</b><br>Matches whitespace. Equivalent to [\t\n\r\f].                                                                               |
| 26 | <b>\S</b><br>Matches nonwhitespace.                                                                                                      |
| 27 | <b>\d</b><br>Matches digits. Equivalent to [0-9].                                                                                        |
| 28 | <b>\D</b><br>Matches nondigits.                                                                                                          |
| 29 | <b>\A</b><br>Matches beginning of string.                                                                                                |
| 30 | <b>\Z</b><br>Matches end of string. If a newline exists, it matches just before newline.                                                 |
| 31 | <b>\z</b><br>Matches end of string.                                                                                                      |
| 32 | <b>\G</b><br>Matches point where last match finished.                                                                                    |
| 33 | <b>\b</b><br>Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.                               |
| 34 | <b>\B</b><br>Matches nonword boundaries.                                                                                                 |
| 35 | <b>\n, \t, etc.</b><br>Matches newlines, carriage returns, tabs, etc.                                                                    |
| 36 | <b>\1... \9</b><br>Matches nth grouped subexpression.                                                                                    |
| 37 | <b>\10</b><br>Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code. |

## Regular Expression Examples

### Literal characters

| Sr.No. | Example & Description |
|--------|-----------------------|
| 1      | <b>python</b>         |

|  |                 |
|--|-----------------|
|  | Match "python". |
|--|-----------------|

## Character classes

| Sr.No. | Example & Description                                          |
|--------|----------------------------------------------------------------|
| 1      | <b>[Pp]ython</b><br>Match "Python" or "python"                 |
| 2      | <b>rub[ye]</b><br>Match "ruby" or "rube"                       |
| 3      | <b>[aeiou]</b><br>Match any one lowercase vowel                |
| 4      | <b>[0-9]</b><br>Match any digit; same as [0123456789]          |
| 5      | <b>[a-z]</b><br>Match any lowercase ASCII letter               |
| 6      | <b>[A-Z]</b><br>Match any uppercase ASCII letter               |
| 7      | <b>[a-zA-Z0-9]</b><br>Match any of the above                   |
| 8      | <b>[^aeiou]</b><br>Match anything other than a lowercase vowel |
| 9      | <b>[^0-9]</b><br>Match anything other than a digit             |

## Special Character Classes

| Sr.No. | Example & Description                                     |
|--------|-----------------------------------------------------------|
| 1      | <b>.</b><br>Match any character except newline            |
| 2      | <b>\d</b><br>Match a digit: [0-9]                         |
| 3      | <b>\D</b><br>Match a nondigit: [^0-9]                     |
| 4      | <b>\s</b><br>Match a whitespace character: [<br>\t\r\n\f] |
| 5      | <b>\S</b><br>Match nonwhitespace: [^ \t\r\n\f]            |
| 6      | <b>\w</b><br>Match a single word character: [A-Za-z0-9_]  |
| 7      | <b>\W</b><br>Match a nonword character: [^A-Za-z0-9_]     |

## Repetition Cases

| Sr.No. | Example & Description |
|--------|-----------------------|
|--------|-----------------------|

|   |                                          |
|---|------------------------------------------|
| 1 | <b>ruby?</b>                             |
|   | Match "rub" or "ruby": the y is optional |
| 2 | <b>ruby*</b>                             |
|   | Match "rub" plus 0 or more ys            |
| 3 | <b>ruby+</b>                             |
|   | Match "rub" plus 1 or more ys            |
| 4 | <b>\d{3}</b>                             |
|   | Match exactly 3 digits                   |
| 5 | <b>\d{3,}</b>                            |
|   | Match 3 or more digits                   |
| 6 | <b>\d{3,5}</b>                           |
|   | Match 3, 4, or 5 digits                  |

## Nongreedy repetition

This matches the smallest number of repetitions –

| Sr.No. | Example & Description                                                                                               |
|--------|---------------------------------------------------------------------------------------------------------------------|
| 1      | <b>&lt;.*&gt;</b><br>Greedy repetition: matches " <code>&lt;python&gt;perl&gt;</code> "                             |
| 2      | <b>&lt;.*?&gt;</b><br>Nongreedy: matches " <code>&lt;python&gt;</code> " in " <code>&lt;python&gt;perl&gt;</code> " |

## Grouping with Parentheses

| Sr.No. | Example & Description                                                      |
|--------|----------------------------------------------------------------------------|
| 1      | <b>\D\d+</b><br>No group: + repeats \d                                     |
| 2      | <b>(\D\d)+</b><br>Grouped: + repeats \D\d pair                             |
| 3      | <b>([Pp]ython(, )?)+</b><br>Match "Python", "Python, python, python", etc. |

## Backreferences

This matches a previously matched group again –

| Sr.No. | Example & Description                                                                                                                               |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>([Pp])ython&amp;\1ails</b><br>Match python&pails or Python&Pails                                                                                 |
| 2      | <b>([""])[^\1]*\1</b><br>Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc. |

## Alternatives

| Sr.No. | Example & Description                                               |
|--------|---------------------------------------------------------------------|
| 1      | <b>python perl</b><br>Match "python" or "perl"                      |
| 2      | <b>rub(y le)</b><br>Match "ruby" or "ruble"                         |
| 3      | <b>Python(!+ \?)</b><br>"Python" followed by one or more ! or one ? |

## Anchors

This needs to specify match position.

| Sr.No. | Example & Description                                                                    |
|--------|------------------------------------------------------------------------------------------|
| 1      | <b>^Python</b><br>Match "Python" at the start of a string or internal line               |
| 2      | <b>Python\$</b><br>Match "Python" at the end of a string or line                         |
| 3      | <b>\APython</b><br>Match "Python" at the start of a string                               |
| 4      | <b>Python\Z</b><br>Match "Python" at the end of a string                                 |
| 5      | <b>\bPython\b</b><br>Match "Python" at a word boundary                                   |
| 6      | <b>\brub\B</b><br>\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone |
| 7      | <b>Python(?!=!)</b><br>Match "Python", if followed by an exclamation point.              |
| 8      | <b>Python(?!=!)</b><br>Match "Python", if not followed by an exclamation point.          |

## Special Syntax with Parentheses

| Sr.No. | Example & Description                                               |
|--------|---------------------------------------------------------------------|
| 1      | <b>R(?#comment)</b><br>Matches "R". All the rest is a comment       |
| 2      | <b>R(?i)uby</b><br>Case-insensitive while matching "uby"            |
| 3      | <b>R(?i:uby)</b><br>Same as above                                   |
| 4      | <b>rub(?:y le))</b><br>Group only without creating \1 backreference |

# 189. Python - PIP

In Python, pip is the standard package management system used to install and manage software packages written in Python. It allows you to easily install libraries and frameworks to extend the functionality of Python applications. pip comes bundled with Python, starting from Python version 3.4 and above.

## Installing pip

If you are using Python 3.4 or above, pip is already included. However, if you don't have pip installed, you can install it using the following steps –

- Download get-pip.py script –

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

- Run the Script

```
python get-pip.py
```

## Installing Packages with pip

You can use pip to install any package from the Python Package Index (PyPI), which is the official third-party software repository for Python.

*PyPI hosts thousands of packages that you can easily integrate into your projects. These packages range from essential libraries for scientific computing, such as numpy and pandas, to web development frameworks like Django and Flask, and many more.*

## Syntax

Following is the basic syntax to install packages with pip in Python –

```
pip install package_name
```

## Example

To install the requests library, you can use the following command –

```
pip install requests
```

## Example: Specifying Versions

Sometimes, you may need a specific version of a package to ensure compatibility with your project. You can specify the version by using the == operator –

```
pip install requests==2.25.1
```

## Example: Installing Multiple Packages

You can also install multiple packages at once by listing their names separated by spaces –

```
pip install numpy pandas matplotlib
```

## Upgrading Packages

To upgrade a package to the latest version, you can use the --upgrade option with the pip install command.

### Syntax

Following is the basic syntax to upgrade a package in Python –

```
pip install --upgrade package_name
```

### Example

To upgrade the requests library, you can use the following command –

```
pip install --upgrade requests
```

## Listing Installed Packages

You can list all the installed packages in your Python environment using the pip list command.

When working on Python projects, it is often necessary to know which packages and versions are installed in your environment. pip provides several commands to list and manage installed packages.

### Basic Listing

To list all installed packages in your current environment, use the following command –

```
pip list
```

This command outputs a list of all installed packages along with their respective versions. This is useful for quickly checking the state of your environment.

### Detailed Information

For more detailed information about each installed package, you can use the pip show command followed by the package name –

```
pip show requests
```

This command displays detailed information about the specified package, including –

- Name
- Version
- Summary
- Home-page
- Author
- Author-email
- License
- Location
- Requires
- Required-by

## Outdated Packages

To check for outdated packages in your environment, you can use the following command –

```
pip list --outdated
```

This command lists all installed packages that have newer versions available. The output includes the current version and the latest version available.

## Uninstalling Packages

To uninstall a package, you can use the pip uninstall command.

When you no longer need a Python package in your environment, you can uninstall it using pip. Here is how you can uninstall packages –

### Uninstalling a Single Package

To uninstall a single package, use the pip uninstall command followed by the package name. For example, to uninstall the requests package –

```
pip uninstall requests
```

You will be prompted to confirm the uninstallation. Type y and press "Enter" to proceed.

### Uninstalling Multiple Packages

You can also uninstall multiple packages in a single command by listing them all after pip uninstall –

```
pip uninstall numpy pandas
```

This command will uninstall both numpy and pandas packages.

## Freezing Installed Packages

Freezing installed packages in Python refers to generating a list of all packages installed in your environment along with their versions. This list is saved to a "requirements.txt" file and can be used to recreate the exact environment elsewhere.

### Using "pip freeze"

The pip freeze command lists all installed packages and their versions. You can direct its output to a "requirements.txt" file using the shell redirection > operator –

```
pip freeze > requirements.txt
```

This command creates or overwrites "requirements.txt" with a list of packages and versions in the format "package==version".

## Using a requirements.txt File

A requirements.txt file is a way to specify a list of packages to be installed using pip. This is useful for ensuring that all dependencies are installed for a project.

## Creating requirements.txt

To create a "requirements.txt" file with the current environment's packages, you can use the following command –

```
pip freeze > requirements.txt
```

## Installing from requirements.txt

To install all packages listed in a requirements.txt file, you can use the following command –

```
pip install -r requirements.txt
```

## Using Virtual Environments

Virtual environments allow you to create isolated Python environments for different projects. This ensures that dependencies for different projects do not interfere with each other.

### Creating a Virtual Environment

You can create a virtual environment using the following command –

```
python -m venv myenv
```

Replace myenv with your preferred name for the virtual environment. This command creates a directory named myenv (or your specified name) containing a self-contained Python environment.

### Activating the Virtual Environment

Depending on your operating system, activate the virtual environment –

- On Windows –

```
myenv\Scripts\activate
```

- On macOS and Linux –

```
source myenv/bin/activate
```

Once activated, your command prompt will change to show the name of the virtual environment (myenv in this case), indicating that you are now working within it.

### Deactivating the Virtual Environment

To deactivate the virtual environment and return to the global Python environment, you can use the following command –

```
deactivate
```

### Deleting the Virtual Environment

If you no longer need the virtual environment, simply delete its directory (myenv or your chosen name) using the following command –

```
rm -rf myenv # On macOS and Linux  
rmdir /s myenv # On Windows
```

# 190. Python - Database Access

## Database Access in Python

Database access in Python is used to interact with databases, allowing applications to store, retrieve, update, and manage data consistently. Various relational database management systems (RDBMS) are supported for these tasks, each requiring specific Python packages for connectivity –

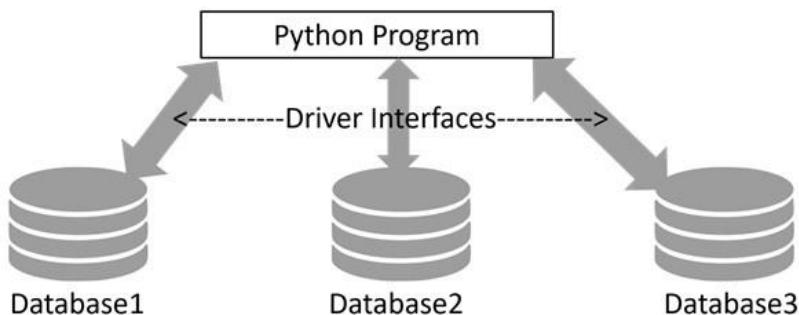
- GadFly
- MySQL
- PostgreSQL
- Microsoft SQL Server
- Informix
- Oracle
- Sybase
- SQLite
- and many more...

Data input and generated during execution of a program is stored in RAM. If it is to be stored persistently, it needs to be stored in database tables.

Relational databases use SQL (Structured Query Language) for performing INSERT/ DELETE/ UPDATE operations on the database tables. However, implementation of SQL varies from one type of database to other. This raises incompatibility issues. SQL instructions for one database do not match with other.

## DB-API (Database API)

To address this issue of compatibility, Python Enhancement Proposal (PEP) 249 introduced a standardized interface known as DB-API. This interface provides a consistent framework for database drivers, ensuring uniform behavior across different database systems. It simplifies the process of transitioning between various databases by establishing a common set of rules and methods.



## Using SQLite with Python

Python's standard library includes `sqlite3` module, a DB-API compatible driver for SQLite3 database. It serves as a reference implementation for DB-API. For other types of databases, you will have to install the relevant Python package –

| Database   | Python Package                  |
|------------|---------------------------------|
| Oracle     | cx_oracle, pyodbc               |
| SQL Server | pymssql, pyodbc                 |
| PostgreSQL | psycopg2                        |
| MySQL      | MySQL Connector/Python, pymysql |

## Working with SQLite

Using SQLite with Python is very easy due to the built-in `sqlite3` module. The process involves –

- **Connection Establishment** – Create a connection object using `sqlite3.connect()`, providing necessary connection credentials such as server name, port, username, and password.
- **Transaction Management** – The connection object manages database operations, including opening, closing, and transaction control (committing or rolling back transactions).
- **Cursor Object** – Obtain a cursor object from the connection to execute SQL queries. The cursor serves as the gateway for CRUD (Create, Read, Update, Delete) operations on the database.

In this tutorial, we shall learn how to access database using Python, how to store data of Python objects in a SQLite database, and how to retrieve data from SQLite database and process it using Python program.

## The `sqlite3` Module

SQLite is a server-less, file-based lightweight transactional relational database. It doesn't require any installation and no credentials such as username and password are needed to access the database.

Python's `sqlite3` module contains DB-API implementation for SQLite database. It is written by Gerhard Häring. Let us learn how to use `sqlite3` module for database access with Python.

Let us start by importing `sqlite3` and check its version.

```
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.39.4'
```

## The Connection Object

A connection object is set up by `connect()` function in `sqlite3` module. First positional argument to this function is a string representing path (relative or absolute) to a SQLite database file. The function returns a connection object referring to the database.

```
>>> conn=sqlite3.connect('testdb.sqlite3')
>>> type(conn)
<class 'sqlite3.Connection'>
```

Various methods are defined in connection class. One of them is `cursor()` method that returns a cursor object, about which we shall know in next section. Transaction control is

achieved by `commit()` and `rollback()` methods of connection object. Connection class has important methods to define custom functions and aggregates to be used in SQL queries.

## The Cursor Object

Next, we need to get the cursor object from the connection object. It is your handle to the database when performing any CRUD operation on the database. The `cursor()` method on connection object returns the cursor object.

```
>>> cur=conn.cursor()
>>> type(cur)
<class 'sqlite3.Cursor'>
```

We can now perform all SQL query operations, with the help of its `execute()` method available to cursor object. This method needs a string argument which must be a valid SQL statement.

## Creating a Database Table

We shall now add Employee table in our newly created 'testdb.sqlite3' database. In following script, we call `execute()` method of cursor object, giving it a string with CREATE TABLE statement inside.

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry=''''
CREATE TABLE Employee (
    EmpID INTEGER PRIMARY KEY AUTOINCREMENT,
    FIRST_NAME TEXT (20),
    LAST_NAME TEXT(20),
    AGE INTEGER,
    SEX TEXT(1),
    INCOME FLOAT
);
'''
try:
    cur.execute(qry)
    print ('Table created successfully')
except:
    print ('error in creating table')
conn.close()
```

When the above program is run, the database with Employee table is created in the current working directory.

We can verify by listing out tables in this database in SQLite console.

```
sqlite> .open mydb.sqlite
sqlite> .tables
Employee
```

## INSERT Operation

The INSERT Operation is required when you want to create your records into a database table.

### Example

The following example, executes SQL INSERT statement to create a record in the EMPLOYEE table –

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="""INSERT INTO EMPLOYEE(FIRST_NAME,
    LAST_NAME, AGE, SEX, INCOME)
VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
    cur.execute(qry)
    conn.commit()
    print ('Record inserted successfully')
except:
    conn.rollback()
print ('error in INSERT operation')
conn.close()
```

You can also use the parameter substitution technique to execute the INSERT query as follows –

```
import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="""INSERT INTO EMPLOYEE(FIRST_NAME,
    LAST_NAME, AGE, SEX, INCOME)
VALUES (?, ?, ?, ?, ?)"""
try:
    cur.execute(qry, ('Makrand', 'Mohan', 21, 'M', 5000))
```

```

    conn.commit()
    print ('Record inserted successfully')
except Exception as e:
    conn.rollback()
    print ('error in INSERT operation')
conn.close()

```

## READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once the database connection is established, you are ready to make a query into this database. You can use either `fetchone()` method to fetch a single record or `fetchall()` method to fetch multiple values from a database table.

- **`fetchone()`** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **`fetchall()`** – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- **`rowcount`** – This is a read-only attribute and returns the number of rows that were affected by an `execute()` method.

### Example

In the following code, the cursor object executes `SELECT * FROM EMPLOYEE` query. The resultset is obtained with `fetchall()` method. We print all the records in the resultset with a for loop.

```

import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="SELECT * FROM EMPLOYEE"

try:
    # Execute the SQL command
    cur.execute(qry)
    # Fetch all the rows in a list of lists.
    results = cur.fetchall()
    for row in results:
        fname = row[1]
        lname = row[2]
        age = row[3]
        sex = row[4]

```

```

income = row[5]

# Now print fetched result

print ("fname={},lname={},age={},sex={},income={}".format(fname, lname,
age, sex, income))

except Exception as e:

    print (e)

    print ("Error: unable to fetch data")

conn.close()

```

It will produce the following output –

```

fname=Mac,lname=Mohan,age=20,sex=M,income=2000.0
fname=Makrand,lname=Mohan,age=21,sex=M,income=5000.0

```

## UPDATE Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having income=2000. Here, we increase the income by 1000.

```

import sqlite3

conn=sqlite3.connect('testdb.sqlite3')

cur=conn.cursor()

qry="UPDATE EMPLOYEE SET INCOME = INCOME+1000 WHERE INCOME=?"

try:

    # Execute the SQL command
    cur.execute(qry, (1000,))

    # Fetch all the rows in a list of lists.
    conn.commit()

    print ("Records updated")

except Exception as e:

    print ("Error: unable to update data")

conn.close()

```

## DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where INCOME is less than 2000.

```

import sqlite3
conn=sqlite3.connect('testdb.sqlite3')
cur=conn.cursor()
qry="DELETE FROM EMPLOYEE WHERE INCOME<?"
try:
    # Execute the SQL command
    cur.execute(qry, (2000,))
    # Fetch all the rows in a list of lists.
    conn.commit()
    print ("Records deleted")
except Exception as e:
    print ("Error: unable to delete data")

conn.close()

```

## Performing Transactions

Transactions are a mechanism that ensure data consistency. Transactions have the following four properties –

- **Atomicity** – Either a transaction completes or nothing happens at all.
- **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.



The Python DB API 2.0 provides two methods to either commit or rollback a transaction.

### Example

You already know how to implement transactions. Here is a similar example –

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > ?"
try:
    # Execute the SQL command
    cursor.execute(sql, (20,))
    # Commit your changes in the database
    db.commit()
except:
    # Rollback in case there is any error
    db.rollback()
```

## COMMIT Operation

Commit is an operation, which gives a green signal to the database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call the commit method.

```
db.commit()
```

## ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use the rollback() method.

Here is a simple example to call the rollback() method.

```
db.rollback()
```

## The PyMySQL Module

PyMySQL is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb.

### Installing PyMySQL

Before proceeding further, you make sure you have PyMySQL installed on your machine. Just type the following in your Python script and execute it –

```
import PyMySQL
```

If it produces the following result, then it means MySQLdb module is not installed –

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    Import PyMySQL
ImportError: No module named PyMySQL
```

The last stable release is available on PyPI and can be installed with pip –

```
pip install PyMySQL
```

Note – Make sure you have root privilege to install the above module.

## MySQL Database Connection

Before connecting to a MySQL database, make sure of the following points –

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST\_NAME, LAST\_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Python module PyMySQL is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

### Example

To use MySQL database instead of SQLite database in earlier examples, we need to change the connect() function as follows –

```
import PyMySQL
# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )
```

Apart from this change, every database operation can be performed without difficulty.

## Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already cancelled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

| Sr.No. | Exception & Description                                                                                        |
|--------|----------------------------------------------------------------------------------------------------------------|
| 1      | <b>Warning</b><br>Used for non-fatal issues. Must subclass StandardError.                                      |
| 2      | <b>Error</b><br>Base class for errors. Must subclass StandardError.                                            |
| 3      | <b>InterfaceError</b><br>Used for errors in the database module, not the database itself. Must subclass Error. |
| 4      | <b>DatabaseError</b><br>Used for errors in the database. Must subclass Error.                                  |
| 5      | <b>DataError</b><br>Subclass of DatabaseError that refers to errors in the data.                               |

|    |                                                                                                                                                                             |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <b>OperationalError</b>                                                                                                                                                     |
| 6  | Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter. |
| 7  | <b>IntegrityError</b><br>Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.               |
| 8  | <b>InternalError</b><br>Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.                           |
| 9  | <b>ProgrammingError</b><br>Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.                      |
| 10 | <b>NotSupportedError</b><br>Subclass of DatabaseError that refers to trying to call unsupported functionality.                                                              |

# 191. Python - Weak References

Python uses reference counting mechanism while implementing garbage collection policy. Whenever an object in the memory is referred, the count is incremented by one. On the other hand, when the reference is removed, the count is decremented by 1. If the garbage collector running in the background finds any object with count as 0, it is removed and the memory occupied is reclaimed.

Weak reference is a reference that does not protect the object from getting garbage collected. It proves important when you need to implement caches for large objects, as well as in a situation where reduction of Pain from circular references is desired.

To create weak references, Python has provided us with a module named `weakref`.

The `ref` class in this module manages the weak reference to an object. When called, it retrieves the original object.

To create a weak reference –

```
weakref.ref(class())
```

## Example

```
import weakref

class Myclass:

    def __del__(self):
        print('Deleting {}'.format(self))

obj = Myclass()
r = weakref.ref(obj)

print('object:', obj)
print('reference:', r)
print('call r():', r())

print('deleting obj')
del obj
print('r():', r())
```

Calling the reference object after deleting the referent returns `None`.

It will produce the following output –

```
object: <__main__.Myclass object at 0x00000209D7173290>
reference: <weakref at 0x00000209D7175940; to 'Myclass' at
0x00000209D7173290>
call r(): <__main__.Myclass object at 0x00000209D7173290>
```

```
deleting obj
(Deleting <__main__.Myclass object at 0x00000209D7173290>
r(): None
```

## The callback Function

The constructor of ref class has an optional parameter called callback function, which gets called when the referred object is deleted.

```
import weakref

class Myclass:

    def __del__(self):
        print('Deleting {}'.format(self))

    def mycallback(rfr):
        """called when referenced object is deleted"""
        print('calling {}'.format(rfr))

    obj = Myclass()
    r = weakref.ref(obj, mycallback)

    print('object:', obj)
    print('reference:', r)
    print('call r():', r())

    print('deleting obj')
    del obj
    print('r():', r())
```

It will produce the following output –

```
object: <__main__.Myclass object at 0x000002A0499D3590>
reference: <weakref at 0x000002A0499D59E0; to 'Myclass' at
0x000002A0499D3590>
call r(): <__main__.Myclass object at 0x000002A0499D3590>
deleting obj
(Deleting <__main__.Myclass object at 0x000002A0499D3590>
calling (<weakref at 0x000002A0499D59E0; dead>)
r(): None
```

## Finalizing Objects

The weakref module provides finalize class. Its object is called when the garbage collector collects the object. The object survives until the reference object is called.

```
import weakref

class Myclass:
    def __del__(self):
        print('Deleting {}'.format(self))

    def finalizer(*args):
        print('Finalizer{}'.format(args))

    obj = Myclass()
    r = weakref.finalize(obj, finalizer, "Call to finalizer")

    print('object:', obj)
    print('reference:', r)
    print('call r():', r())

    print('deleting obj')
    del obj
    print('r():', r())
```

It will produce the following output –

```
object: <__main__.Myclass object at 0x0000021015103590>
reference: <finalize object at 0x21014eabe80; for 'Myclass' at
0x21015103590>
Finalizer('Call to finalizer',())
call r(): None
deleting obj
(Deleting <__main__.Myclass object at 0x0000021015103590>)
r(): None
```

The weakref module provides WeakKeyDictionary and WeakValueDictionary classes. They don't keep the objects alive as they appear in the mapping objects. They are more appropriate for creating a cache of several objects.

## WeakKeyDictionary

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key.

An instance of WeakKeyDictionary class is created with an existing dictionary or without any argument. The functionality is the same as a normal dictionary to add and remove mapping entries to it.

In the code given below three Person instances are created. It then creates an instance of WeakKeyDictionary with a dictionary where the key is the Person instance and the value is the Person's name.

We call the keyrefs() method to retrieve weak references. When the reference to Person1 is deleted, dictionary keys are printed again. A new Person instance is added to a dictionary with weakly referenced keys. At last, we are printing keys of dictionary again.

### Example

```
import weakref

class Person:
    def __init__(self, person_id, name, age):
        self.emp_id = person_id
        self.name = name
        self.age = age

    def __repr__(self):
        return "{} : {} : {}".format(self.person_id, self.name, self.age)

Person1 = Person(101, "Jeevan", 30)
Person2 = Person(102, "Ramanna", 35)
Person3 = Person(103, "Simran", 28)

weak_dict = weakref.WeakKeyDictionary({Person1: Person1.name, Person2: Person2.name, Person3: Person3.name})

print("Weak Key Dictionary : {}\n".format(weak_dict.data))
print("Dictionary Keys : {}\n".format([key().name for key in weak_dict.keyrefs()]))

del Person1

print("Dictionary Keys : {}\n".format([key().name for key in weak_dict.keyrefs()]))

Person4 = Person(104, "Partho", 32)
weak_dict.update({Person4: Person4.name})

print("Dictionary Keys : {}\n".format([key().name for key in weak_dict.keyrefs()]))
```

It will produce the following output –

```
Weak Key Dictionary : {<weakref at 0x7f542b6d4180; to 'Person' at 0x7f542b8bbfd0>: 'Jeevan', <weakref at 0x7f542b6d5530; to 'Person' at 0x7f542b8bbeb0>: 'Ramanna', <weakref at 0x7f542b6d55d0; to 'Person' at 0x7f542b8bb7c0>: 'Simran'}
```

```
Dictionary Keys : ['Jeevan', 'Ramanna', 'Simran']
```

```
Dictionary Keys : ['Ramanna', 'Simran']
```

```
Dictionary Keys : ['Ramanna', 'Simran', 'Partho']
```

## WeakValueDictionary

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

We shall demonstrate how to create a dictionary with weakly referenced values using WeakValueDictionary.

The code is similar to previous example but this time we are using Person name as key and Person instance as values. We are using valuerefs() method to retrieve weakly referenced values of the dictionary.

### Example

```
import weakref

class Person:
    def __init__(self, person_id, name, age):
        self.emp_id = person_id
        self.name = name
        self.age = age

    def __repr__(self):
        return "{} : {} : {}".format(self.person_id, self.name, self.age)

Person1 = Person(101, "Jeevan", 30)
Person2 = Person(102, "Ramanna", 35)
Person3 = Person(103, "Simran", 28)

weak_dict = weakref.WeakValueDictionary({Person1.name:Person1,
Person2.name:Person2, Person3.name:Person3})

print("Weak Value Dictionary : {}\n".format(weak_dict.data))
print("Dictionary Values : {}\n".format([value().name for value in weak_dict.valuerefs()]))
```

```
del Person1

print("Dictionary Values : {}\n".format([value().name for value in
weak_dict.valuerefs()]))

Person4 = Person(104, "Partho", 32)

weak_dict.update({Person4.name: Person4})

print("Dictionary Values : {}\n".format([value().name for value in
weak_dict.valuerefs()]))
```

It will produce the following output –

```
Weak Value Dictionary : {'Jeevan': <weakref at 0x7f3af9fe4180; to 'Person' at
0x7f3afa1c7fd0>, 'Ramanna': <weakref at 0x7f3af9fe5530; to 'Person' at
0x7f3afa1c7eb0>, 'Simran': <weakref at 0x7f3af9fe55d0; to 'Person' at
0x7f3afa1c77c0>}

Dictionary Values : ['Jeevan', 'Ramanna', 'Simran']

Dictionary Values : ['Ramanna', 'Simran']

Dictionary Values : ['Ramanna', 'Simran', 'Partho']
```

# 192. Python - Serialization

## Serialization in Python

Serialization refers to the process of converting an object into a format that can be easily stored, transmitted, or reconstructed later. In Python, this involves converting complex data structures, such as objects or dictionaries, into a byte stream.

### Why do we use Serialization?

Serialization allows data to be easily saved to disk or transmitted over a network, and later reconstructed back into its original form. It is important for tasks like saving game states, storing user preferences, or exchanging data between different systems.

## Serialization Libraries in Python

Python offers several libraries for serialization, each with its own advantages. Here is a detailed overview of some commonly used serialization libraries in Python –

- **Pickle** – This is Python's built-in module for serializing and deserializing Python objects. It is simple to use but specific to Python and may have security implications if used with untrusted data.
- **JSON** – JSON (JavaScript Object Notation) is a lightweight data interchange format that is human-readable and easy to parse. It is ideal for web APIs and cross-platform communication.
- **YAML** – YAML (YAML Ain't Markup Language) is a human-readable data serialization standard that is also easy for both humans and machines to read and write. It supports complex data structures and is often used in configuration files.

### Serialization Using Pickle Module

The pickle module in Python is used for serializing and deserializing objects. Serialization, also known as pickling, involves converting a Python object into a byte stream, which can then be stored in a file or transmitted over a network.

Deserialization, or unpickling, is the reverse process, converting the byte stream back into a Python object.

### Serializing an Object

We can serialize an object using the `dump()` function and write it to a file. The file must be opened in binary write mode ('wb').

### Example

In the following example, a dictionary is serialized and written to a file named "data.pkl"

```
import pickle

data = {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

```
# Open a file in binary write mode
with open('data.pkl', 'wb') as file:
    # Serialize the data and write it to the file
    pickle.dump(data, file)
    print ("File created!!")
```

When above code is executed, the dictionary object's byte representation will be stored in data.pkl file.

### Deserializing an Object

To deserialize or unpickle the object, you can use the load() function. The file must be opened in binary read mode ('rb') as shown below –

```
import pickle

# Open the file in binary read mode
with open('data.pkl', 'rb') as file:
    # Deserialize the data
    data = pickle.load(file)
print(data)
```

This will read the byte stream from "data.pkl" and convert it back into the original dictionary as shown below –

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

### Pickle Protocols

Protocols are the conventions used in constructing and deconstructing Python objects to/from binary data.

The pickle module supports different serialization protocols, with higher protocols generally offering more features and better performance. Currently pickle module defines 6 different protocols as listed below –

| Sr.No. | Protocol & Description                                                                                         |
|--------|----------------------------------------------------------------------------------------------------------------|
|        |                                                                                                                |
| 1      | <b>Protocol version 0</b><br>Original "human-readable" protocol<br>backwards compatible with earlier versions. |
| 2      | <b>Protocol version 1</b><br>Old binary format also compatible with<br>earlier versions of Python.             |
| 3      | <b>Protocol version 2</b><br>Introduced in Python 2.3 provides efficient<br>pickling of new-style classes.     |
| 4      | <b>Protocol version 3</b>                                                                                      |

|   |                                                                                                |
|---|------------------------------------------------------------------------------------------------|
|   | Added in Python 3.0. recommended when compatibility with other Python 3 versions is required.  |
| 5 | <b>Protocol version 4</b><br>Introduced in Python 3.4. It adds support for very large objects. |
| 6 | <b>Protocol version 5</b><br>Introduced in Python 3.8. It adds support for out-of-band data.   |

You can specify the protocol by passing it as an argument to `pickle.dump()` function.

To know the highest and default protocol version of your Python installation, use the following constants defined in the `pickle` module –

```
>>> import pickle
>>> pickle.HIGHEST_PROTOCOL
5
>>> pickle.DEFAULT_PROTOCOL
4
```

## Pickler and Unpickler Classes

The `pickle` module in Python also defines Pickler and Unpickler classes for more detailed control over the serialization and deserialization processes. The "Pickler" class writes pickle data to a file, while the "Unpickler" class reads binary data from a file and reconstructs the original Python object.

### Using the Pickler Class

To serialize a Python object using the Pickler class, you can follow these steps –

```
from pickle import Pickler

# Open a file in binary write mode
with open("data.txt", "wb") as f:
    # Create a dictionary
    dct = {'name': 'Ravi', 'age': 23, 'Gender': 'M', 'marks': 75}
    # Create a Pickler object and write the dictionary to the file
    Pickler(f).dump(dct)
    print ("Success!!")
```

After executing the above code, the dictionary object's byte representation will be stored in "data.txt" file.

### Using the Unpickler Class

To deserialize the data from a binary file using the Unpickler class, you can do the following –

```
from pickle import Unpickler

# Open the file in binary read mode
with open("data.txt", "rb") as f:

    # Create an Unpickler object and load the dictionary from the file
    dct = Unpickler(f).load()

    # Print the dictionary
    print(dct)
```

We get the output as follows –

```
{'name': 'Ravi', 'age': 23, 'Gender': 'M', 'marks': 75}
```

## Pickling Custom Class Objects

The pickle module can also serialize and deserialize custom classes. The class definition must be available at both the time of pickling and unpickling.

### Example

In this example, an instance of the "Person" class is serialized and then deserialized, maintaining the state of the object –

```
import pickle

class Person:

    def __init__(self, name, age, city):
        self.name = name
        self.age = age
        self.city = city

    # Create an instance of the Person class
    person = Person('Alice', 30, 'New York')

    # Serialize the person object
    with open('person.pkl', 'wb') as file:
        pickle.dump(person, file)

    # Deserialize the person object
    with open('person.pkl', 'rb') as file:
        person = pickle.load(file)
```

```
print(person.name, person.age, person.city)
```

After executing the above code, we get the following output –

```
Alice 30 New York
```

*The Python standard library also includes the `marshal` module, which is used for internal serialization of Python objects. Unlike `pickle`, which is designed for general-purpose use, `marshal` is primarily intended for use by Python itself (e.g., for writing `.pyc` files).*

*It is generally not recommended for general-purpose serialization due to potential compatibility issues between Python versions.*

## Using JSON for Serialization

JSON (JavaScript Object Notation) is a popular format for data interchange. It is human-readable, easy to write, and language-independent, making it ideal for serialization.

Python provides built-in support for JSON through the `json` module, which allows you to serialize and deserialize data to and from JSON format.

### Serialization

Serialization is the process of converting a Python object into a JSON string or writing it to a file.

#### Example: Serialize Data to a JSON String

In the example below, we use the `json.dumps()` function to convert a Python dictionary to a JSON string –

```
import json

# Create a dictionary
data = {"name": "Alice", "age": 25, "city": "San Francisco"}

# Serialize the dictionary to a JSON string
json_string = json.dumps(data)
print(json_string)
```

Following is the output of the above code –

```
{"name": "Alice", "age": 25, "city": "San Francisco"}
```

#### Example: Serialize Data and Write to a File

In here, we use the `json.dump()` function to write the serialized JSON data directly to a file –

```
import json
```

```
# Create a dictionary
data = {"name": "Alice", "age": 25, "city": "San Francisco"}

# Serialize the dictionary and write it to a file
with open("data.json", "w") as f:
    json.dump(data, f)
    print ("Success!!")
```

## Deserialization

Deserialization is the process of converting a JSON string back into a Python object or reading it from a file.

### Example: Deserialize a JSON String

In the following example, we use the `json.loads()` function to convert a JSON string back into a Python dictionary –

```
import json

# JSON string
json_string = '{"name": "Alice", "age": 25, "city": "San Francisco"}'

# Deserialize the JSON string into a Python dictionary
loaded_data = json.loads(json_string)
print(loaded_data)
```

It will produce the following output –

```
{"name": 'Alice', 'age': 25, 'city': 'San Francisco'}
```

### Example: Deserialize Data from a File

Here, we use the `json.load()` function to read JSON data from a file and convert it to a Python dictionary –

```
import json

# Open the file and load the JSON data into a Python dictionary
with open("data.json", "r") as f:
    loaded_data = json.load(f)
    print(loaded_data)
```

The output obtained is as follows –

```
{"name": 'Alice', 'age': 25, 'city': 'San Francisco'}
```

## Using YAML for Serialization

YAML (YAML Ain't Markup Language) is a human-readable data serialization standard that is commonly used for configuration files and data interchange.

Python supports YAML serialization and deserialization through the `pyyaml` package, which needs to be installed first as shown below –

```
pip install pyyaml
```

### Example: Serialize Data and Write to a YAML File

In the below example, `yaml.dump()` function converts the Python dictionary data into a YAML string and writes it to the file "data.yaml".

The "default\_flow\_style" parameter ensures that the YAML output is more human-readable with expanded formatting –

```
import yaml

# Create a Python dictionary
data = {"name": "Emily", "age": 35, "city": "Seattle"}

# Serialize the dictionary and write it to a YAML file
with open("data.yaml", "w") as f:
    yaml.dump(data, f, default_flow_style=False)
    print("Success!!")
```

### Example: Deserialize Data from a YAML File

Here, `yaml.safe_load()` function is used to safely load the YAML data from "data.yaml" and convert it into a Python dictionary (`loaded_data`) –

*Using `safe_load()` is preferred for security reasons as it only allows basic Python data types and avoids executing arbitrary code from YAML files.*

```
import yaml

# Deserialize data from a YAML file
with open("data.yaml", "r") as f:
    loaded_data = yaml.safe_load(f)
    print(loaded_data)
```

The output produced is as shown below –

```
{'age': 35, 'city': 'Seattle', 'name': 'Emily'}
```

# 193. Python - Templating

## Templating in Python

Templating in Python is a technique used in web development to dynamically generate static HTML pages using templates and data.

In this tutorial, we will explore the basics of templating in Python, including installation, creating templates, and rendering templates with data, with a focus on the Jinja2 templating engine.

## String Templates in Python

String templates in Python is a simple way to perform string substitutions. Python's string module includes the `Template` class, which provides an easy way to replace placeholders in a string with actual values.

The `Template` class in the `string` module is useful for dynamically forming a string object through a substitution technique described in PEP 292. Its simpler syntax and functionality make it easier to translate for internationalization purposes compared to other built-in string formatting facilities in Python.

Template strings use the `$` symbol for substitution, immediately followed by an identifier that follows the rules of forming a valid Python identifier.

## Creating a Template

To create a template, you instantiate the `Template` class with a string that contains placeholders prefixed with `$` as shown below –

```
from string import Template

template = Template("Hello, $name!")
```

## Substituting Values

You can substitute values into the template using the `substitute()` method, which takes a dictionary of key-value pairs.

The `substitute()` method replaces the placeholders (identifiers) in the template with actual values. You can provide these values using keyword arguments or a dictionary. The method then returns a new string with the placeholders filled in.

### Example: Using Keyword Arguments

Following code substitute identifiers in a template string using keyword arguments –

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $age')
```

```
newStr = tempStr.substitute(name = 'Pushpa', age = 26)
print (newStr)
```

It will produce the following output –

```
Hello. My name is Pushpa and my age is 26
```

### **Example: Using a Dictionary**

In the following example, we use a dictionary object to map the substitution identifiers in the template string –

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $age')
dct = {'name' : 'Pushpalata', 'age' : 25}
newStr = tempStr.substitute(dct)
print (newStr)
```

Following is the output of the above code –

```
Hello. My name is Pushpalata and my age is 25
```

### **Example: Missing Parameters Raises KeyError**

If the substitute() method is not provided with sufficient parameters to be matched against the identifiers in the template string, Python raises `KeyError` –

```
from string import Template

tempStr = Template('Hello. My name is $name and my age is $age')
dct = {'name' : 'Pushpalata'}
newStr = tempStr.substitute(dct)
print (newStr)
```

Following is the error produced –

```
Traceback (most recent call last):
  File "/home/cg/root/667e441d9ebd5/main.py", line 5, in <module>
    newStr = tempStr.substitute(dct)
  File "/usr/lib/python3.10/string.py", line 121, in substitute
    return self.pattern.sub(convert, self.template)
  File "/usr/lib/python3.10/string.py", line 114, in convert
    return str(mapping[named])
KeyError: 'age'
```

### **Substituting Values Using `safe_substitute()` Method**

The `safe_substitute()` method behaves similarly to `substitute()` method, except for the fact that it doesn't throw error if the keys are not sufficient or are not matching. Instead, the original placeholder will appear in the resulting string intact.

### Example

In the following example, we are using the `safe_substitute()` method for substituting values –

```
from string import Template
tempStr = Template('Hello. My name is $name and my age is $age')
dct = {'name' : 'Pushpalata'}
newStr = tempStr.safe_substitute(dct)
print (newStr)
```

It will produce the following output –

```
Hello. My name is Pushpalata and my age is $age
```

## Installing Ninja2

To use Ninja2 for templating in Python, you first need to install the library. Ninja2 is a powerful templating engine that is widely used in web development for rendering HTML. It can be installed easily using pip, Python's package installer –

```
pip install jinja2
```

## Creating and Rendering Ninja2 Templates

Jinja2 is a powerful templating engine for Python that allows you to create dynamic content by blending static template files with data. This section explores how to create Ninja2 templates and render them with data.

### Creating a Ninja2 Template

To create a Ninja2 template, you define a template string or load it from a file. Templates use double curly braces `{{ ... }}` for placeholders and support control structures like "loops" and "conditionals" with `{% ... %}`.

### Example

Following is an example of a simple Ninja2 template stored in a file "template.html" –

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello, {{ name }}!</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
    <p>Welcome to our site.</p>
```

```
</body>
</html>
```

## Rendering a Jinja2 Template

To render a Jinja2 template, follow these steps –

- **Loading the Template** – Load the template from a file or create it from a string.
- **Creating a Template Object** – Use "jinja2.Template" to create a template object.
- **Rendering** – Use the render() method on the template object, passing data as arguments or a dictionary.

### Example

In here, we are rendering Jinja2 template –

```
from jinja2 import Template, FileSystemLoader, Environment

# Loading a template from a file (template.html)
file_loader = FileSystemLoader('.')
env = Environment(loader=file_loader)
template = env.get_template('template.html')

# Rendering the template with data
output = template.render(name='Alice')

# Output the rendered template
print(output)
```

The output of the rendered Jinja2 template would be an HTML document with the placeholders replaced by the actual data passed during rendering –

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello, Alice!</title>
</head>
<body>
    <h1>Hello, Alice!</h1>
    <p>Welcome to our site.</p>
</body>
</html>
```

## Advanced Jinja2 Features

Jinja2 supports various advanced features such as loops, conditionals, and custom filters, making it a powerful tool for creating complex templates.

### Template Inheritance

Jinja2 supports template inheritance, allowing you to create a base template with common elements (like headers, footers, navigation bars) and extend or override specific blocks in child templates. This promotes code reuse and maintainability in large projects.

#### Example

This HTML template file named "base.html" defines a basic structure for a web page using Jinja2 templating syntax.

It includes blocks "{% block title %}" and "{% block content %}" that can be overridden in derived templates to customize the title and main content of the page, respectively –

```
<!-- base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Default Title{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

The following Jinja2 template file "child.html" extends the "base.html" template, overriding the title block to set it to "Child Page" and the content block to include an `<h1>` header with the text "Child Page Content".

```
<!-- child.html -->
{% extends "base.html" %}

{% block title %}Child Page{% endblock %}

{% block content %}
<h1>Child Page Content</h1>
{% endblock %}
```

### Loops

Jinja2 allows you to iterate over lists or other iterable objects using `{% for %}` loops. Following is an example of how you can use a loop to generate an unordered list (`<ul>`) in HTML –

```
<ul>
{% for item in items %}
<li>{{ item }}</li>
{% endfor %}
</ul>
```

## Conditionals

Conditional statements in Jinja2 (`{% if %}` and `{% else %}`) is used to control the flow of your templates based on conditions. Here is an example where "Jinja2" checks if user exists and displays a personalized greeting if true; otherwise, it prompts to log in –

```
{% if user %}
    <p>Welcome, {{ user }}!</p>
{% else %}
    <p>Please log in.</p>
{% endif %}
```

## Custom Filters

Custom filters in Jinja2 is used to define your own filters to manipulate data before displaying it in the template.

In the following example, a custom filter reverse is defined in Jinja2 to reverse the string "hello", resulting in "olleh" when applied in the template –

```
# Define a custom filter function
def reverse_string(s):
    return s[::-1]

# Register the filter with the Jinja2 environment
env.filters['reverse'] = reverse_string
```

In your template, you can then apply the "reverse" filter to any string –

```
{{ "hello" | reverse }}
```

Following is the output obtained –

```
olleh
```

# 194. Python - Output Formatting

## Output Formatting in Python

Output formatting in Python is used to make your code more readable and your output more user-friendly. Whether you are displaying simple text strings, complex data structures, or creating reports, Python offers several ways for formatting output.

These ways include using –

- The string modulo operator (%)
- The format() method
- The f-strings (formatted string literals)
- The template strings

Additionally, Python's "textwrap" and "pprint" modules offer advanced functionalities for wrapping text and pretty-printing data structures.

### Using String Modulo Operator (%)

We can format output using the string modulo operator %. This operator is unique to strings and makes up for the lack of having functions from C's printf() family. Format specification symbols like %d, %c, %f, and %s are used as placeholders in a string, similar to those in C.

Following is a simple example –

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

It will produce the following output –

```
My name is Zara and weight is 21 kg!
```

### Using the format() Method

We can format output using the format() method, which was introduced in Python 3.0 and has been backported to Python 2.6 and 2.7.

The format() method is part of the built-in string class and allows for complex variable substitutions and value formatting. It is considered a more elegant and flexible way to format strings compared to the string modulo operator.

#### Syntax

The general syntax of format() method is as follows –

```
str.format(var1, var2,...)
```

#### Return Value

The method returns a formatted string.

The string itself contains placeholders {} in which values of variables are successively inserted.

**Example**

```
name="Rajesh"
age=23
print ("my name is {} and my age is {} years".format(name, age))
```

It will produce the following output –

```
my name is Rajesh and my age is 23 years
```

You can use variables as keyword arguments to format() method and use the variable name as the placeholder in the string.

```
print ("my name is {name} and my age is {age}
years".format(name="Rajesh", age=23))
```

**Using F-Strings**

F-strings, or formatted string literals, are a way to format strings in Python that is simple, fast, and easy to read. You create an f-string by adding an f before the opening quotation mark of a string.

Inside the string, you can include placeholders for variables, which are enclosed in curly braces {}. The values of these variables will be inserted into the string at those places.

**Example**

In this example, the variables "name" and "age" are inserted into the string where their placeholders "{name}" and "{age}" are located. F-strings make it easy to include variable values in strings without having to use the format() method or string concatenation –

```
name = 'Rajesh'
age = 23

fstring = f'My name is {name} and I am {age} years old'
print (fstring)
```

It will produce the following output –

```
My name is Rajesh and I am 23 years old
```

**Format Conversion Rule in Python**

You can also specify C-style formatting symbols. The only change is using : instead of %. For example, instead of %s use {:s} and instead of %d use {:d} as shown below –

```
name = "Rajesh"
age = 23
print("my name is {:s} and my age is {:d} years".format(name, age))
```

You will get the output as shown below –

```
my name is Rajesh and my age is 23 years
```

## Template Strings

The Template class in string module provides an alternative method to format the strings dynamically. One of the benefits of Template class is to be able to customize the formatting rules.

A valid template string, or placeholder, consists of two parts: The \$ symbol followed by a valid Python identifier.

You need to create an object of Template class and use the template string as an argument to the constructor. Next, call the substitute() method of Template class. It puts the values provided as the parameters in place of template strings.

### Example

```
from string import Template

temp_str = "My name is $name and I am $age years old"
tempobj = Template(temp_str)
ret = tempobj.substitute(name='Rajesh', age=23)
print (ret)
```

It will produce the following output –

```
My name is Rajesh and I am 23 years old
```

## The textwrap Module

The wrap class in Python's textwrap module contains functionality to format and wrap plain texts by adjusting the line breaks in the input paragraph. It helps in making the text well-formatted and beautiful.

The textwrap module has the following convenience functions –

### Python textwrap.wrap(text, width=70)

The textwrap.wrap() function wraps the single paragraph in text (a string) so every line is at most width characters long. Returns a list of output lines, without final newlines. Optional keyword arguments correspond to the instance attributes of TextWrapper. width defaults to 70.

### Python textwrap.fill(text, width=70)

The textwrap.fill() function wraps the single paragraph in text, and returns a single string containing the wrapped paragraph.

Both methods internally create an object of TextWrapper class and calling a single method on it. Since the instance is not reused, it will be more efficient for you to create your own TextWrapper object.

### Example

```

import textwrap

text = '''

Python is a high-level, general-purpose programming language. Its design
philosophy emphasizes code readability with the use of significant indentation
via the off-side rule.

Python is dynamically typed and garbage-collected. It supports multiple
programming paradigms, including structured (particularly procedural), object-
oriented and functional programming. It is often described as a "batteries
included" language due to its comprehensive standard library.

'''

wrapper = textwrap.TextWrapper(width=40)
wrapped = wrapper.wrap(text = text)

# Print output
for element in wrapped:
    print(element)

```

It will produce the following output –

```

Python is a high-level, general-purpose
programming language. Its design
philosophy emphasizes code readability
with the use of significant indentation
via the off-side rule. Python is
dynamically typed and garbage-collected.

It supports multiple programming
paradigms, including structured
(particularly procedural), object-oriented and functional programming. It
is often described as a "batteries
included" language due to its
comprehensive standard library.

```

Following attributes are defined for a TextWrapper object –

- **width** – (default: 70) The maximum length of wrapped lines.
- **expand\_tabs** – (default: True) If true, then all tab characters in text will be expanded to spaces using the expandtabs() method of text.

- **tabsize** – (default: 8) If expand\_tabs is true, then all tab characters in text will be expanded to zero or more spaces, depending on the current column and the given tab size.
- **replace\_whitespace** – (default: True) If true, after tab expansion but before wrapping, the wrap() method will replace each whitespace character with a single space.
- **drop\_whitespace** – (default: True) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.
- **initial\_indent** – (default: "") String that will be prepended to the first line of wrapped output.
- **subsequent\_indent** – (default: "") String that will be prepended to all lines of wrapped output except the first.
- **fix\_sentence\_endings** – (default: False) If true, TextWrapper attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font.
- **break\_long\_words** – (default: True) If true, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width.
- **break\_on\_hyphens** – (default: True) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks.

## The shorten() Function

The shorten() function collapse and truncate the given text to fit in the given width. The text first has its whitespace collapsed. If it then fits in the \*width\*, it is returned as is. Otherwise, as many words as possible are joined and then the placeholder is appended –

### Example

```
import textwrap

python_desc = """Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language."""

my_wrap = textwrap.TextWrapper(width = 40)

short_text = textwrap.shorten(text = python_desc, width=150)
print('\n\n' + my_wrap.fill(text = short_text))
```

It will produce the following output –

```
Python is a general-purpose interpreted,
```

```
interactive, object-oriented, and high
level programming language. It was
created by Guido van Rossum [...]
```

## The pprint Module

The pprint module in Python's standard library enables aesthetically good looking appearance of Python data structures. The name pprint stands for pretty printer. Any data structure that can be correctly parsed by Python interpreter is elegantly formatted.

The formatted expression is kept in one line as far as possible, but breaks into multiple lines if the length exceeds the width parameter of formatting. One unique feature of pprint output is that the dictionaries are automatically sorted before the display representation is formatted.

### PrettyPrinter Class

The pprint module contains definition of PrettyPrinter class. Its constructor takes following format –

#### Syntax

```
pprint.PrettyPrinter(indent, width, depth, stream, compact)
```

#### Parameters

- **indent** – defines indentation added on each recursive level. Default is 1.
- **width** – by default is 80. Desired output is restricted by this value. If the length is greater than width, it is broken in multiple lines.
- **depth** – controls number of levels to be printed.
- **stream** – is by default std.out – the default output device. It can take any stream object such as file.
- **compact** – is set to False by default. If true, only the data adjustable within width will be displayed.

The PrettyPrinter class defines following methods –

#### Python pprint() method

The pprint() method prints the formatted representation of PrettyPrinter object.

#### Python pformat() method

The pformat() method returns the formatted representation of object, based on parameters to the constructor.

#### Example

The following example demonstrates a simple use of PrettyPrinter class –

```
import pprint
students={"Dilip":["English", "Maths",
"Science"],"Raju":{"English":50,"Maths":60, "Science":70}, "Kalpana":(50,60,70)}
pp=pprint.PrettyPrinter()
print ("normal print output")
```

```
print (students)
print ("----")
print ("pprint output")
pp.pprint(students)
```

The output shows normal as well as pretty print display –

```
normal print output
{'Dilip': ['English', 'Maths', 'Science'], 'Raju': {'English': 50, 'Maths': 60,
'Science': 70}, 'Kalpana': (50, 60, 70)}
-----
pprint output
{'Dilip': ['English', 'Maths', 'Science'],
'Kalpana': (50, 60, 70),
'Raju': {'English': 50, 'Maths': 60, 'Science': 70}}
```

The pprint module also defines convenience functions pprint() and pformat() corresponding to PrettyPrinter methods. The example below uses pprint() function.

```
from pprint import pprint
students={"Dilip":["English", "Maths", "Science"],
          "Raju":{"English":50,"Maths":60, "Science":70},
          "Kalpana":(50,60,70)}
print ("normal print output")
print (students)
print ("----")
print ("pprint output")
pprint (students)
```

### **Example: Using pformat() Method**

The next example uses pformat() method as well as pprint() function. To use pformat() method, the PrettyPrinter object is first set up. In both cases, the formatted representation is displayed using normal print() function.

```
import pprint
students={"Dilip":["English", "Maths", "Science"],
          "Raju":{"English":50,"Maths":60, "Science":70},
          "Kalpana":(50,60,70)}
print ("using pformat method")
pp=pprint.PrettyPrinter()
string=pp.pformat(students)
print (string)
```

```

print ('-----')
print ("using pformat function")
string=pprint.pformat(students)
print (string)

```

Here is the output of the above code –

```

using pformat method
{'Dilip': ['English', 'Maths', 'Science'],
 'Kalpana': (50, 60, 70),
 'Raju': {'English': 50, 'Maths': 60, 'Science': 70}}
-----
using pformat function
{'Dilip': ['English', 'Maths', 'Science'],
 'Kalpana': (50, 60, 70),
 'Raju': {'English': 50, 'Maths': 60, 'Science': 70}}

```

Pretty printer can also be used with custom classes. Inside the class `__repr__()` method is overridden. The `__repr__()` method is called when `repr()` function is used. It is the official string representation of Python object. When we use object as parameter to `print()` function it prints return value of `repr()` function.

### Example

In this example, the `__repr__()` method returns the string representation of `player` object –

```

import pprint

class player:
    def __init__(self, name, formats=[], runs=[]):
        self.name=name
        self.formats=formats
        self.runs=runs
    def __repr__(self):
        dct={}
        dct[self.name]=dict(zip(self.formats,self.runs))
        return (repr(dct))

l1=['Tests','ODI','T20']
l2=[[140, 45, 39],[15,122,36,67, 100, 49],[78,44, 12, 0, 23, 75]]
p1=player("virat",l1,l2)
pp=pprint.PrettyPrinter()

```

```
pp.pprint(p1)
```

The output of above code is –

```
{'virat': {'Tests': [140, 45, 39], 'ODI': [15, 122, 36, 67, 100, 49],  
'T20': [78, 44, 12, 0, 23, 75]}}
```

# 195. Python - Performance Measurement

A given problem may be solved by more than one alternative algorithms. Hence, we need to optimize the performance of the solution. Python's `timeit` module is a useful tool to measure the performance of a Python application.

The `timeit()` function in this module measures execution time of your Python code.

## Syntax

```
timeit.timeit(stmt, setup, timer, number)
```

## Parameters

- **stmt** – code snippet for measurement of performance.
- **setup** – setup details arguments to be passed or variables.
- **timer** – uses default timer, so, it may be skipped.
- **number** – the code will be executed this number of times. The default is 1000000.

## Example

The following statement uses list comprehension to return a list of multiple of 2 for each number in the range up to 100.

```
>>> [n*2 for n in range(100)]  
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,  
36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,  
70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100,  
102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126,  
128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152,  
154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178,  
180, 182, 184, 186, 188, 190, 192, 194, 196, 198]
```

To measure the execution time of the above statement, we use the `timeit()` function as follows –

```
>>> from timeit import timeit  
>>> timeit('[n*2 for n in range(100)]', number=10000)  
0.0862189000035869
```

Compare the execution time with the process of appending the numbers using a for loop.

```
>>> string = ''  
... numbers=[]  
... for n in range(100):  
...     numbers.append(n*2)  
...     ...
```

```
>>> timeit(string, number=10000)
0.1010853999905521
```

The result shows that list comprehension is more efficient.

The statement string can contain a Python function to which one or more arguments may be passed as setup code.

We shall find and compare the execution time of a factorial function using a loop with that of its recursive version.

The normal function using for loop is –

```
def fact(x):
    fact = 1
    for i in range(1, x+1):
        fact*=i
    return fact
```

Definition of recursive factorial.

```
def rfact(x):
    if x==1:
        return 1
    else:
        return x*fact(x-1)
```

Test these functions to calculate factorial of 10.

```
print ("Using loop:",fact(10))
print ("Using Recursion",rfact(10))

Result
Using loop: 3628800
Using Recursion 3628800
```

Now we shall find their respective execution time with timeit() function.

```
import timeit

setup1"""
from __main__ import fact
x = 10
"""

setup2"""
from __main__ import rfact
```

```
x = 10
"""

print ("Performance of factorial function with loop")
print(timeit.timeit(stmt = "fact(x)", setup=setup1, number=10000))

print ("Performance of factorial function with Recursion")
print(timeit.timeit(stmt = "rfact(x)", setup=setup2, number=10000))
```

### Output

```
Performance of factorial function with loop
0.00330029999895487
Performance of factorial function with Recursion
0.006506800003990065
```

The recursive function is slower than the function with loop.

In this way, we can perform performance measurement of Python code.

# 196. Python - Data Compression

Python's standard library has a rich collection of modules for data compression and archiving. One can select whichever is suitable for his job.

There are following modules related to data compression –

| Sr.No. | Module & Description                                          |
|--------|---------------------------------------------------------------|
| 1      | <a href="#">zlib</a><br>Compression compatible with gzip.     |
| 2      | <a href="#">gzip</a><br>Support for gzip files.               |
| 3      | <a href="#">bz2</a><br>Support for bz2 compression.           |
| 4      | <a href="#">lzma</a><br>Compression using the LZMA algorithm. |
| 5      | <a href="#">zipfile</a><br>Work with ZIP archives.            |
| 6      | <a href="#">tarfile</a><br>Read and write tar archive files.  |

# 197. Python - CGI Programming

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA.

## What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

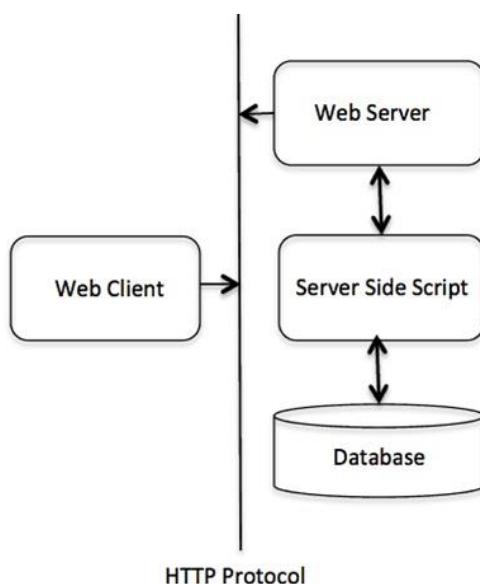
## Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
- Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface (CGI) and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

## CGI Architecture Diagram



## Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /var/www/cgi-bin. By convention, CGI files have extension as .cgi, but you can keep your files with python extension .py as well.

By default, the Linux server is configured to run only the scripts in the cgi-bin directory in /var/www. If you want to specify any other directory to run your CGI scripts, comment the following lines in the httpd.conf file –

```
<Directory "/var/www/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
    Options All
</Directory>
```

The following line should also be added for apache server to treat .py file as cgi script.

```
AddHandler cgi-script .py
```

Here, we assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell, etc.

## First CGI Program

Here is a simple link, which is linked to a CGI script called hello.py. This file is kept in /var/www/cgi-bin directory and it has following content. Before running your CGI program, make sure you have change mode of file using chmod 755 hello.py UNIX command to make file executable.

```
print ("Content-type:text/html\r\n\r\n")
print ('<html>')
print ('<head>')
print ('<title>Hello Word - First CGI Program</title>')
print ('</head>')
print ('<body>')
print ('<h2>Hello Word! This is my first CGI program</h2>')
print ('</body>')
print ('</html>')
```

**Note –** First line in the script must be the path to Python executable. It appears as a comment in Python program, but it is called shebang line.

In Linux, it should be `#!/usr/bin/python3`.

In Windows, it should be `#!c:/python311/python.exe`.

Enter the following URL in your browser –

```
http://localhost/cgi-bin/hello.py
Hello Word! This is my first CGI program
```

This `hello.py` script is a simple Python script, which writes its output on STDOUT file, i.e., screen. There is one important and extra feature available which is first line to be printed `Content-type:text/html\r\n\r\n`. This line is sent back to the browser and it specifies the content type to be displayed on the browser screen.

By now you must have understood basic concept of CGI and you can write many complicated CGI programs using Python. This script can interact with any other external system also to exchange information such as RDBMS.

## HTTP Header

The line `Content-type:text/html\r\n\r\n` is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form –

HTTP Field Name: Field Content

For Example

`Content-type: text/html\r\n\r\n`

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

| Sr.No. | Header & Description                                                                                                                                                            |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | <b>Content-type:</b>                                                                                                                                                            |
| 1      | A MIME string defining the format of the file being returned. Example is <code>Content-type:text/html</code>                                                                    |
|        | <b>Expires: Date</b>                                                                                                                                                            |
| 2      | The date the information becomes invalid. It is used by the browser to decide when a page needs to be refreshed. A valid date string is in the format 01 Jan 1998 12:00:00 GMT. |
|        | <b>Location: URL</b>                                                                                                                                                            |
| 3      | The URL that is returned instead of the URL requested. You can use this field to redirect a request to any file.                                                                |
|        | <b>Last-modified: Date</b>                                                                                                                                                      |
| 4      | The date of last modification of the resource.                                                                                                                                  |
| 5      | <b>Content-length: N</b>                                                                                                                                                        |

|   |                                                                                                                                 |
|---|---------------------------------------------------------------------------------------------------------------------------------|
|   | The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file. |
| 6 | <b>Set-Cookie: String</b>                                                                                                       |
|   | Set the cookie passed through the <i>string</i>                                                                                 |

## CGI Environment Variables

All the CGI programs have access to the following environment variables. These variables play an important role while writing any CGI program.

| Sr.No. | Variable Name & Description                                                                                                                                          |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>CONTENT_TYPE</b><br>The data type of the content. Used when the client is sending attached content to the server. For example, file upload.                       |
| 2      | <b>CONTENT_LENGTH</b><br>The length of the query information. It is available only for POST requests.                                                                |
| 3      | <b>HTTP_COOKIE</b><br>Returns the set cookies in the form of key & value pair.                                                                                       |
| 4      | <b>HTTP_USER_AGENT</b><br>The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.      |
| 5      | <b>PATH_INFO</b><br>The path for the CGI script.                                                                                                                     |
| 6      | <b>QUERY_STRING</b><br>The URL-encoded information that is sent with GET method request.                                                                             |
| 7      | <b>REMOTE_ADDR</b><br>The IP address of the remote host making the request. This is useful for logging or for authentication.                                        |
| 8      | <b>REMOTE_HOST</b><br>The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address. |
| 9      | <b>REQUEST_METHOD</b><br>The method used to make the request. The most common methods are GET and POST.                                                              |
| 10     | <b>SCRIPT_FILENAME</b><br>The full path to the CGI script.                                                                                                           |
| 11     | <b>SCRIPT_NAME</b><br>The name of the CGI script.                                                                                                                    |

|    |                                                             |
|----|-------------------------------------------------------------|
| 12 | <b>SERVER_NAME</b>                                          |
|    | The server's hostname or IP Address                         |
| 13 | <b>SERVER_SOFTWARE</b>                                      |
|    | The name and version of the software the server is running. |

Here is small CGI program to list out all the CGI variables. Click this link to see the result  
[Get Environment](#)

```
import os

print ("Content-type: text/html\r\n\r\n");
print ("<font size=+1>Environment</font><br>");
for param in os.environ.keys():
    print ("<b>%20s</b>: %s<br>" % (param, os.environ[param]))
```

## GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

### Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows –

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

- The GET method is the default method to pass information from the browser to the web server and it produces a long string that appears in your browser's Location:box.
- Never use GET method if you have password or other sensitive information to pass to the server.
- The GET method has size limitation: only 1024 characters can be sent in a request string.
- The GET method sends information using QUERY\_STRING header and will be accessible in your CGI Program through QUERY\_STRING environment variable.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

### Simple URL Example: Get Method

Here is a simple URL, which passes two values to hello\_get.py program using GET method.

```
/cgi-bin/hello_get.py?first_name=Malhar&last_name=Lathkar
```

Given below is the hello\_get.py script to handle the input given by web browser. We are going to use the cgi module, which makes it very easy to access the passed information –

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')

print ("Content-type:text/html")
print()
print ("<html>")
print ('<head>')
print ("<title>Hello - Second CGI Program</title>")
print ('</head>')
print ('<body>')
print ("<h2>Hello %s %s</h2>" % (first_name, last_name))
print ('</body>')
print ('</html>')
```

This would generate the following result –

Hello Malhar Lathkar

## Simple FORM Example:GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script hello\_get.py to handle this input.

```
<form action = "/cgi-bin/hello_get.py" method = "get">
    First Name: <input type = "text" name = "first_name"> <br />

    Last Name: <input type = "text" name = "last_name" />
    <input type = "submit" value = "Submit" />
</form>
```

Here is the actual output of the above form, you enter First and Last Name and then click submit button to see the result.

First Name:

Last Name:

## Passing Information Using POST Method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

Below is same hello\_get.py script which handles GET as well as POST method.

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

Let us take again same example as above which passes two values using HTML FORM and submit button. We use same CGI script hello\_get.py to handle this input.

```
<form action = "/cgi-bin/hello_get.py" method = "post">
First Name: <input type = "text" name = "first_name"><br />
Last Name: <input type = "text" name = "last_name" />

<input type = "submit" value = "Submit" />
```

```
</form>
```

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

First Name:   
 Last Name:

## Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes –

```
<form action = "/cgi-bin/checkbox.cgi" method = "POST" target = "_blank">
  <input type = "checkbox" name = "maths" value = "on" /> Maths
  <input type = "checkbox" name = "physics" value = "on" /> Physics
  <input type = "submit" value = "Select Subject" />
</form>
```

The result of this code is the following form –

Maths       Physics

Below is checkbox.cgi script to handle input given by web browser for checkbox button.

```
# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('maths'):
    math_flag = "ON"
else:
    math_flag = "OFF"

if form.getvalue('physics'):
    physics_flag = "ON"
else:
```

```

physics_flag = "OFF"

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Checkbox - Third CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> CheckBox Maths is : %s</h2>" % math_flag
print "<h2> CheckBox Physics is : %s</h2>" % physics_flag
print "</body>"
print "</html>"

```

## Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio buttons –

```

<form action = "/cgi-bin/radiobutton.py" method = "post" target = "_blank">
    <input type = "radio" name = "subject" value = "maths" /> Maths
    <input type = "radio" name = "subject" value = "physics" /> Physics
    <input type = "submit" value = "Select Subject" />
</form>

```

The result of this code is the following form –

Maths  Physics

Below is radiobutton.py script to handle input given by web browser for radio button –

```

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('subject'):
    subject = form.getvalue('subject')
else:
    subject = "Not set"

```

```

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Radio - Fourth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"

```

## Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

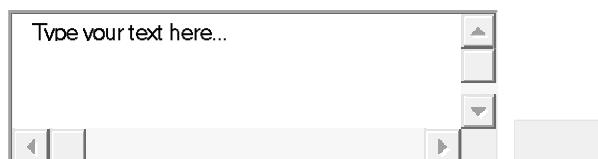
Here is example HTML code for a form with a TEXTAREA box –

```

<form action = "/cgi-bin/textarea.py" method = "post" target = "_blank">
    <textarea name = "textcontent" cols = "40" rows = "4">
        Type your text here...
    </textarea>
    <input type = "submit" value = "Submit" />
</form>

```

The result of this code is the following form –



Below is textarea.cgi script to handle input given by web browser –

```

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('textcontent'):
    text_content = form.getvalue('textcontent')
else:

```

```

text_content = "Not entered"

print "Content-type:text/html\r\n\r\n"

print "<html>"
print "<head>";
print "<title>Text Area - Fifth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Entered Text Content is %s</h2>" % text_content
print "</body>"

```

## Passing Drop Down Box Data to CGI Program

Drop Down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box –

```

<form action = "/cgi-bin/dropdown.py" method = "post" target = "_blank">
    <select name = "dropdown">
        <option value = "Maths" selected>Maths</option>
        <option value = "Physics">Physics</option>
    </select>
    <input type = "submit" value = "Submit"/>
</form>

```

The result of this code is the following form –



Below is dropdown.py script to handle input given by web browser.

```

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('dropdown'):
    subject = form.getvalue('dropdown')

```

```

else:
    subject = "Not entered"

print "Content-type:text/html\r\n\r\n"

print "<html>"
print "<head>"
print "<title>Dropdown Box - Sixth CGI Program</title>"
print "</head>"
print "<body>"
print "<h2> Selected Subject is %s</h2>" % subject
print "</body>"
print "</html>"

```

## Using Cookies in CGI

HTTP protocol is a stateless protocol. For a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. How to maintain user's session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

### How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields –

- **Expires** – The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain** – The domain name of your site.
- **Path** – The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure** – If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name = Value** – Cookies are set and retrieved in the form of key and value pairs.

### Setting up Cookies

It is very easy to send cookies to browser. These cookies are sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. Setting the cookies is done as follows –

```
print "Set-Cookie:UserID = XYZ; \r\n"
```

```

print "Set-Cookie:Password = XYZ123;\r\n"
print "Set-Cookie:Expires = Tuesday, 31-Dec-2007 23:12:40 GMT;\r\n"
print "Set-Cookie:Domain = www.tutorialspoint.com;\r\n"
print "Set-Cookie:Path = /perl;\r\n"
print "Content-type:text/html\r\n\r\n"
.....Rest of the HTML Content....

```

From this example, you must have understood how to set cookies. We use Set-Cookie HTTP header to set cookies.

It is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line "Content-type:text/html\r\n\r\n".

## Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP\_COOKIE and they will have following form –

```
key1 = value1;key2 = value2;key3 = value3....
```

Here is an example of how to retrieve cookies.

```

# Import modules for CGI handling
from os import environ
import cgi, cgitb

if environ.has_key('HTTP_COOKIE'):
    for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
        (key, value) = split(cookie, '=')
        if key == "UserID":
            user_id = value

        if key == "Password":
            password = value

print "User ID = %s" % user_id
print "Password = %s" % password

```

This produces the following result for the cookies set by above script –

```
User ID = XYZ
Password = XYZ123
```

## File Upload Example

To upload a file, the HTML form must have the enctype attribute set to multipart/form-data. The input tag with the file type creates a "Browse" button.

```
<html>
  <body>
    <form enctype = "multipart/form-data" action = "save_file.py" method =
"post">
      <p>File: <input type = "file" name = "filename" /></p>
      <p><input type = "submit" value = "Upload" /></p>
    </form>
  </body>
</html>
```

The result of this code is the following form –

File:  Choose File No file chosen

Above example has been disabled intentionally to save people uploading file on our server, but you can try above code with your server.

Here is the script save\_file.py to handle file upload –

```
import cgi, os
import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# Get filename here.
fileitem = form['filename']

# Test if the file was uploaded
if fileitem.filename:
    # strip leading path from file name to avoid
    # directory traversal attacks
    fn = os.path.basename(fileitem.filename)
    open('/tmp/' + fn, 'wb').write(fileitem.file.read())

    message = 'The file "' + fn + '" was uploaded successfully'
```

```

else:
    message = 'No file was uploaded'

print """\
Content-Type: text/html\n
<html>
  <body>
    <p>%s</p>
  </body>
</html>
""" % (message,)

```

If you run the above script on Unix/Linux, then you need to take care of replacing file separator as follows, otherwise on your windows machine open() statement should work fine.

```
fn = os.path.basename(fileitem.filename.replace("\\", "/"))
```

## How to Raise a "File Download" Dialog Box?

Sometimes, it is desired that you want to give option where a user can click a link and it will pop up a "File Download" dialog box to the user instead of displaying actual content. This is very easy and can be achieved through HTTP header. This HTTP header is be different from the header mentioned in previous section.

For example, if you want make a FileName file downloadable from a given link, then its syntax is as follows –

```

# HTTP Header
print "Content-Type:application/octet-stream; name = \"FileName\"\r\n";
print "Content-Disposition: attachment; filename = \"FileName\"\r\n\r\n";

# Actual File Content will go here.
fo = open("foo.txt", "rb")

str = fo.read();
print str

# Close opend file
fo.close()

```

# 198. Python - XML Processing

XML is a portable, open-source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

## What is XML?

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

XML is extremely useful for keeping track of small to medium amounts of data without requiring an SQL- based backbone.

## XML Parser Architectures and APIs.

The Python standard library provides a minimal but useful set of interfaces to work with XML. All the submodules for XML processing are available in the `xml` package.

- **`xml.etree.ElementTree`** – the ElementTree API, a simple and lightweight XML processor
- **`xml.dom`** – the DOM API definition.
- **`xml.dom.minidom`** – a minimal DOM implementation.
- **`xml.dom.pulldom`** – support for building partial DOM trees.
- **`xml.sax`** – SAX2 base classes and convenience functions.
- **`xml.parsers.expat`** – the Expat parser binding.

The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

- **Simple API for XML (SAX)** – Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from the disk and the entire file is never stored in the memory.
- **Document Object Model (DOM)** – This is a World Wide Web Consortium recommendation wherein the entire file is read into the memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously cannot process information as fast as DOM, when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on many small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other, there is no reason why you cannot use them both for large projects.

For all our XML code examples, let us use a simple XML file `movies.xml` as an input –

```
<collection shelf="New Arrivals">
  <movie title="Enemy Behind">
    <type>War, Thriller</type>
    <format>DVD</format>
```

```

<year>2003</year>
<rating>PG</rating>
<stars>10</stars>
<description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
    <type>Anime, Science Fiction</type>
    <format>DVD</format>
    <year>1989</year>
    <rating>R</rating>
    <stars>8</stars>
    <description>A schientific fiction</description>
</movie>
<movie title="Trigun">
    <type>Anime, Action</type>
    <format>DVD</format>
    <episodes>4</episodes>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
    <type>Comedy</type>
    <format>VHS</format>
    <rating>PG</rating>
    <stars>2</stars>
    <description>Viewable boredom</description>
</movie>
</collection>

```

## Parsing XML with SAX APIs

SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing `xml.sax.ContentHandler`.

Your ContentHandler handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.

The methods `startDocument` and `endDocument` are called at the start and the end of the XML file. The method `characters(text)` is passed the character data of the XML file via the parameter `text`.

The `ContentHandler` is called at the start and end of each element. If the parser is not in namespace mode, the methods `startElement(tag, attributes)` and `endElement(tag)` are called; otherwise, the corresponding methods `startElementNS` and `endElementNS` are called. Here, `tag` is the element tag, and `attributes` is an `Attributes` object.

Here are other important methods to understand before proceeding –

### The `make_parser` Method

The following method creates a new parser object and returns it. The parser object created will be of the first parser type, the system finds.

```
xml.sax.make_parser( [parser_list] )
```

Here is the detail of the parameters –

- **`parser_list`** – The optional argument consisting of a list of parsers to use, which must all implement the `make_parser` method.

### The `parse` Method

The following method creates a SAX parser and uses it to parse a document.

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler] )
```

Here are the details of the parameters –

- **`xmlfile`** – This is the name of the XML file to read from.
- **`contenthandler`** – This must be a `ContentHandler` object.
- **`errorhandler`** – If specified, `errorhandler` must be a SAX `ErrorHandler` object.

### The `parseString` Method

There is one more method to create a SAX parser and to parse the specified XML string.

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

Here are the details of the parameters –

- **`xmlstring`** – This is the name of the XML string to read from.
- **`contenthandler`** – This must be a `ContentHandler` object.
- **`errorhandler`** – If specified, `errorhandler` must be a SAX `ErrorHandler` object.

### Example

```
import xml.sax

class MovieHandler( xml.sax.ContentHandler ):

    def __init__(self):
        self.CurrentData = ""
        self.type = ""
        self.format = ""
        self.year = ""
```

```
self.rating = ""  
self.stars = ""  
self.description = ""  
  
# Call when an element starts  
def startElement(self, tag, attributes):  
    self.CurrentData = tag  
    if tag == "movie":  
        print ("*****Movie*****")  
        title = attributes["title"]  
        print ("Title:", title)  
  
# Call when an elements ends  
def endElement(self, tag):  
    if self.CurrentData == "type":  
        print ("Type:", self.type)  
    elif self.CurrentData == "format":  
        print ("Format:", self.format)  
    elif self.CurrentData == "year":  
        print ("Year:", self.year)  
    elif self.CurrentData == "rating":  
        print ("Rating:", self.rating)  
    elif self.CurrentData == "stars":  
        print ("Stars:", self.stars)  
    elif self.CurrentData == "description":  
        print ("Description:", self.description)  
    self.CurrentData = ""  
  
# Call when a character is read  
def characters(self, content):  
    if self.CurrentData == "type":  
        self.type = content  
    elif self.CurrentData == "format":  
        self.format = content  
    elif self.CurrentData == "year":  
        self.year = content
```

```

        elif self.CurrentData == "rating":
            self.rating = content
        elif self.CurrentData == "stars":
            self.stars = content
        elif self.CurrentData == "description":
            self.description = content

if ( __name__ == "__main__"):

    # create an XMLReader
    parser = xml.sax.make_parser()

    # turn off namespaces
    parser.setFeature(xml.sax.handler.feature_namespaces, 0)

    # override the default ContextHandler
    Handler = MovieHandler()
    parser.setContentHandler( Handler )

parser.parse("movies.xml")

```

This would produce the following result –

```

*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Year: 2003
Rating: PG
Stars: 10
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Year: 1989
Rating: R

```

```

Stars: 8
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Stars: 10
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Stars: 2
Description: Viewable boredom

```

For a complete detail on SAX API documentation, please refer to standard Python SAX APIs.

## Parsing XML with DOM APIs

The Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying the XML documents.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.

Here is the easiest way to load an XML document quickly and to create a minidom object using the `xml.dom` module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.

The sample phrase calls the `parse( file [,parser] )` function of the minidom object to parse the XML file, designated by `file` into a DOM tree object.

```

from xml.dom.minidom import parse
import xml.dom.minidom

# Open XML document using minidom parser
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
    print ("Root element : %s" % collection.getAttribute("shelf"))

```

```

# Get all the movies in the collection
movies = collection.getElementsByTagName("movie")

# Print detail of each movie.
for movie in movies:
    print ("*****Movie*****")
    if movie.hasAttribute("title"):
        print ("Title: %s" % movie.getAttribute("title"))

        type = movie.getElementsByTagName('type')[0]
        print ("Type: %s" % type.childNodes[0].data)
        format = movie.getElementsByTagName('format')[0]
        print ("Format: %s" % format.childNodes[0].data)
        rating = movie.getElementsByTagName('rating')[0]
        print ("Rating: %s" % rating.childNodes[0].data)
        description = movie.getElementsByTagName('description')[0]
        print ("Description: %s" % description.childNodes[0].data)

```

This would produce the following output –

```

Root element : New Arrivals

*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Rating: PG
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Rating: R
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action

```

```

Format: DVD
Rating: PG
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Description: Viewable boredom

```

For a complete detail on DOM API documentation, please refer to standard Python DOM APIs.

## ElementTree XML API

The `xml` package has an `ElementTree` module. This is a simple and lightweight XML processor API.

XML is a tree-like hierarchical data format. The '`ElementTree`' in this module treats the whole XML document as a tree. The '`Element`' class represents a single node in this tree. Reading and writing operations on XML files are done on the `ElementTree` level. Interactions with a single XML element and its sub-elements are done on the `Element` level.

## Create an XML File

The tree is a hierarchical structure of elements starting with root followed by other elements. Each element is created by using the `Element()` function of this module.

```

import xml.etree.ElementTree as et
e=et.Element('name')

```

Each element is characterized by a tag and `attrib` attribute which is a dict object. For tree's starting element, `attrib` is an empty dictionary.

```

>>> root=xml.Element('employees')
>>> root.tag
'employees'
>>> root.attrib
{}

```

You may now set up one or more child elements to be added under the root element. Each child may have one or more sub elements. Add them using the `SubElement()` function and define its `text` attribute.

```

child=xml.Element("employee")
nm = xml.SubElement(child, "name")

```

```
nm.text = student.get('name')
age = xml.SubElement(child, "salary")
age.text = str(student.get('salary'))
```

Each child is added to root by append() function as –

```
root.append(child)
```

After adding required number of child elements, construct a tree object by elementTree() function –

```
tree = et.ElementTree(root)
```

The entire tree structure is written to a binary file by tree object's write() function –

```
f=open('employees.xml', "wb")
tree.write(f)
```

### Example

In this example, a tree is constructed out of a list of dictionary items. Each dictionary item holds key-value pairs describing a student data structure. The tree so constructed is written to 'myfile.xml'

```
import xml.etree.ElementTree as et
employees=[{'name':'aaa','age':21,'sal':5000},{'name':xyz,'age':22,'sal':6000}]
root = et.Element("employees")
for employee in employees:
    child=xml.SubElement(root, "employee")
    child.append(child)
    nm = xml.SubElement(child, "name")
    nm.text = student.get('name')
    age = xml.SubElement(child, "age")
    age.text = str(student.get('age'))
    sal=xml.SubElement(child, "sal")
    sal.text=str(student.get('sal'))
tree = et.ElementTree(root)
with open('employees.xml', "wb") as fh:
    tree.write(fh)
```

The 'myfile.xml' is stored in current working directory.

```
<employees><employee><name>aaa</name><age>21</age><sal>5000</sal></employee><employee><name>xyz</name><age>22</age><sal>60</sal></employee></employees>
```

## Parse an XML File

Let us now read back the 'myfile.xml' created in above example. For this purpose, following functions in ElementTree module will be used –

**ElementTree()** – This function is overloaded to read the hierarchical structure of elements to a tree objects.

```
tree = et.ElementTree(file='students.xml')
```

**getroot()** – This function returns root element of the tree.

```
root = tree.getroot()
```

You can obtain the list of sub-elements one level below an element.

```
children = list(root)
```

In the following example, elements and sub-elements of the 'myfile.xml' are parsed into a list of dictionary items.

### Example

```
import xml.etree.ElementTree as et
tree = et.ElementTree(file='employees.xml')
root = tree.getroot()
employees=[]
children = list(root)
for child in children:
    employee={}
    pairs = list(child)
    for pair in pairs:
        employee[pair.tag]=pair.text
    employees.append(employee)
print (employees)
```

It will produce the following output –

```
[{'name': 'aaa', 'age': '21', 'sal': '5000'}, {'name': 'xyz', 'age': '22', 'sal': '6000'}]
```

## Modify an XML file

We shall use iter() function of Element. It creates a tree iterator for a given tag with the current element as the root. The iterator iterates over this element and all elements below it in document (depth) first order.

Let us build iterator for all 'marks' subelements and increment text of each sal tag by 100.

```
import xml.etree.ElementTree as et
tree = et.ElementTree(file='students.xml')
```

```
root = tree.getroot()
for x in root.iter('sal'):
    s=int (x.text)
    s=s+100
    x.text=str(s)
with open("employees.xml", "wb") as fh:
    tree.write(fh)
```

Our 'employees.xml' will now be modified accordingly. We can also use set() to update value of a certain key.

```
x.set(marks, str(mark))
```

# 199. Python - GUI Programming

Python provides various options for developing graphical user interfaces (GUIs). The most important features are listed below:

- **Tkinter** – Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look at this option in this chapter.
- **wxPython** – This is an open-source Python interface for wxWidgets GUI toolkit. You can find a complete tutorial on WxPython here.
- **PyQt** – This is also a Python interface for a popular cross-platform Qt GUI library. TutorialsPoint has a very good tutorial on PyQt5 here.
- **PyGTK** – PyGTK is a set of wrappers written in Python and C for GTK + GUI library. The complete PyGTK tutorial is available here.
- **PySimpleGUI** – PySimpleGui is an open source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide and WxPython toolkits. For a detailed PySimpleGUI tutorial, click here.
- **Pygame** – Pygame is a popular Python library used for developing video games. It is free, open source and cross-platform wrapper around Simple DirectMedia Library (SDL). For a comprehensive tutorial on Pygame, visit this link.
- **Jython** – Jython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machine <http://www.jython.org>.

There are many other interfaces available, which you can find them on the net.

## Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

The tkinter package includes following modules –

- **Tkinter** – Main Tkinter module.
- **tkinter.colorchooser** – Dialog to let the user choose a color.
- **tkinter.commondialog** – Base class for the dialogs defined in the other modules listed here.
- **tkinter.dialog** – Common dialogs to allow the user to specify a file to open or save.
- **tkinter.font** – Utilities to help work with fonts.
- **tkinter.messagebox** – Access to standard Tk dialog boxes.
- **tkinter.scrolledtext** – Text widget with a vertical scroll bar built in.
- **tkinter.simpledialog** – Basic dialogs and convenience functions.
- **tkinter.ttk** – Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main tkinter module.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps.

- Import the Tkinter module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

**Example**

```
# note that module name has changed from Tkinter in Python 2
# to tkinter in Python 3

import tkinter
top = tkinter.Tk()

# Code to add widgets will go here...
top.mainloop()
```

This would create a following window –



When the program becomes more complex, using an object-oriented programming approach makes the code more organized.

```
import tkinter as tk

class App(tk.Tk):
    def __init__(self):
        super().__init__()

app = App()
app.mainloop()
```

**Tkinter Widgets**

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 19 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table –

| Sr.No. | Operator & Description |
|--------|------------------------|
| 1      | <a href="#">Button</a> |

|    |                                                                                                                                                               |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | The Button widget is used to display the buttons in your application.                                                                                         |
| 2  | <a href="#">Canvas</a><br>The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.                       |
| 3  | <a href="#">Checkbutton</a><br>The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.   |
| 4  | <a href="#">Entry</a><br>The Entry widget is used to display a single-line text field for accepting values from a user.                                       |
| 5  | <a href="#">Frame</a><br>The Frame widget is used as a container widget to organize other widgets.                                                            |
| 6  | <a href="#">Label</a><br>The Label widget is used to provide a single-line caption for other widgets. It can also contain images.                             |
| 7  | <a href="#">Listbox</a><br>The Listbox widget is used to provide a list of options to a user.                                                                 |
| 8  | <a href="#">Menubutton</a><br>The Menubutton widget is used to display menus in your application.                                                             |
| 9  | <a href="#">Menu</a><br>The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.                        |
| 10 | <a href="#">Message</a><br>The Message widget is used to display multiline text fields for accepting values from a user.                                      |
| 11 | <a href="#">Radiobutton</a><br>The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. |
| 12 | <a href="#">Scale</a><br>The Scale widget is used to provide a slider widget.                                                                                 |
| 13 | <a href="#">Scrollbar</a><br>The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.                                 |

|    |                                                                                                                                  |
|----|----------------------------------------------------------------------------------------------------------------------------------|
| 14 | <a href="#"><u>Text</u></a>                                                                                                      |
|    | The Text widget is used to display text in multiple lines.                                                                       |
| 15 | <a href="#"><u>Toplevel</u></a>                                                                                                  |
|    | The Toplevel widget is used to provide a separate window container.                                                              |
| 16 | <a href="#"><u>Spinbox</u></a>                                                                                                   |
|    | The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values. |
| 17 | <a href="#"><u>PanedWindow</u></a>                                                                                               |
|    | A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.                   |
| 18 | <a href="#"><u>LabelFrame</u></a>                                                                                                |
|    | A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.    |
| 19 | <a href="#"><u>tkMessageBox</u></a>                                                                                              |
|    | This module is used to display message boxes in your applications.                                                               |

Let us study these widgets in detail.

## Standard Attributes

Let us look at how some of the common attributes, such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors
- Relief styles
- Bitmaps
- Cursors

Let us study them briefly –

## Geometry Management

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- [The pack\(\) Method](#) – This geometry manager organizes widgets in blocks before placing them in the parent widget.
- [The grid\(\) Method](#) – This geometry manager organizes widgets in a table-like structure in the parent widget.
- [The place\(\) Method](#) – This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Let us study the geometry management methods briefly –

### SimpleDialog

The `simpledialog` module in `tkinter` package includes a dialog class and convenience functions for accepting user input through a modal dialog. It consists of a label, an entry widget and two buttons Ok and Cancel. These functions are –

- **`askfloat(title, prompt, **kw)`** – Accepts a floating point number.
- **`askinteger(title, prompt, **kw)`** – Accepts an integer input.
- **`askstring(title, prompt, **kw)`** – Accepts a text input from the user.

The above three functions provide dialogs that prompt the user to enter a value of the desired type. If Ok is pressed, the input is returned, if Cancel is pressed, None is returned.

### askinteger

```
from tkinter.simpledialog import askinteger
from tkinter import *
from tkinter import messagebox
top = Tk()

top.geometry("100x100")
def show():
    num = askinteger("Input", "Input an Integer")
    print(num)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following output –



### askfloat

```
from tkinter.simpledialog import askfloat
from tkinter import *
```

```

top = Tk()

top.geometry("100x100")

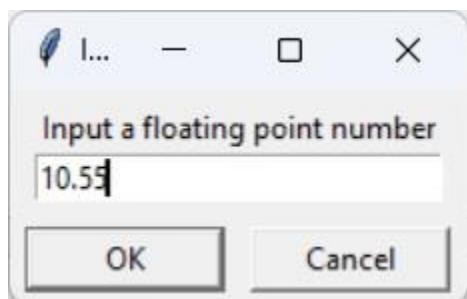
def show():
    num = askfloat("Input", "Input a floating point number")
    print(num)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()

```

It will produce the following output –



### **askstring**

```

from tkinter.simpledialog import askstring
from tkinter import *

top = Tk()

top.geometry("100x100")

def show():
    name = askstring("Input", "Enter your name")
    print(name)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()

```

It will produce the following output –



## The FileDialog Module

The `filedialog` module in Tkinter package includes a `FileDialog` class. It also defines convenience functions that enable the user to perform open file, save file, and open directory activities.

- `filedialog.asksaveasfilename()`
- `filedialog.asksaveasfile()`
- `filedialog.askopenfilename()`
- `filedialog.askopenfile()`
- `filedialog.askdirectory()`
- `filedialog.askopenfilenames()`
- `filedialog.askopenfiles()`

### `askopenfile`

This function lets the user choose a desired file from the file system. The file dialog window has Open and Cancel buttons. The file name along with its path is returned when Ok is pressed, None if Cancel is pressed.

```
from tkinter.filedialog import askopenfile
from tkinter import *

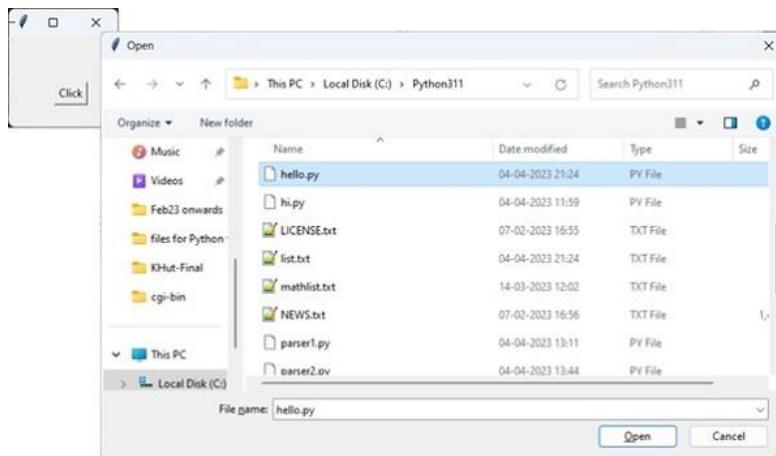
top = Tk()

top.geometry("100x100")
def show():
    filename = askopenfile()
    print(filename)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following output –



## ColorChooser

The colorchooser module included in tkinter package has the feature of letting the user choose a desired color object through the color dialog. The `askcolor()` function presents with the color dialog with predefined color swatches and facility to choose custom color by setting RGB values. The dialog returns a tuple of RGB values of chosen color as well as its hex value.

```
from tkinter.colorchooser import askcolor
from tkinter import *

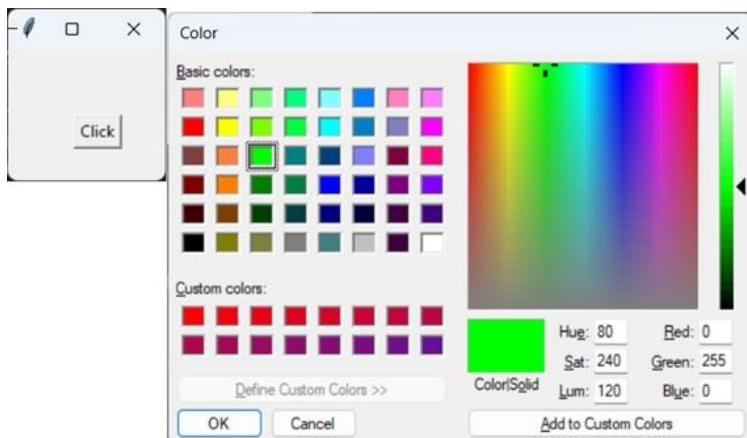
top = Tk()

top.geometry("100x100")
def show():
    color = askcolor()
    print(color)

B = Button(top, text ="Click", command = show)
B.place(x=50,y=50)

top.mainloop()
```

It will produce the following output –



```
((0, 255, 0), '#00ff00')
```

## ttk module

The term ttk stands from Tk Themed widgets. The ttk module was introduced with Tk 8.5 onwards. It provides additional benefits including anti-aliased font rendering under X11 and window transparency. It provides theming and styling support for Tkinter.

The ttk module comes bundled with 18 widgets, out of which 12 are already present in Tkinter. Importing ttk over-writes these widgets with new ones which are designed to have a better and more modern look across all platforms.

The 6 new widgets in ttk are, the Combobox, Separator, Sizegrip, Treeview, Notebook and ProgressBar.

To override the basic Tk widgets, the import should follow the Tk import –

```
from tkinter import *
from tkinter.ttk import *
```

The original Tk widgets are automatically replaced by tkinter.ttk widgets. They are Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar.

New widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

The new widgets in ttk module are –

- **Notebook** – This widget manages a collection of "tabs" between which you can swap, changing the currently displayed window.
- **ProgressBar** – This widget is used to show progress or the loading process through the use of animations.
- **Separator** – Used to separate different widgets using a separator line.
- **Treeview** – This widget is used to group together items in a tree-like hierarchy. Each item has a textual label, an optional image, and an optional list of data values.
- **ComboBox** – Used to create a dropdown list of options from which the user can select one.
- **Sizegrip** – Creates a little handle near the bottom-right of the screen, which can be used to resize the window.

## Combobox Widget

The Python ttk Combobox presents a drop down list of options and displays them one at a time. It is a sub class of the widget Entry. Hence it inherits many options and methods from the Entry class.

### Syntax

```
from tkinter import ttk

Combo = ttk.Combobox(master, values.....)
```

The get() function to retrieve the current value of the Combobox.

### Example

```
from tkinter import *
from tkinter import ttk

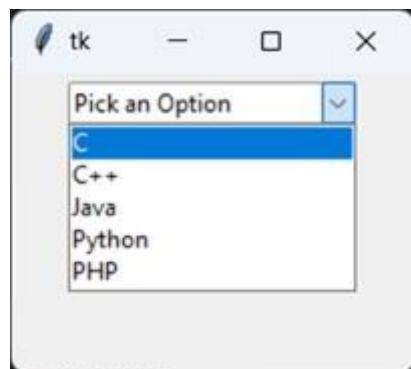
top = Tk()
top.geometry("200x150")

frame = Frame(top)
frame.pack()

langs = ["C", "C++", "Java",
         "Python", "PHP"]

Combo = ttk.Combobox(frame, values = langs)
Combo.set("Pick an Option")
Combo.pack(padx = 5, pady = 5)
top.mainloop()
```

It will produce the following output –



## Progressbar

The ttk Progressbar widget, and how it can be used to create loading screens or show the progress of a current task.

### Syntax

```
ttk.Progressbar(parent, orient, length, mode)
```

#### Parameters

- **Parent** – The container in which the Progressbar is to be placed, such as root or a Tkinter frame.
- **Orient** – Defines the orientation of the Progressbar, which can be either vertical or horizontal.
- **Length** – Defines the width of the Progressbar by taking in an integer value.
- **Mode** – There are two options for this parameter, determinate and indeterminate.

#### Example

The code given below creates a progressbar with three buttons which are linked to three different functions.

The first function increments the "value" or "progress" in the progressbar by 20. This is done with the step() function which takes an integer value to change progress amount. (Default is 1.0)

The second function decrements the "value" or "progress" in the progressbar by 20.

The third function prints out the current progress level in the progressbar.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
frame= ttk.Frame(root)

def increment():
    progressBar.step(20)

def decrement():
    progressBar.step(-20)

def display():
    print(progressBar["value"])

progressBar= ttk.Progressbar(frame, mode='determinate')
progressBar.pack(padx = 10, pady = 10)

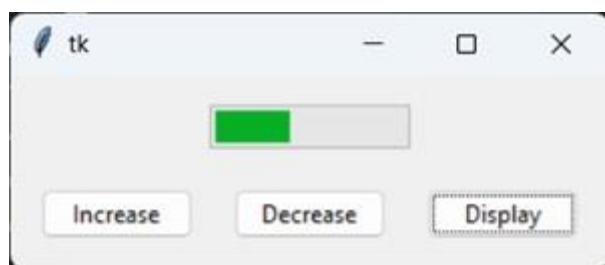
button= ttk.Button(frame, text= "Increase", command= increment)
```

```
button.pack(padx = 10, pady = 10, side = tk.LEFT)

button= ttk.Button(frame, text= "Decrease", command= decrement)
button.pack(padx = 10, pady = 10, side = tk.LEFT)
button= ttk.Button(frame, text= "Display", command= display)
button.pack(padx = 10, pady = 10, side = tk.LEFT)

frame.pack(padx = 5, pady = 5)
root.mainloop()
```

It will produce the following output –



## Notebook

Tkinter ttk module has a new useful widget called Notebook. It is a collection of containers (e.g frames) which have many widgets as children inside.

Each "tab" or "window" has a tab ID associated with it, which is used to determine which tab to swap to.

You can swap between these containers like you would on a regular text editor.

### Syntax

```
notebook = ttk.Notebook(master, *options)
```

### Example

In this example, we will add 3 windows to our Notebook widget in two different ways. The first method involves the add() function, which simply appends a new tab to the end. The other method is the insert() function which can be used to add a tab to a specific position.

The add() function takes one mandatory parameter which is the container widget to be added, and the rest are optional parameters such as text (text to be displayed as tab title), image and compound.

The insert() function requires a tab\_id, which defines the location where it should be inserted. The tab\_id can be either an index value or it can be string literal like "end", which will append it to the end.

```
import tkinter as tk
from tkinter import ttk
```

```
root = tk.Tk()
nb = ttk.Notebook(root)

# Frame 1 and 2
frame1 = ttk.Frame(nb)
frame2 = ttk.Frame(nb)

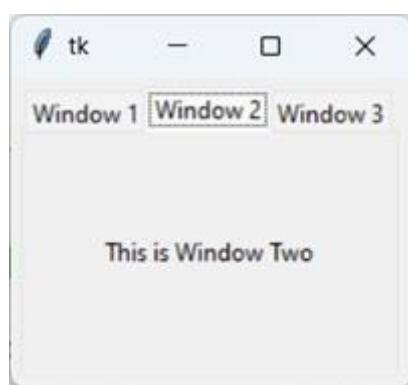
label1 = ttk.Label(frame1, text = "This is Window One")
label1.pack(pady = 50, padx = 20)
label2 = ttk.Label(frame2, text = "This is Window Two")
label2.pack(pady = 50, padx = 20)

frame1.pack(fill= tk.BOTH, expand=True)
frame2.pack(fill= tk.BOTH, expand=True)
nb.add(frame1, text = "Window 1")
nb.add(frame2, text = "Window 2")

frame3 = ttk.Frame(nb)
label3 = ttk.Label(frame3, text = "This is Window Three")
label3.pack(pady = 50, padx = 20)
frame3.pack(fill= tk.BOTH, expand=True)
nb.insert("end", frame3, text = "Window 3")
nb.pack(padx = 5, pady = 5, expand = True)

root.mainloop()
```

It will produce the following output –



## Treeview

The Treeview widget is used to display items in a tabular or hierarchical manner. It has support for features like creating rows and columns for items, as well as allowing items to have children as well, leading to a hierarchical format.

## Syntax

```
tree = ttk.Treeview(container, **options)
```

## Options

| Sr.No. | Option & Description                                                                                                                                                                                                                           |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <b>columns</b><br>A list of column names                                                                                                                                                                                                       |
| 2      | <b>displaycolumns</b><br>A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".                                               |
| 3      | <b>height</b><br>The number of rows visible.                                                                                                                                                                                                   |
| 4      | <b>padding</b><br>Specifies the internal padding for the widget. Can be either an integer or a list of 4 values.                                                                                                                               |
| 5      | <b>selectmode</b><br>One of "extended", "browse" or "none". If set to "extended" (default), multiple items can be selected. If "browse", only a single item can be selected at a time. If "none", the selection cannot be changed by the user. |
| 6      | <b>show</b><br>A list containing zero or more of the following values, specifying which elements of the tree to display. The default is "tree headings", i.e., show all elements.                                                              |

## Example

In this example we will create a simple Treeview ttk Widget and fill in some data into it. We have some data already stored in a list which will be reading and adding to the Treeview widget in our `read_data()` function.

We first need to define a list/tuple of column names. We have left out the column "Name" because there already exists a (default) column with a blank name.

We then assign that list/tuple to the `columns` option in Treeview, followed by defining the "headings", where the column is the actual column, whereas the heading is just the title of the column that appears when the widget is displayed. We give each a column a name. "#0" is the name of the default column.

The `tree.insert()` function has the following parameters –

- **Parent** – which is left as an empty string if there is none.
- **Position** – where we want to add the new item. To append, use `tk.END`
- **Iid** – which is the item ID used to later track the item in question.
- **Text** – to which we will assign the first value in the list (the name).

Value we will pass the other 2 values we obtained from the list.

### The Complete Code

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter import simpledialog

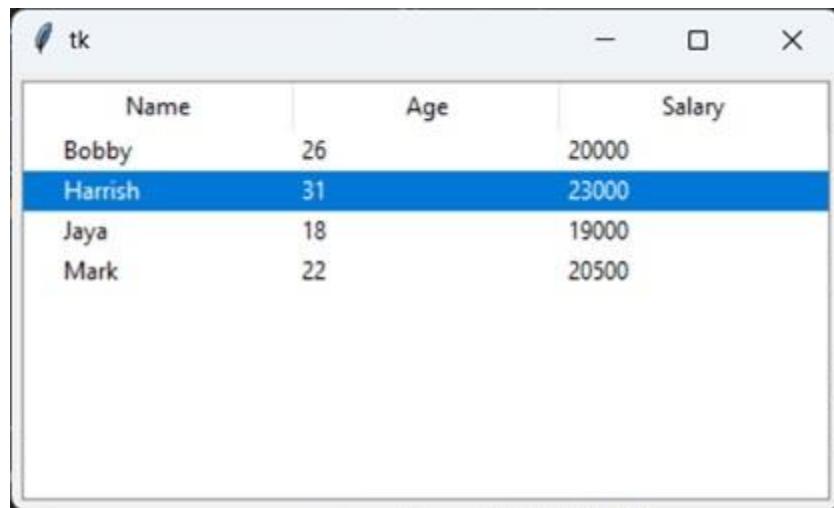
root = tk.Tk()
data = [
    ["Bobby", 26, 20000],
    ["Harrish", 31, 23000],
    ["Jaya", 18, 19000],
    ["Mark", 22, 20500],
]
index=0
def read_data():
    for index, line in enumerate(data):
        tree.insert('', tk.END, iid = index,
                    text = line[0], values = line[1:])
columns = ("age", "salary")

tree= ttk.Treeview(root, columns=columns ,height = 20)
tree.pack(padx = 5, pady = 5)

tree.heading('#0', text='Name')
tree.heading('age', text='Age')
tree.heading('salary', text='Salary')

read_data()
root.mainloop()
```

It will produce the following output –



| Name    | Age | Salary |
|---------|-----|--------|
| Bobby   | 26  | 20000  |
| Harrish | 31  | 23000  |
| Jaya    | 18  | 19000  |
| Mark    | 22  | 20500  |

## Sizegrip

The Sizegrip widget is basically a small arrow-like grip that is typically placed at the bottom-right corner of the screen. Dragging the Sizegrip across the screen also resizes the container to which it is attached to.

### Syntax

```
sizegrip = ttk.Sizegrip(parent, **options)
```

### Example

```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
root.geometry("100x100")

frame = ttk.Frame(root)
label = ttk.Label(root, text = "Hello World")
label.pack(padx = 5, pady = 5)
sizegrip = ttk.Sizegrip(frame)
sizegrip.pack(expand = True, fill = tk.BOTH, anchor = tk.SE)
frame.pack(padx = 10, pady = 10, expand = True, fill = tk.BOTH)

root.mainloop()
```

It will produce the following output –



## Separator

The ttk Separator widget is a very simple widget, that has just one purpose and that is to help "separate" widgets into groups/partitions by drawing a line between them. We can change the orientation of this line (separator) to either horizontal or vertical, and change its length/height.

### Syntax

```
separator = ttk.Separator(parent, **options)
```

The "orient", which can either be tk.VERTICAL or tk.HORIZONTAL, for a vertical and horizontal separator respectively.

### Example

Here we have created two Label widgets, and then created a Horizontal Separator between them.

```
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
root.geometry("200x150")

frame = ttk.Frame(root)

label = ttk.Label(frame, text = "Hello World")
label.pack(padx = 5)

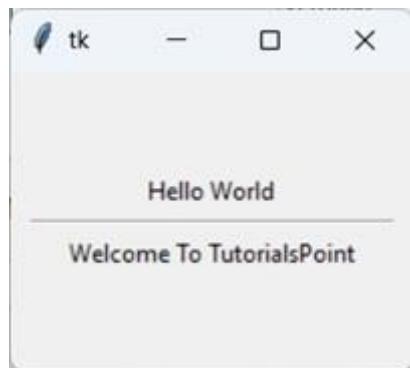
separator = ttk.Separator(frame,orient= tk.HORIZONTAL)
separator.pack(expand = True, fill = tk.X)

label = ttk.Label(frame, text = "Welcome To TutorialsPoint")
label.pack(padx = 5)
```

```
frame.pack(padx = 10, pady = 50, expand = True, fill = tk.BOTH)

root.mainloop()
```

It will produce the following output –



# 200. Python - Command-Line Arguments

## Python Command Line Arguments

Python Command Line Arguments provides a convenient way to accept some information at the command line while running the program. We usually pass these values along with the name of the Python script.

To run a Python program, we execute the following command in the command prompt terminal of the operating system. For example, in windows, the following command is entered in Windows command prompt terminal.

```
$ python script.py arg1 arg2 arg3
```

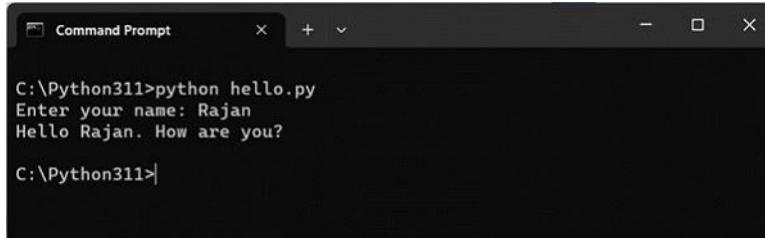
Here Python script name is script.py and rest of the three arguments - arg1 arg2 arg3 are command line arguments for the program.

If the program needs to accept input from the user, Python's `input()` function is used. When the program is executed from command line, user input is accepted from the command terminal.

### Example

```
name = input("Enter your name: ")
print ("Hello {}. How are you?".format(name))
```

The program is run from the command prompt terminal as follows –



```
C:\Python311>python hello.py
Enter your name: Rajan
Hello Rajan. How are you?

C:\Python311>
```

## Passing Arguments at the Time of Execution

Very often, you may need to input the data to be used by the program in the command line itself and use it inside the program. An example of giving the data in the command line could be any DOS commands in Windows or Linux.

In Windows, you use the following DOS command to rename a file `hello.py` to `hi.py`.

```
C:\Python311>ren hello.py hi.py
```

In Linux you may use the `mv` command –

```
$ mv hello.py hi.py
```

Here `ren` or `mv` are the commands which need the old and new file names. Since they are put in line with the command, they are called command-line arguments.

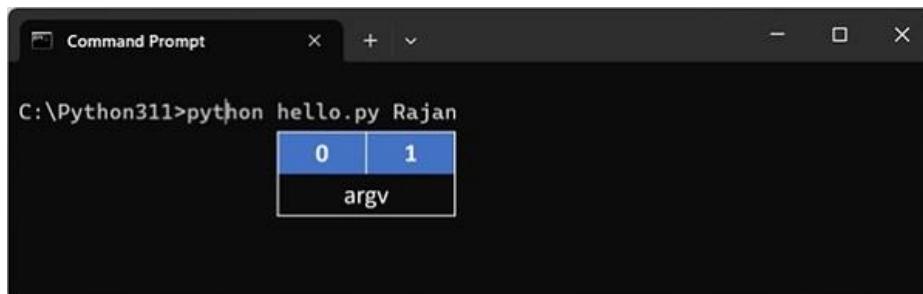
You can pass values to a Python program from command line. Python collects the arguments in a list object. Python's sys module provides access to any command-line arguments via the sys.argv variable. sys.argv is the list of command-line arguments and sys.argv[0] is the program i.e. the script name.

### Example

The hello.py script used input() function to accept user input after the script is run. Let us change it to accept input from command line.

```
import sys
print ('argument list', sys.argv)
name = sys.argv[1]
print ("Hello {}. How are you?".format(name))
```

Run the program from command-line as shown in the following figure –



The output is shown below –

```
C:\Python311>python hello.py Rajan
argument list ['hello.py', 'Rajan']
Hello Rajan. How are you?
```

The command-line arguments are always stored in string variables. To use them as numerics, you can convert them suitably with type conversion functions.

### Example

In the following example, two numbers are entered as command-line arguments. Inside the program, we use int() function to parse them as integer variables.

```
import sys
print ('argument list', sys.argv)
first = int(sys.argv[1])
second = int(sys.argv[2])
print ("sum = {}".format(first+second))
```

It will produce the following output –

```
C:\Python311>python hello.py 10 20
argument list ['hello.py', '10', '20']
sum = 30
```

Python's standard library includes a couple of useful modules to parse command line arguments and options –

- **getopt** – C-style parser for command line options.
- **argparse** – Parser for command-line options, arguments and sub-commands.

## Python getopt Module

Python provides a getopt module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

### getopt.getopt() method

This method parses the command line options and parameter list. Following is a simple syntax for this method –

```
getopt.getopt(args, options, [long_options])
```

Here is the detail of the parameters –

- **args** – This is the argument list to be parsed.
- **options** – This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long\_options** – This is an optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.

This method returns a value consisting of two elements – the first is a list of (option, value) pairs, the second is a list of program arguments left after the option list was stripped.

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

### Exception getopt.GetoptError

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes msg and opt give the error message and related option.

### Example

Suppose we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows –

```
usage: test.py -i <inputfile> -o <outputfile>
```

Here is the following script to test.py –

```
import sys, getopt
def main(argv):
    inputfile = ''
    outputfile = ''
```

```

try:
    opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile"])
except getopt.GetoptError:
    print ('test.py -i <inputfile> -o <outputfile>')
    sys.exit(2)
for opt, arg in opts:
    if opt == '-h':
        print ('test.py -i <inputfile> -o <outputfile>')
        sys.exit()
    elif opt in ("-i", "--ifile"):
        inputfile = arg
    elif opt in ("-o", "--ofile"):
        outputfile = arg
print ('Input file is "', inputfile)
print ('Output file is "', outputfile)
if __name__ == "__main__":
    main(sys.argv[1:])

```

Now, run the above script as follows –

```

$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i inputfile -o outputfile
Input file is " inputfile
Output file is " outputfile

```

## Python argparse Module

The argparse module provides tools for writing very easy to use command line interfaces. It handles how to parse the arguments collected in sys.argv list, automatically generate help and issues error message when invalid options are given.

First step to design the command line interface is to set up parser object. This is done by ArgumentParser() function in argparse module. The function can be given an explanatory string as description parameter.

Our script will be executed from command line without any arguments. If we use parse\_args() method of parser object, it does nothing because there aren't any arguments given.

```
import argparse
parser=argparse.ArgumentParser(description="sample argument parser")
args=parser.parse_args()
```

When the above script is run –

```
C:\Python311>python parser1.py
C:\Python311>python parser1.py -h
usage: parser1.py [-h]
sample argument parser
options:
-h, --help show this help message and exit
```

The second command line usage gives –help option which produces a help message as shown. The –help parameter is available by default.

Now let us define an argument which is mandatory for the script to run and if not given script should throw error. Here we define argument 'user' by add\_argument() method.

```
import argparse
parser=argparse.ArgumentParser(description="sample argument parser")
parser.add_argument("user")
args=parser.parse_args()
if args.user=="Admin":
    print ("Hello Admin")
else:
    print ("Hello Guest")
```

This script's help now shows one positional argument in the form of 'user'. The program checks if it's value is 'Admin' or not and prints corresponding message.

```
C:\Python311>python parser2.py --help
usage: parser2.py [-h] user
sample argument parser
positional arguments:
user
options:
-h, --help show this help message and exit
```

Use the following command –

```
C:\Python311>python parser2.py Admin
Hello Admin
```

But the following usage displays Hello Guest message.

```
C:\Python311>python parser2.py Rajan
Hello Guest
```

### The add\_argument() method

We can assign default value to an argument in add\_argument() method.

```
import argparse

parser=argparse.ArgumentParser(description="sample argument parser")
parser.add_argument("user", nargs='?', default="Admin")
args=parser.parse_args()

if args.user=="Admin":
    print ("Hello Admin")
else:
    print ("Hello Guest")
```

Here nargs is the number of command-line arguments that should be consumed. '?'. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced.

By default, all arguments are treated as strings. To explicitly mention type of argument, use type parameter in the add\_argument() method. All Python data types are valid values of type.

```
import argparse

parser=argparse.ArgumentParser(description="add numbers")
parser.add_argument("first", type=int)
parser.add_argument("second", type=int)
args=parser.parse_args()
x=args.first
y=args.second
z=x+y
print ('addition of {} and {} = {}'.format(x,y,z))
```

It will produce the following output –

```
C:\Python311>python parser3.py 10 20
addition of 10 and 20 = 30
```

In the above examples, the arguments are mandatory. To add optional argument, prefix its name by double dash --. In following case surname argument is optional because it is prefixed by double dash (--surname).

```
import argparse

parser=argparse.ArgumentParser()
parser.add_argument("name")
```

```
parser.add_argument("--surname")
args=parser.parse_args()
print ("My name is ", args.name, end=' ')
if args.surname:
    print (args.surname)
```

A one letter name of argument prefixed by single dash acts as a short name option.

```
C:\Python311>python parser3.py Anup
My name is Anup
C:\Python311>python parser3.py Anup --surname Gupta
My name is Anup Gupta
```

If it is desired that an argument should value only from a defined list, it is defined as choices parameter.

```
import argparse
parser=argparse.ArgumentParser()
parser.add_argument("sub", choices=['Physics', 'Maths', 'Biology'])
args=parser.parse_args()
print ("My subject is ", args.sub)
```

Note that if value of parameter is not from the list, invalid choice error is displayed.

```
C:\Python311>python parser3.py Physics
My subject is Physics
C:\Python311>python parser3.py History
usage: parser3.py [-h] {Physics,Maths,Biology}
parser3.py: error: argument sub: invalid choice: 'History' (choose from
'Physics', 'Maths', 'Biology')
```

# 201. Python - Docstrings

## Docstrings in Python

In Python, docstrings are a way of documenting modules, classes, functions, and methods. They are written within triple quotes ("""" """) and can span multiple lines.

Docstrings serve as convenient way of associating documentation with Python code. They are accessible through the `__doc__` attribute of the respective Python objects they document. Below are the different ways to write docstrings –

### Single-Line Docstrings

Single-line docstrings are used for brief and simple documentation. They provide a concise description of what the function or method does. Single-line docstrings should fit on one line within triple quotes and end with a period.

#### Example

In the following example, we are using a single line docstring to write a text –

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b

result = add(5, 3)
print("Sum:", result)
```

### Multi-Line Docstrings

Multi-line docstrings are used for more detailed documentation. They provide a more comprehensive description, including the parameters, return values, and other relevant details. Multi-line docstrings start and end with triple quotes and include a summary line followed by a blank line and a more detailed description.

#### Example

The following example uses multi-line docstrings as an explanation of the code –

```
def multiply(a, b):
    """
    Multiply two numbers and return the result.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.
```

```

>Returns:
int or float: The result of multiplying a and b.
"""

return a * b

result = multiply(5, 3)
print("Product:", result)

```

## Docstrings for Modules

When writing docstrings for modules, place the docstring at the top of the module, right after any import statements. The module docstring provide an overview of the module's functionality and list its primary components, such as list of functions, classes, and exceptions provided by the module.

### Example

In this example, we demonstrate the use of docstrings for modules in Python –

```

import os

"""

This module provides Utility functions for file handling operations.

Functions:
- 'read_file(filepath)': Reads and returns the contents of the file.
- 'write_file(filepath, content)': Writes content to the specified file.

Classes:
- 'FileNotFoundException': Raised when a file is not found.

Example usage:

>>> import file_utils
>>> content = file_utils.read_file("example.txt")
>>> print(content)
'Hello, world!'
>>> file_utils.write_file("output.txt", "This is a test.")
"""

print("This is os module")

```

## Docstrings for Classes

Classes can have docstrings to describe their purpose and usage. Each method within the class can also have its own docstring. The class docstring should provide an overview of the class and its methods.

### Example

In the example below, we showcase the use of docstrings for classes in Python –

```
class Calculator:  
    """  
    A simple calculator class to perform basic arithmetic operations.  
  
    Methods:  
    - add(a, b): Return the sum of two numbers.  
    - multiply(a, b): Return the product of two numbers.  
    """  
  
    def add(self, a, b):  
        """Return the sum of two numbers."""  
        return a + b  
  
    def multiply(self, a, b):  
        """  
        Multiply two numbers and return the result.  
  
        Parameters:  
        a (int or float): The first number.  
        b (int or float): The second number.  
  
        Returns:  
        int or float: The result of multiplying a and b.  
        """  
        return a * b  
  
cal = Calculator()  
print(cal.add(87, 98))  
print(cal.multiply(87, 98))
```

## Accessing Docstrings

Docstrings in Python are accessed using the `__doc__` attribute of the object they document. This attribute contains the documentation string (docstring) associated with the object, providing a way to access and display information about the purpose and usage of functions, classes, modules, or methods.

### Example

In the following example, we are defining two functions, "add" and "multiply", each with a docstring describing their parameters and return values. We then use the "`__doc__`" attribute to access and print these docstrings –

```
# Define a function with a docstring

def add(a, b):
    """
    Adds two numbers together.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of a and b.
    """

    return a + b

result = add(5, 3)
print("Sum:", result)

# Define another function with a docstring

def multiply(x, y):
    """
    Multiplies two numbers together.

    Parameters:
    x (int): The first number.
    y (int): The second number.

    Returns:
    int: The product of x and y.
    """


```

```

    return x * y
result = multiply(4, 7)
print("Product:", result)

# Accessing the docstrings
print(add.__doc__)
print(multiply.__doc__)

```

## Best Practices for Writing Docstrings

Following are the best practices for writing docstrings in Python –

- **Be Clear and Concise** – Ensure the docstring clearly explains the purpose and usage of the code, avoiding unnecessary details.
- **Use Proper Grammar and Spelling** – Ensure the docstring is well-written with correct grammar and spelling.
- **Follow Conventions** – Use the standard conventions for formatting docstrings, such as the Google style, NumPy style, or Sphinx style.
- **Include Examples** – Provide examples where applicable to illustrate how to use the documented code.

## Google Style Docstring

Google style docstrings provide a structured way to document Python code using indentation and headings. They are designed to be readable and informative, following a specific format.

### Example

Following is an example of a function with a Google style docstring –

```

def divide(dividend, divisor):
    """
    Divide two numbers and return the result.

    Args:
        dividend (float): The number to be divided.
        divisor (float): The number to divide by.

    Returns:
        float: The result of the division.

    Raises:
        ValueError: If `divisor` is zero.
    """

```

```

if divisor == 0:
    raise ValueError("Cannot divide by zero")
return dividend / divisor

result = divide(4, 7)
print("Division:", result)

```

## NumPy/SciPy Style Docstring

NumPy/SciPy style docstrings are common in scientific computing. They include sections for parameters, returns, and examples.

### Example

Following is an example of a function with a NumPy/SciPy style docstring –

```

def fibonacci(n):
    """
    Compute the nth Fibonacci number.

    Parameters
    -----
    n : int
        The index of the Fibonacci number to compute.

    Returns
    -----
    int
        The nth Fibonacci number.

    Examples
    -----
    >>> fibonacci(0)
    0
    >>> fibonacci(5)
    5
    >>> fibonacci(10)
    55
    """
    if n == 0:

```

```

        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

result = fibonacci(4)
print("Result:", result)

```

## Sphinx Style Docstring

Sphinx style docstrings are compatible with the Sphinx documentation generator and use reStructuredText formatting.

The reStructuredText (reST) is a lightweight markup language used for creating structured text documents. The Sphinx documentation generator takes "reStructuredText" files as input and generates high-quality documentation in various formats, including HTML, PDF, ePub, and more.

### Example

Following is an example of a function with a Sphinx style docstring –

```

def divide(dividend, divisor):
    """
    Divide two numbers and return the result.

    Args:
        dividend (float): The number to be divided.
        divisor (float): The number to divide by.

    Returns:
        float: The result of the division.

    Raises:
        ValueError: If `divisor` is zero.
    """
    if divisor == 0:
        raise ValueError("Cannot divide by zero")
    return dividend / divisor

result = divide(76, 37)

```

```
print("Result:", result)
```

## Docstring vs Comment

Following are the differences highlighted between Python docstrings and comments, focusing on their purposes, formats, usages, and accessibility respectively –

| <b>Docstring</b>                                                                                           | <b>Comment</b>                                                                         |
|------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Used to document Python objects such as functions, classes, methods, modules, or packages.                 | Used to annotate code for human readers, provide context, or temporarily disable code. |
| Written within triple quotes ("""" """) or (''' ''') and placed immediately after the object's definition. | Start with the # symbol and are placed on the same line as the annotated code.         |
| Stored as an attribute of the object and accessible programmatically.                                      | Ignored by the Python interpreter during execution, purely for human understanding.    |
| Accessed using the __doc__ attribute of the object.                                                        | Not accessible programmatically; exists only in the source code.                       |

## 202. Python - JSON

### JSON in Python

JSON in Python is a popular data format used for data exchange between systems. The `json` module provides functions to work with JSON data, allowing you to serialize Python objects into JSON strings and deserialize JSON strings back into Python objects.

*JSON (JavaScript Object Notation) is a Lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is mainly used to transmit data between a server and web application as text.*

### JSON Serialization

JSON serialization is the process of converting a Python object into a JSON format. This is useful for saving data in a format that can be easily transmitted or stored, and later reconstructed back into its original form.

Python provides the `json` module to handle JSON serialization and deserialization. We can use the `json.dumps()` method for serialization in this module.

You can serialize the following Python object types into JSON strings –

- dict
- list
- tuple
- str
- int
- float
- bool
- None

### Example

Following a basic example of how to serialize a Python dictionary into a JSON string –

```
import json

# Python dictionary
data = {"name": "Alice", "age": 30, "city": "New York"}

# Serialize to JSON string
json_string = json.dumps(data)
print(json_string)
```

It will produce the following output –

```
{"name": "Alice", "age": 30, "city": "New York"}
```

## JSON Deserialization

JSON deserialization is the process of converting a JSON string back into a Python object. This is essential for reading and processing data that has been transmitted or stored in JSON format.

In Python, we can use `json.loads()` method to deserialize JSON data from a string, and `json.load()` method to deserialize JSON data from a file.

### Example: Deserialize JSON string to Python object

In the following example we are deserializing a JSON string into a Python dictionary using the `json.loads()` method –

```
import json

# JSON string
json_string = '{"name": "John", "age": 30, "is_student": false, "courses": ["Math", "Science"], "address": {"city": "New York", "state": "NY"}}'

# Deserialize JSON string to Python object
python_obj = json.loads(json_string)

print(python_obj)
```

Following is the output of the above code –

```
{'name': 'John', 'age': 30, 'is_student': False, 'courses': ['Math', 'Science'], 'address': {'city': 'New York', 'state': 'NY'}}
```

### Example: Deserialize JSON from File

Now, to read and deserialize JSON data from a file, we use the `json.load()` method –

```
import json

# Read and deserialize from file
with open("data.json", "r") as f:
    python_obj = json.load(f)

print(python_obj)
```

Output of the above code is as follows –

```
{'name': 'John', 'age': 30, 'is_student': False, 'courses': ['Math', 'Science'], 'address': {'city': 'New York', 'state': 'NY'}}
```

## Advanced JSON Handling

If your JSON data includes objects that need special handling (e.g., custom classes), you can define custom deserialization functions. Use the `object_hook` parameter of `json.loads()` or `json.load()` method to specify a function that will be called with the result of every JSON object decoded.

### Example

In the example below, we are demonstrating the usage of custom object serialization –

```
import json
from datetime import datetime

# Custom deserialization function
def custom_deserializer(dct):
    if 'joined' in dct:
        dct['joined'] = datetime.fromisoformat(dct['joined'])
    return dct

# JSON string with datetime
json_string = '{"name": "John", "joined": "2021-05-17T10:15:00"}'

# Deserialize with custom function
python_obj = json.loads(json_string, object_hook=custom_deserializer)

print(python_obj)
```

We get the output as shown below –

```
{"name": 'John', 'joined': datetime.datetime(2021, 5, 17, 10, 15)}
```

## JSONEncoder Class

The `JSONEncoder` class in Python is used to encode Python data structures into JSON format. Each Python data type is converted into its corresponding JSON type, as shown in the following table –

| <b>Python</b>                          | <b>JSON</b> |
|----------------------------------------|-------------|
| Dict                                   | object      |
| list, tuple                            | array       |
| Str                                    | string      |
| int, float, int- & float-derived Enums | number      |
| TRUE                                   | TRUE        |
| FALSE                                  | FALSE       |

|      |      |
|------|------|
| None | null |
|------|------|

The JSONEncoder class is instantiated using the JSONEncoder() constructor. The following important methods are defined in this class –

- **encode(obj)** – Serializes a Python object into a JSON formatted string.
- **iterencode(obj)** – Encodes the object and returns an iterator that yields the encoded form of each item in the object.
- **indent** – Determines the indent level of the encoded string.
- **sort\_keys** – If True, the keys appear in sorted order.
- **check\_circular** – If True, checks for circular references in container-type objects.

### Example

In the following example, we are encoding Python list object. We use the iterencode() method to display each part of the encoded string –

```
import json

data = ['Rakesh', {'marks': (50, 60, 70)}]
e = json.JSONEncoder()

# Using iterencode() method
for obj in e.iterencode(data):
    print(obj)
```

It will produce the following output –

```
["Rakesh"
,
{
"marks"
:
[50
,
60
,
70
]
}
```

### JSONDecoder class

The JSONDecoder class is used to decode a JSON string back into a Python data structure. The main method in this class is decode().

### Example

In this example, the "JSONEncoder" is used to encode a Python list into a JSON string, and the "JSONDecoder" is then used to decode the JSON string back into a Python list –

```
import json

data = ['Rakesh', {'marks': (50, 60, 70)}]
e = json.JSONEncoder()
s = e.encode(data)
d = json.JSONDecoder()
obj = d.decode(s)
print(obj, type(obj))
```

The result obtained is as shown below –

```
['Rakesh', {'marks': [50, 60, 70]}] <class 'list'>
```

# 203. Python - Sending Email

## Sending Email in Python

You can send email in Python by using several libraries, but the most common ones are `smtplib` and `email`.

The "`smtplib`" module in Python defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. The `email` "package" is a library for managing email messages, including MIME and other RFC 2822-based message documents.

*An application that handles and delivers e-mail over the Internet is called a "mail server". Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending an e-mail and routing it between mail servers. It is an Internet standard for email transmission.*

## Python `smtplib.SMTP()` Function

The Python `smtplib.SMTP()` function is used to create an SMTP client session object, which establishes a connection to an SMTP server. This connection allows you to send emails through the server.

### Setting Up an SMTP Server

Before sending an email, you need to set up an SMTP server. Common SMTP servers include those provided by Gmail, Yahoo, or other mail service providers.

### Creating an SMTP Object

To send an email, you need to obtain the object of `SMTP` class with the following function –

```
import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

Here is the detail of the parameters –

- **host** – This is the host running your SMTP server. You can specify IP address of the host or a domain name like `tutorialspoint.com`. This is an optional argument.
- **port** – If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
- **local\_hostname** – If your SMTP server is running on your local machine, then you can specify just `localhost` as the option.

### Example

The following script connects to the SMTP server at "smtp.example.com" on port 25, optionally identifies and secures the connection, logs in (if required), sends an email, and then quits the session –

```
import smtplib

# Create an SMTP object and establish a connection to the SMTP server
smtpObj = smtplib.SMTP('smtp.example.com', 25)

# Identify yourself to an ESMTP server using EHLO
smtpObj.ehlo()

# Secure the SMTP connection
smtpObj.starttls()

# Login to the server (if required)
smtpObj.login('username', 'password')

# Send an email
from_address = 'your_email@example.com'
to_address = 'recipient@example.com'
message = """\
Subject: Test Email

This is a test email message.

"""

smtpObj.sendmail(from_address, to_address, message)

# Quit the SMTP session
smtpObj.quit()
```

## The Python smtplib Module

The Python `smtplib` module is used to create and manage a simple Mail Transfer Protocol (SMTP) server. This module allows you to set up an SMTP server that can receive and process incoming email messages, making it valuable for testing and debugging email functionalities within applications.

## Setting up an SMTP Debugging Server

The `smtplib` module comes pre-installed with Python and includes a local SMTP debugging server. This server is useful for testing email functionality without actually sending emails to specified addresses; instead, it prints the email content to the console.

Running this local server eliminates the need to handle message encryption or use credentials to log in to an email server.

## Starting the SMTP Debugging Server

You can start the local SMTP debugging server using the following command in Command Prompt or terminal –

```
python -m smtplib -c DebuggingServer -n localhost:1025
```

### Example

The following example demonstrates how to send a dummy email using the `smtplib` functionality along with the local SMTP debugging server –

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg += line

print("Message length is", len(msg))
server = smtplib.SMTP('localhost', 1025)
```

```
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

In this example –

- **From** – You input the sender's email address (fromaddr).
- **To** – You input the recipient's email address (toaddrs), which can be multiple addresses separated by spaces.
- **Message** – You input the message content, terminated by ^D (Unix) or ^Z (Windows).

The sendmail() method of "smtplib" sends the email using the specified sender, recipient(s), and message content to the local SMTP debugging server running on "localhost" at port "1025".

### Output

When you run the program, the console establishes the communication between the program and the SMTP server. Meanwhile, the terminal running the SMTPD server displays the incoming message content, helping you debug and verify the email sending process.

```
python example.py
From: abc@xyz.com
To: xyz@abc.com
Enter message, end with ^D (Unix) or ^Z (Windows):
Hello World
^Z
```

The console reflects the following log –

```
From: abc@xyz.com
reply: retcode (250); Msg: b'OK'
send: 'rcpt T0:<xyz@abc.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'From: abc@xyz.com\r\nTo: xyz@abc.com\r\n\r\nHello
World\r\n.\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
data: (250, b'OK')
send: 'quit\r\n'
```

```
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'
```

The terminal in which the SMTPD server is running shows this output –

```
----- MESSAGE FOLLOWS -----
b'From: abc@xyz.com'
b'To: xyz@abc.com'
b'X-Peer: ::1'
b''
b'Hello World'
----- END MESSAGE -----
```

## Sending an HTML e-mail using Python

To send an HTML email using Python, you can use the `smtplib` library to connect to an SMTP server and the `email.mime` modules to construct and format your email content appropriately.

### Constructing the HTML Email Message

When sending an HTML email, you need to specify certain headers and structure the message content accordingly to ensure it is recognized and rendered as HTML by the recipient's email client.

#### Example

Following is the example to send HTML content as an e-mail in Python –

```
import smtplib
from email.mime.multipart import MIME Multipart
from email.mime.text import MIMEText

# Email content
sender = 'from@fromdomain.com'
receivers = ['to@todomain.com']

# Create message container - the correct MIME type is multipart/alternative.
msg = MIME Multipart('alternative')
msg['From'] = 'From Person <from@fromdomain.com>'
msg['To'] = 'To Person <to@todomain.com>'
msg['Subject'] = 'SMTP HTML e-mail test'

# HTML message content
```

```

html = """\
<html>
  <head></head>
  <body>
    <p>This is an e-mail message to be sent in <b>HTML format</b></p>
    <p><b>This is HTML message.</b></p>
    <h1>This is headline.</h1>
  </body>
</html>
"""

# Attach HTML content to the email
part2 = MIMEText(html, 'html')
msg.attach(part2)

# Connect to SMTP server and send email
try:
    smtpObj = smtplib.SMTP('localhost')
    smtpObj.sendmail(sender, receivers, msg.as_string())
    print("Successfully sent email")
except smtplib.SMTPException as e:
    print(f"Error: unable to send email. Error message: {str(e)}")

```

## Sending Attachments as an E-mail

To send email attachments in Python, you can use the `smtplib` library to connect to an SMTP server and the `email.mime` modules to construct and format your email content, including attachments.

## Constructing an Email with Attachments

When sending an email with attachments, you need to format the email correctly using MIME (Multipurpose Internet Mail Extensions). This involves setting the Content-Type header to multipart/mixed to denote that the email contains both text and attachments. Each part of the email (text and attachments) is separated by boundaries.

### Example

Following is the example, which sends an email with a file `/tmp/test.txt` as an attachment

```

import smtplib
import base64

```

```
filename = "/tmp/test.txt"

# Read a file and encode it into base64 format
fo = open(filename, "rb")
filecontent = fo.read()
encodedcontent = base64.b64encode(filecontent) # base64

sender = 'webmaster@tutorialpoint.com'
reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"

body = """
This is a test email to send an attachment.
"""

# Define the main headers.
part1 = """From: From Person <me@fromdomain.net>
To: To Person <amrood.admin@gmail.com>
Subject: Sending Attachment
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=%s
--%s
""" % (marker, marker)

# Define the message action
part2 = """Content-Type: text/plain
Content-Transfer-Encoding:8bit

%s
--%s
""" % (body,marker)

# Define the attachment section
part3 = """Content-Type: multipart/mixed; name=\"%s\"
Content-Transfer-Encoding:base64
```

```

Content-Disposition: attachment; filename=%s

%s
--%s--
""" %(filename, filename, encodedcontent, marker)
message = part1 + part2 + part3

try:
    smtpObj = smtplib.SMTP('localhost')
    smtpObj.sendmail(sender, receiver, message)
    print "Successfully sent email"
except Exception:
    print "Error: unable to send email"

Ezoic

```

## Sending Email Using Gmail's SMTP Server

To send an email using Gmail's SMTP server in Python, you need to set up a connection to `smtp.gmail.com` on port "587" with "TLS" encryption, authenticate using your Gmail credentials, construct the email message using Python's `smtplib` and `email.mime` libraries, and send it using the `sendmail()` method. Finally, close the SMTP connection with `quit()`.

### Example

Following is an example script that demonstrates how to send an email using Gmail's SMTP server –

```

import smtplib

# Email content
content = "Hello World"

# Set up SMTP connection to Gmail's SMTP server
mail = smtplib.SMTP('smtp.gmail.com', 587)
# Identify yourself to the SMTP server
mail.ehlo()
# Start TLS encryption for the connection
mail.starttls()

# Gmail account credentials
sender = 'your_email@gmail.com'

```

```

password = 'your_password'

# Login to Gmail's SMTP server
mail.login(sender, password)

# Email details
recipient = 'recipient_email@example.com'
subject = 'Test Email'

# Construct email message with headers
header = f'To: {recipient}\nFrom: {sender}\nSubject: {subject}\n'
content = header + content

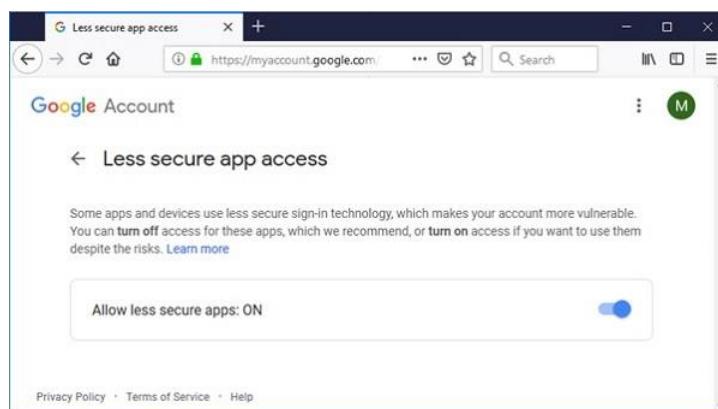
# Send email
mail.sendmail(sender, recipient, content)

# Close SMTP connection
mail.quit()

```

Before running above script, sender's gmail account must be configured to allow access for 'less secure apps'. Visit following link.

<https://myaccount.google.com/lesssecureapps> Set the shown toggle button to ON.



If everything goes well, execute the above script. The message should be delivered to the recipient's inbox.

# 204. Python - Further Extensions

Any code that you write using any compiled language like C, C++, or Java can be integrated or imported into another Python script. This code is considered as an "extension."

A Python extension module is nothing more than a normal C library. On Unix machines, these libraries usually end in .so (for shared object). On Windows machines, you typically see .dll (for dynamically linked library).

## Pre-Requisites for Writing Extensions

To start writing your extension, you are going to need the Python header files.

- On Unix machines, this usually requires installing a developer-specific package.
- Windows users get these headers as part of the package when they use the binary Python installer.

Additionally, it is assumed that you have a good knowledge of C or C++ to write any Python Extension using C programming.

## First look at a Python Extension

For your first look at a Python extension module, you need to group your code into four parts –

- The header file Python.h.
- The C functions you want to expose as the interface from your module..
- A table mapping the names of your functions as Python developers see them as C functions inside the extension module..
- An initialization function.

### The Header File Python.h

You need to include Python.h header file in your C source file, which gives you the access to the internal Python API used to hook your module into the interpreter.

Make sure to include Python.h before any other headers you might need. You need to follow the includes with the functions you want to call from Python.

### The C Functions

The signatures of the C implementation of your functions always takes one of the following three forms –

```
static PyObject *MyFunction(PyObject *self, PyObject *args);
static PyObject *MyFunctionWithKeywords(PyObject *self,
                                         PyObject *args,
                                         PyObject *kw);
static PyObject *MyFunctionWithNoArgs(PyObject *self);
```

Each one of the preceding declarations returns a Python object. There is no such thing as a void function in Python as there is in C. If you do not want your functions to return a value, return the C equivalent of Python's None value. The Python headers define a macro, Py\_RETURN\_NONE, that does this for us.

The names of your C functions can be whatever you like as they are never seen outside of the extension module. They are defined as static function.

Your C functions usually are named by combining the Python module and function names together, as shown here –

```
static PyObject *module_func(PyObject *self, PyObject *args) {
    /* Do your stuff here. */
    Py_RETURN_NONE;
}
```

This is a Python function called func inside the module called module. You will be putting pointers to your C functions into the method table for the module that usually comes next in your source code.

## The Method Mapping Table

This method table is a simple array of PyMethodDef structures. That structure looks something like this –

```
struct PyMethodDef {
    char *ml_name;
    PyCFunction ml_meth;
    int ml_flags;
    char *ml_doc;
};
```

Here is the description of the members of this structure –

- **ml\_name** – This is the name of the function as the Python interpreter presents when it is used in Python programs.
- **ml\_meth** – This is the address of a function that has any one of the signatures, described in the previous section.
- **ml\_flags** – This tells the interpreter which of the three signatures ml\_meth is using.
  - This flag usually has a value of METH\_VARARGS.
  - This flag can be bitwise OR'ed with METH\_KEYWORDS if you want to allow keyword arguments into your function.
  - This can also have a value of METH\_NOARGS that indicates you do not want to accept any arguments.
- **ml\_doc** – This is the docstring for the function, which could be NULL if you do not feel like writing one.

This table needs to be terminated with a sentinel that consists of NULL and 0 values for the appropriate members.

### Example

For the above-defined function, we have the following method mapping table –

```
static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { NULL, NULL, 0, NULL }
};
```

### The Initialization Function

The last part of your extension module is the initialization function. This function is called by the Python interpreter when the module is loaded. It is required that the function be named `initModule`, where `Module` is the name of the module.

The initialization function needs to be exported from the library you will be building. The Python headers define `PyMODINIT_FUNC` to include the appropriate incantations for that to happen for the particular environment in which we are compiling. All you have to do is use it when defining the function.

Your C initialization function generally has the following overall structure –

```
PyMODINIT_FUNC initModule() {
    Py_Initialize3(func, module_methods, "docstring...");
}
```

Here is the description of `Py_Initialize3` function –

- **func** – This is the function to be exported.
- **module\_methods** – This is the mapping table name defined above.
- **docstring** – This is the comment you want to give in your extension.

Putting all this together, it looks like the following –

```
#include <Python.h>

static PyObject *module_func(PyObject *self, PyObject *args) {
    /* Do your stuff here. */
    Py_RETURN_NONE;
}

static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC initModule() {
    Py_Initialize3(func, module_methods, "docstring...");
}
```

### Example

A simple example that makes use of all the above concepts –

```
#include <Python.h>

static PyObject* helloworld(PyObject* self)
{
    return Py_BuildValue("s", "Hello, Python extensions!!");
}

static char helloworld_docs[] =
    "helloworld( ): Any message you want to put here!!\n";
static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld,
     METH_NOARGS, helloworld_docs},
    {NULL}
};
void initelloworld(void)
{
    Py_InitModule3("helloworld", helloworld_funcs,
                  "Extension module example!");
}
```

Here the `Py_BuildValue` function is used to build a Python value. Save above code in `hello.c` file. We would see how to compile and install this module to be called from Python script.

## Building and Installing Extensions

The `distutils` package makes it very easy to distribute Python modules, both pure Python and extension modules, in a standard way. Modules are distributed in the source form, built and installed via a setup script usually called `setup.py`.

For the above module, you need to prepare the following `setup.py` script –

```
from distutils.core import setup, Extension
setup(name='helloworld', version='1.0', \
      ext_modules=[Extension('helloworld', ['hello.c'])])
```

Now, use the following command, which would perform all needed compilation and linking steps, with the right compiler and linker commands and flags, and copies the resulting dynamic library into an appropriate directory –

```
$ python setup.py install
```

On Unix-based systems, you will most likely need to run this command as root in order to have permissions to write to the site-packages directory. This usually is not a problem on Windows.

## Importing Extensions

Once you install your extensions, you would be able to import and call that extension in your Python script as follows –

```
import helloworld
print helloworld.helloworld()
```

This would produce the following output –

```
Hello, Python extensions!!
```

## Passing Function Parameters

As you will most likely want to define functions that accept arguments, you can use one of the other signatures for your C functions. For example, the following function, that accepts some number of parameters, would be defined like this –

```
static PyObject *module_func(PyObject *self, PyObject *args) {
    /* Parse args and do something interesting here. */
    Py_RETURN_NONE;
}
```

The method table containing an entry for the new function would look like this –

```
static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { "func", module_func, METH_VARARGS, NULL },
    { NULL, NULL, 0, NULL }
};
```

You can use the API `PyArg_ParseTuple` function to extract the arguments from the one `PyObject` pointer passed into your C function.

The first argument to `PyArg_ParseTuple` is the `args` argument. This is the object you will be parsing. The second argument is a format string describing the arguments as you expect them to appear. Each argument is represented by one or more characters in the format string as follows.

```
static PyObject *module_func(PyObject *self, PyObject *args) {
    int i;
    double d;
    char *s;
    if (!PyArg_ParseTuple(args, "ids", &i, &d, &s)) {
        return NULL;
    }

    /* Do something interesting here. */
```

```
Py_RETURN_NONE;
}
```

Compiling the new version of your module and importing it enables you to invoke the new function with any number of arguments of any type –

```
module.func(1, s="three", d=2.0)
module.func(i=1, d=2.0, s="three")
module.func(s="three", d=2.0, i=1)
```

You can probably come up with even more variations.

## The PyArg\_ParseTuple Function

Python re is the standard signature for the PyArg\_ParseTuple function –

```
int PyArg_ParseTuple(PyObject* tuple, char* format, ...)
```

This function returns 0 for errors, and a value not equal to 0 for success. Tuple is the PyObject\* that was the C function's second argument. Here format is a C string that describes mandatory and optional arguments.

Here is a list of format codes for the PyArg\_ParseTuple function –

| Code | C type    | Meaning                                                  |
|------|-----------|----------------------------------------------------------|
| c    | char      | A Python string of length 1 becomes a C char.            |
| d    | double    | A Python float becomes a C double.                       |
| f    | float     | A Python float becomes a C float.                        |
| i    | int       | A Python int becomes a C int.                            |
| l    | long      | A Python int becomes a C long.                           |
| L    | long long | A Python int becomes a C long long.                      |
| O    | PyObject* | Gets non-NULL borrowed reference to Python argument.     |
| S    | char*     | Python string without embedded nulls to C char*.         |
| s#   | char*+int | Any Python string to C address and length.               |
| t#   | char*+int | Read-only single-segment buffer to C address and length. |

|       |                 |                                                           |
|-------|-----------------|-----------------------------------------------------------|
| u     | Py_UNICODE*     | Python Unicode without embedded nulls to C.               |
| u#    | Py_UNICODE*+int | Any Python Unicode C address and length.                  |
| w#    | char*+int       | Read/write single-segment buffer to C address and length. |
| z     | char*           | Like s, also accepts None (sets C char* to NULL).         |
| z#    | char*+int       | Like s#, also accepts None (sets C char* to NULL).        |
| (...) | as per ...      | A Python sequence is treated as one argument per item.    |
|       |                 | The following arguments are optional.                     |
| :     |                 | Format end, followed by function name for error messages. |
| ;     |                 | Format end, followed by entire error message text.        |

## Returning Values

Py\_BuildValue takes in a format string much like PyArg\_ParseTuple does. Instead of passing in the addresses of the values you are building, you pass in the actual values. Here is an example showing how to implement an add function.

```
static PyObject *foo_add(PyObject *self, PyObject *args) {
    int a;
    int b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("i", a + b);
}
```

This is what it would look like if implemented in Python –

```
def add(a, b):
    return (a + b)
```

You can return two values from your function as follows. This would be captured using a list in Python.

```
static PyObject *foo_add_subtract(PyObject *self, PyObject *args) {
    int a;
    int b;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("ii", a + b, a - b);
}
```

This is what it would look like if implemented in Python –

```
def add_subtract(a, b):
    return (a + b, a - b)
```

## The Py\_BuildValue Function

Here is the standard signature for Py\_BuildValue function –

```
PyObject* Py_BuildValue(char* format,...)
```

Here format is a C string that describes the Python object to build. The following arguments of Py\_BuildValue are C values from which the result is built. The PyObject\* result is a new reference.

The following table lists the commonly used code strings, of which zero or more are joined into a string format.

| <b>Code</b> | <b>C type</b> | <b>Meaning</b>                                   |
|-------------|---------------|--------------------------------------------------|
| c           | char          | A C char becomes a Python string of length 1.    |
| d           | double        | A C double becomes a Python float.               |
| f           | float         | A C float becomes a Python float.                |
| i           | int           | C int becomes a Python int                       |
| l           | long          | A C long becomes a Python int                    |
| N           | PyObject*     | Passes a Python object and steals a reference.   |
| O           | PyObject*     | Passes a Python object and INCREFs it as normal. |
| O&          | convert+void* | Arbitrary conversion                             |

|       |                 |                                                                      |
|-------|-----------------|----------------------------------------------------------------------|
| s     | char*           | C 0-terminated char* to Python string, or NULL to None.              |
| s#    | char*+int       | C char* and length to Python string, or NULL to None.                |
| u     | Py_UNICODE*     | C-wide, null-terminated string to Python Unicode, or NULL to None.   |
| u#    | Py_UNICODE*+int | C-wide string and length to Python Unicode, or NULL to None.         |
| w#    | char*+int       | Read/write single-segment buffer to C address and length.            |
| z     | char*           | Like s, also accepts None (sets C char* to NULL).                    |
| z#    | char*+int       | Like s#, also accepts None (sets C char* to NULL).                   |
| (...) | as per ...      | Builds Python tuple from C values.                                   |
| [...] | as per ...      | Builds Python list from C values.                                    |
| {...} | as per ...      | Builds Python dictionary from C values, alternating keys and values. |

Code {...} builds dictionaries from an even number of C values, alternately keys and values. For example, Py\_BuildValue("{issi}",23,"zig","zag",42) returns a dictionary like Python's {23:'zig','zag':42}

# 205. Python - Tools/Utilities

The standard library comes with a number of modules that can be used both as modules and as command-line utilities.

## The dis Module

The `dis` module is the Python disassembler. It converts byte codes to a format that is slightly more appropriate for human consumption.

You can run the disassembler from the command line. It compiles the given script and prints the disassembled byte codes to the STDOUT. You can also use `dis` as a module. The `dis` function takes a class, method, function or code object as its single argument.

### Example

```
import dis

def sum():
    vara = 10
    varb = 20

    sum = vara + varb
    print ("vara + varb = %d" % sum)

# Call dis function for the function.
dis.dis(sum)
```

This would produce the following result –

|   |                |           |
|---|----------------|-----------|
| 3 | 0 LOAD_CONST   | 1 (10)    |
|   | 2 STORE_FAST   | 0 (vara)  |
| 4 | 4 LOAD_CONST   | 2 (20)    |
|   | 6 STORE_FAST   | 1 (varb)  |
| 6 | 8 LOAD_FAST    | 0 (vara)  |
|   | 10 LOAD_FAST   | 1 (varb)  |
|   | 12 BINARY_ADD  |           |
|   | 14 STORE_FAST  | 2 (sum)   |
| 7 | 16 LOAD_GLOBAL | 0 (print) |

```

18 LOAD_CONST      3 ('vara + varb = %d')
20 LOAD_FAST       2 (sum)
22 BINARY_MODULO
24 CALL_FUNCTION   1
26 POP_TOP
28 LOAD_CONST      0 (None)
30 RETURN_VALUE

```

## The pdb Module

The pdb module is the standard Python debugger. It is based on the bdb debugger framework.

You can run the debugger from the command line (type n [or next] to go to the next line and help to get a list of available commands) –

### Example

Before you try to run pdb.py, set your path properly to Python lib directory. So let us try with above example sum.py –

```

$pdb.py sum.py
> /test/sum.py(3)<module>()
-> import dis
(Pdb) n
> /test/sum.py(5)<module>()
-> def sum():
(Pdb) n
>/test/sum.py(14)<module>()
-> dis.dis(sum)
(Pdb) n
6      0 LOAD_CONST      1 (10)
            3 STORE_FAST      0 (vara)

7      6 LOAD_CONST      2 (20)
            9 STORE_FAST      1 (varb)

9      12 LOAD_FAST       0 (vara)
            15 LOAD_FAST       1 (varb)
            18 BINARY_ADD
            19 STORE_FAST      2 (sum)

```

```

10      22 LOAD_CONST      3 ('vara + varb = %d')
        25 LOAD_FAST       2 (sum)
        28 BINARY_MODULO
        29 PRINT_ITEM
        30 PRINT_NEWLINE
        31 LOAD_CONST      0 (None)
        34 RETURN_VALUE

--Return--
> /test/sum.py(14)<module>()->None
-v dis.dis(sum)
(Pdb) n
--Return--
> <string>(1)<module>()->None
(Pdb)

```

## The profile Module

The profile module is the standard Python profiler. You can run the profiler from the command line –

### Example

Let us try to profile the following program –

```

vara = 10
varb = 20
sum = vara + varb
print "vara + varb = %d" % sum

```

Now, try running cProfile.py over this file sum.py as follow –

```

$cProfile.py sum.py
vara + varb = 30
    4 function calls in 0.000 CPU seconds

    Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno
  1      0.000    0.000    0.000    0.000 <string>:1(<module>)
  1      0.000    0.000    0.000    0.000 sum.py:3(<module>)
  1      0.000    0.000    0.000    0.000 {execfile}
  1      0.000    0.000    0.000    0.000 {method .....}

```

## The tabnanny Module

The tabnanny module checks Python source files for ambiguous indentation. If a file mixes tabs and spaces in a way that throws off indentation, no matter what tab size you're using, the nanny complains.

### Example

Let us try to profile the following program –

```
vara = 10
varb = 20

sum = vara + varb
print "vara + varb = %d" % sum
```

If you would try a correct file with tabnanny.py, then it won't complain as follows –

```
$tabnanny.py -v sum.py
'sum.py': Clean bill of health.
```

# 206. Python - GUIs

In this chapter, you will learn about some popular Python IDEs (Integrated Development Environment), and how to use IDE for program development.

To use the scripted mode of Python, you need to save the sequence of Python instructions in a text file and save it with .py extension. You can use any text editor available on the operating system. Whenever the interpreter encounters errors, the source code needs to be edited and run again and again. To avoid this tedious method, IDE is used. An IDE is a one stop solution for typing, editing the source code, detecting the errors and executing the program.

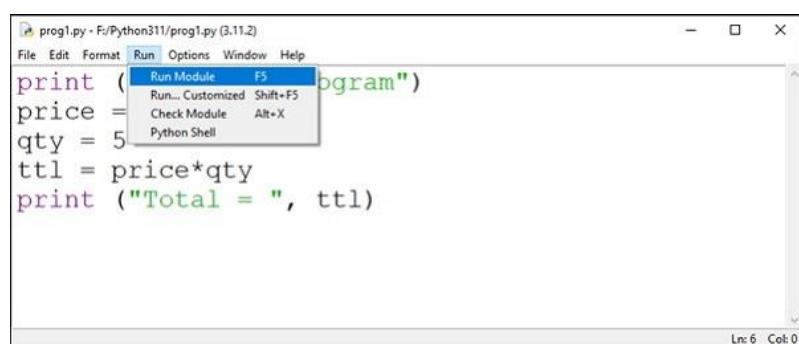
## IDLE

Python's standard library contains the IDLE module. IDLE stands for Integrated Development and Learning Environment. As the name suggests, it is useful when one is in the learning stage. It includes a Python interactive shell and a code editor, customized to the needs of Python language structure. Some of its important features include syntax highlighting, auto-completion, customizable interface etc.

To write a Python script, open a new text editor window from the File menu.



A new editor window opens in which you can enter the Python code. Save it and run it with Run menu.



## Jupyter Notebook

Initially developed as a web interface for IPython, Jupyter Notebook supports multiple languages. The name itself derives from the alphabets from the names of the supported languages – Julia, PYThon and R. Jupyter notebook is a client server application. The server is launched at the localhost, and the browser acts as its client.

Install Jupyter notebook with PIP –

```
pip3 install jupyter
```

Invoke from the command line.

```
C:\Users\Acer>jupyter notebook
```

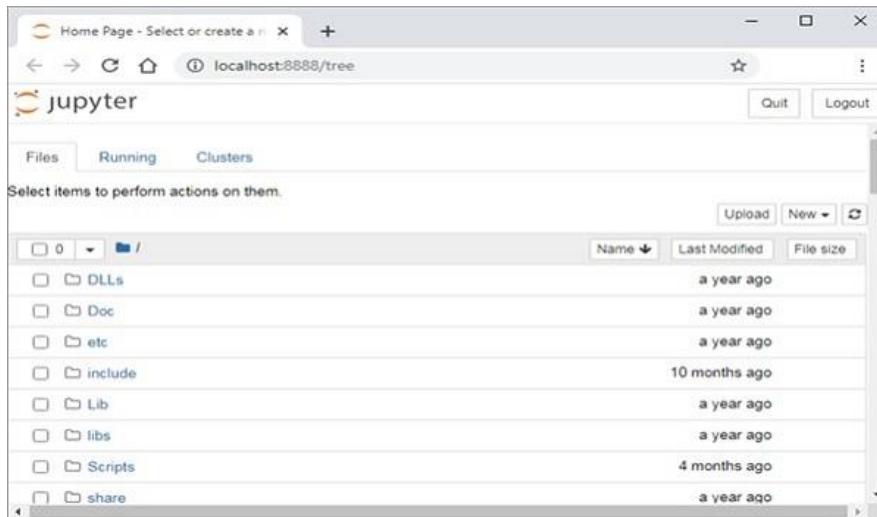
The server is launched at localhost's 8888 port number.

```
Microsoft Windows [Version 10.0.17134.165]
(c) 2018 Microsoft Corporation. All rights reserved.

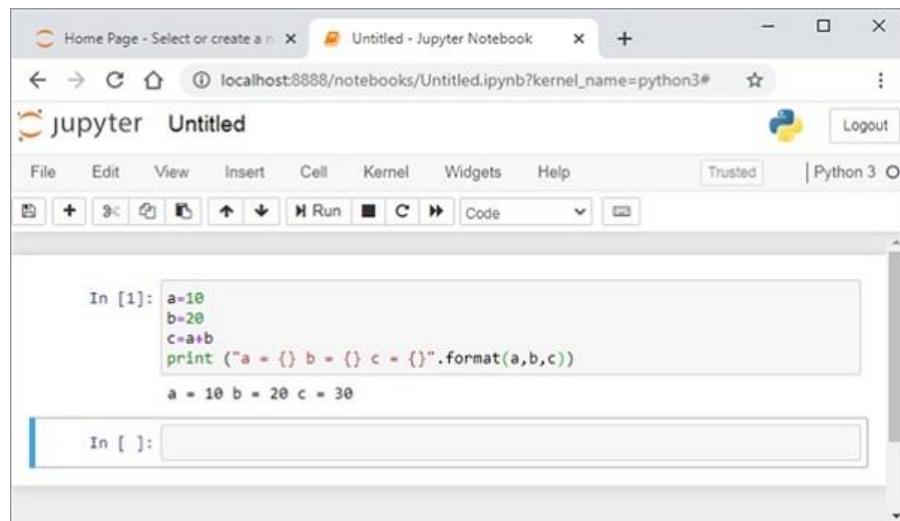
C:\Users\user>jupyter notebook
[I 00:55:22.641 NotebookApp] Serving notebooks from local directory: C:\Users\User
[I 00:55:22.657 NotebookApp] The Jupyter Notebook is running at:
[I 00:55:22.657 NotebookApp] http://localhost:8888/?token=elec02cf20ceeeaa5f876c4b02c498969213a104f658cd10e
[I 00:55:22.657 NotebookApp] or http://127.0.0.1:8888/?token=elec02cf20ceeeaa5f876c4b02c498969213a104f658cd10e
[I 00:55:22.657 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 00:55:23.079 NotebookApp]

To access the notebook, open this file in a browser:
  file:///C:/Users/User/AppData/Roaming/jupyter/runtime/nbserver-1032-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=elec02cf20ceeeaa5f876c4b02c498969213a104f658cd10e
  or http://127.0.0.1:8888/?token=elec02cf20ceeeaa5f876c4b02c498969213a104f658cd10e
```

The default browser of your system opens a link <http://localhost:8888/tree> to display the dashboard.



Open a new Python notebook. It shows IPython style input cell. Enter Python instructions and run the cell.



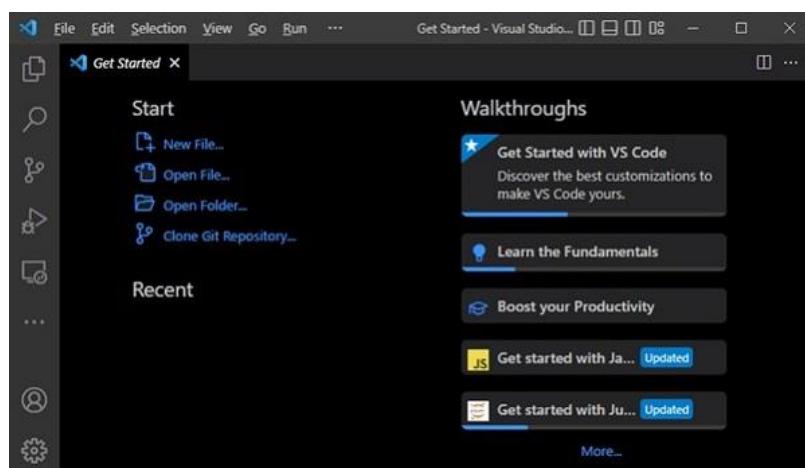
Jupyter notebook is a versatile tool, used very extensively by data scientists to display inline data visualizations. The notebook can be conveniently converted and distributed in PDF, HTML or Markdown format.

## VS Code

Microsoft has developed a source code editor called VS Code (Visual Studio Code) that supports multiple languages including C++, Java, Python and others. It provides features such as syntax highlighting, autocomplete, debugger and version control.

VS Code is a freeware. It is available for download and install from <https://code.visualstudio.com/>.

Launch VS Code from the start menu (in Windows).



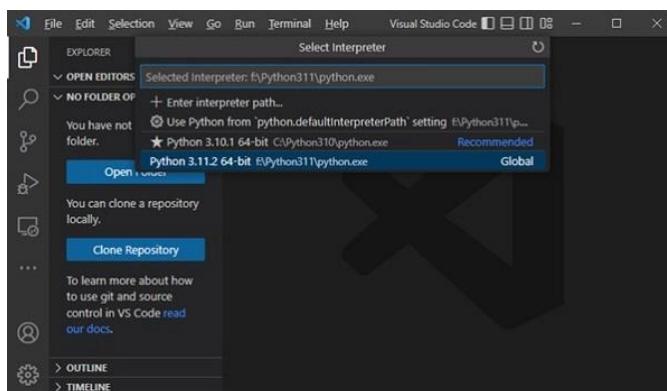
You can also launch VS Code from command line –

```
C:\test>code .
```

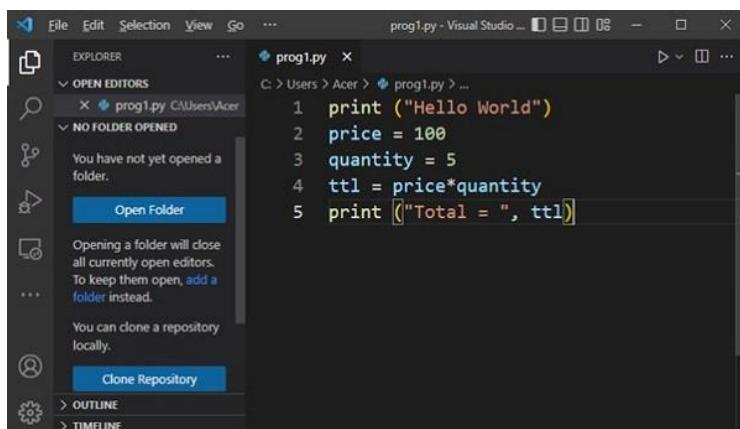
VS Code cannot be used unless respective language extension is not installed. VS Code Extensions marketplace has a number of extensions for language compilers and other utilities. Search for Python extension from the Extension tab (Ctrl+Shift+X) and install it.



After activating Python extension, you need to set the Python interpreter. Press **Ctrl+Shift+P** and select Python interpreter.



Open a new text file, enter Python code and save the file.



Open a command prompt terminal and run the program.

The screenshot shows the Visual Studio Code interface. The left sidebar has icons for Explorer, Open Editors, and No Folder Opened, with 'Open Folder' highlighted. Below these are sections for cloning a repository and viewing outline/timeline. The main area shows a Python file 'prog1.py' with the following code:

```
1 print ("Hello World")
2 price = 100
3 quantity = 5
4 ttl = price*quantity
5 print ("Total = ", ttl)
```

The terminal tab at the bottom shows the command 'cmd' and the output of running the script:

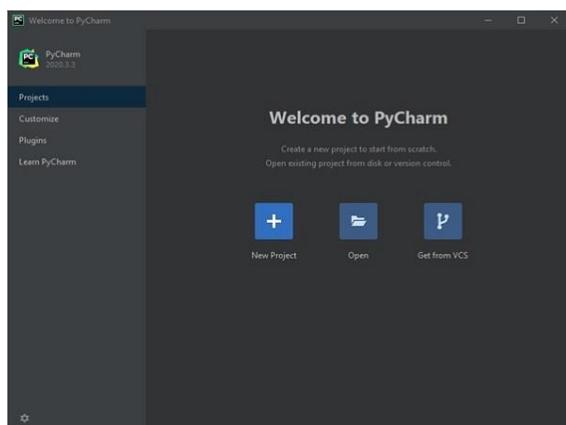
```
C:\Users\Acer>
C:\Users\Acer>python prog1.py
Hello World
Total = 500
```

# PyCharm

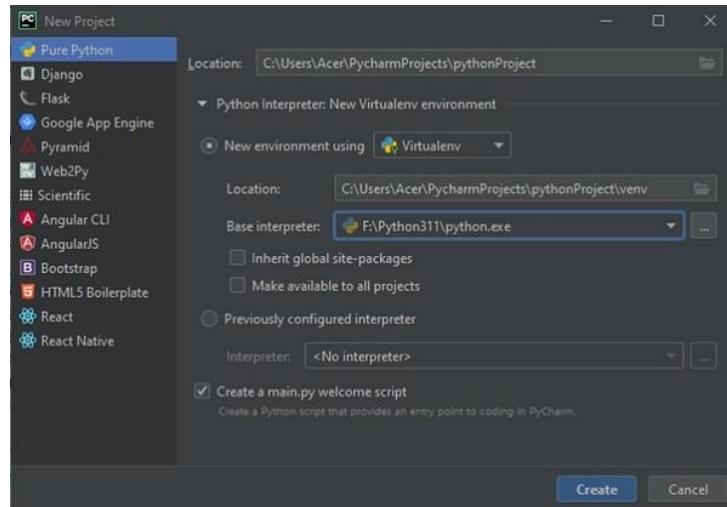
PyCharm is another popular Python IDE. It has been developed by JetBrains, a Czech software company. Its features include code analysis, a graphical debugger, integration with version control systems etc. PyCharm supports web development with Django.

The community as well as professional editions can be downloaded from <https://www.jetbrains.com/pycharm/download>.

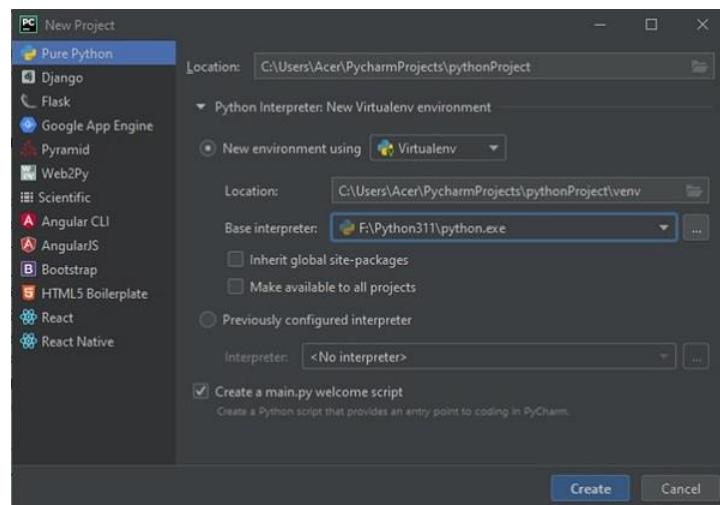
Download and install the latest Version: 2022.3.2 and open PyCharm. The Welcome screen appears as below –



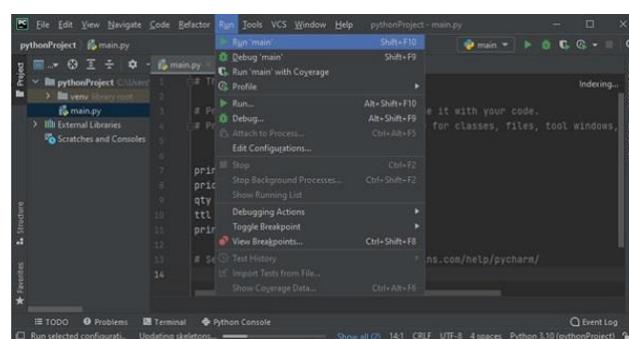
When you start a new project, PyCharm creates a virtual environment for it based on the choice of folder location and the version of Python interpreter chosen.



You can now add one or more Python scripts required for the project. Here we add a sample Python code in main.py file.



To execute the program, choose from Run menu or use Shift+F10 shortcut.



Output will be displayed in the console window as shown below –

```
pythonProject main.py
pythonProject /main.py
11     print ("Total = ", ttl)
12
13 # See PyCharm help at https://www.jetbrains.com/help/pycharm/
14

Indexing...
Database
Scratches and Consoles

Project
Structure
Run: main
Main
My first program
Total = 500
Process finished with exit code 0
Run Problems Terminal Python Console Event Log
PyCharm 2020.3.5 available // 0... (11 minutes ago) Updating skeletons... Show all 141 Python 3.10 (python>Project)
```

# Python Advanced Concepts

# 207. Python - Abstract Base Classes

An Abstract Base Class (ABC) in Python is a class that cannot be instantiated directly and is intended to be subclassed. ABCs serve as blueprints for other classes by providing a common interface that all subclasses must implement.

They are a fundamental part of object-oriented programming in Python which enables the developers to define and enforce a consistent API for a group of related classes.

## Purpose of Abstract Base Classes

Here's an in-depth look at the purpose and functionality of the Abstract Base Classes of Python –

### Defining a Standard Interface

Abstract Base Class (ABC) allows us to define a blueprint for other classes. This blueprint ensures that any class deriving from the Abstract Base Class (ABC) implements certain methods by providing a consistent interface.

Following is the example code of defining the standard Interface of the Abstract Base Class in Python –

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

### Enforcing Implementation

When a class inherits from an Abstract Base Class (ABC) it must implement all abstract methods. If it doesn't then Python will raise a `TypeError`. Here is the example of enforcing implementation of the Abstract Base Class in Python –

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```

def area(self):
    return self.width * self.height

def perimeter(self):
    return 2 * (self.width + self.height)

# This will work
rect = Rectangle(5, 10)

# This will raise TypeError
class IncompleteShape(Shape):
    pass

```

## Providing a Template for Future Development

Abstract Base Class (ABC) is useful in large projects where multiple developers might work on different parts of the codebase. They provide a clear template for developers to follow which ensure consistency and reducing errors.

## Facilitating Polymorphism

Abstract Base Class (ABC) make polymorphism possible by enabling the development of code that can operate with objects from diverse classes as long as they conform to a specific interface. This capability streamlines the extension and upkeep of code.

Below is the example of Facilitating Polymorphism in Abstract Base Class of Python –

```

def print_shape_info(shape: Shape):
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

square = Rectangle(4, 4)
print_shape_info(square)

```

**Note:** To execute the above mentioned example codes, it is necessary to define the standard interface and Enforcing Implementation.

## Components of Abstract Base Classes

Abstract Base Classes (ABCs) in Python consist of several key components that enable them to define and enforce interfaces for subclasses.

These components include the ABC class, the abstractmethod decorator and several others that help in creating and managing abstract base classes. Here are the key components of Abstract Base Classes –

- **ABC Class:** This class from Python's Abstract Base Classes (ABCs) module serves as the foundation for creating abstract base classes. Any class derived from ABC is considered an abstract base class.
- **'abstractmethod' Decorator:** This decorator from the abc module is used to declare methods as abstract. These methods do not have implementations in the ABC and must be overridden in derived classes.
- **'ABCMeta' Metaclass:** This is the metaclass used by ABC. It is responsible for keeping track of which methods are abstract and ensuring that instances of the abstract base class cannot be created if any abstract methods are not implemented.
- **Concrete Methods in ABCs:** Abstract base classes can also define concrete methods that provide a default implementation. These methods can be used or overridden by subclasses.
- **Instantiation Restrictions:** A key feature of ABCs is that they cannot be instantiated directly if they have any abstract methods. Attempting to instantiate an ABC with unimplemented abstract methods will raise a 'TypeError'.
- **Subclass Verification:** Abstract Base Classes (ABCs) can verify if a given class is a subclass using the `issubclass()` function and can check instances with the `isinstance()` function.

## Example of Abstract Base Classes in Python

Following example shows how ABCs enforce method implementation, support polymorphism and provide a clear and consistent interface for related classes –

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

    def description(self):
        return "I am a shape."

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
```

```
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        import math
        return math.pi * self.radius ** 2

    def perimeter(self):
        import math
        return 2 * math.pi * self.radius

def print_shape_info(shape):
    print(shape.description())
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

shapes = [Rectangle(5, 10), Circle(7)]

for shape in shapes:
    print_shape_info(shape)
    print("-" * 20)

class IncompleteShape(Shape):
    pass

try:
    incomplete_shape = IncompleteShape()
except TypeError as e:
    print(e)
```

## Output

On executing the above code we will get the following output –

```
I am a shape.  
Area: 50  
Perimeter: 30  
-----  
I am a shape.  
Area: 153.93804002589985  
Perimeter: 43.982297150257104  
-----  
Can't instantiate abstract class IncompleteShape with abstract methods area,  
perimeter
```

# 208. Python - Custom Exceptions

## What are Custom Exceptions in Python?

Python custom exceptions are user-defined error classes that extend the base Exception class. Developers can define and handle specific error conditions that are unique to their application. Developers can improve their code by creating custom exceptions. This allows for more meaningful error messages and facilitates the debugging process by indicating what kind of error occurred and where it originated.

To define a unique exception, we have to typically create a new class that takes its name from Python's built-in Exception class or one of its subclasses. A corresponding except block can be used to raise this custom class and catch it.

Developers can control the flow of the program when specific errors occur and take appropriate actions such as logging the error, retrying operations or gracefully shutting down the application. Custom exceptions can carry additional context or data about the error by overriding the `__init__` method and storing extra information as instance attributes.

Using custom exceptions improves the clarity of error handling in complex programs. It helps to distinguish between different types of errors that may require different handling strategies. For example, when a file parsing library might define exceptions like `FileFormatError`, `MissingFieldError` or `InvalidFieldError` to handle various issues that can arise during file processing. This level of granularity allows the client code to catch and address specific issues more effectively by improving the robustness and user experience of the software. Python's custom exceptions are a great tool for handling errors and writing better with more expressive code.

## Why to Use Custom Exceptions?

Custom exceptions in Python offer several advantages which enhances the functionality, readability and maintainability of our code. Here are the key reasons for using custom exceptions –

- **Specificity:** Custom exceptions allow us to represent specific error conditions that are unique to our application.
- **Clarity:** They make the code more understandable by clearly describing the nature of the errors.
- **Granularity:** Custom exceptions allow for more precise error handling.
- **Consistency:** They help to maintain a consistent error-handling strategy across the codebase.

## Creating Custom Exceptions

Creating custom exceptions in Python involves defining new exception classes that extend from Python's built-in Exception class or any of its subclasses. This allows developers to create specialized error types that cater to specific scenarios within their applications. Here's how we can create and use custom exceptions effectively –

## Define the Custom Exception Class

We can start creating the custom exceptions by defining a new class that inherits from `Exception` or another exception class such as `RuntimeError`, `ValueError`, etc. depending on the nature of the error.

Following is the example of defining the custom exception class. In this example `CustomError` is a custom exception class that inherits from `Exception`. The `__init__` method initializes the exception with an optional error message –

```
class CustomError(Exception):
    def __init__(self, message):
        super().__init__(message)
        self.message = message
```

## Raise the Custom Exception

To raise the custom exception, we can use the `raise` statement followed by an instance of our custom exception class. Optionally we can pass a message to provide context about the error.

In this function `process_data()` the `CustomError` exception is raised when the `data` parameter is empty by indicating a specific error condition.

```
def process_data(data):
    if not data:
        raise CustomError("Empty data provided")
    # Processing logic here
```

## Handle the Custom Exception

To handle the custom exception, we have to use a `try-except` block. Catch the custom exception class in the `except` block and handle the error as needed.

Here in the below code if `process_data([])` raises a `CustomError` then the `except` block catches it and we can print the error message (`e.message`) or perform other error-handling tasks.

```
try:
    process_data([])
except CustomError as e:
    print(f"Custom Error occurred: {e.message}")
    # Additional error handling logic
```

## Example of Custom Exception

Following is the basic example of custom exception handling in Python. In this example, we define a custom exception by subclassing the built-in `Exception` class and use a `try-except` block to handle the custom exception –

```
# Define a custom exception

class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Function that raises the custom exception

def check_value(value):
    if value < 0:
        raise CustomError("Value cannot be negative!")
    else:
        return f"Value is {value}"

# Using the function with exception handling

try:
    result = check_value(-5)
    print(result)
except CustomError as e:
    print(f"Caught an exception: {e.message}")
```

## Output

On executing the above code, we will get the following output –

```
Caught an exception: Value cannot be negative!
```

# 209. Python - Higher Order Functions

Higher-order functions in Python allows you to manipulate functions for increasing the flexibility and re-usability of your code. You can create higher-order functions using nested scopes or callable objects.

Additionally, the `functools` module provides utilities for working with higher-order functions, making it easier to create decorators and other function-manipulating constructs. This tutorial will explore the concept of higher-order functions in Python and demonstrate how to create them.

## What is a Higher-Order Function?

A higher-order function takes one or more functions as arguments or returns a function as its result. Below you can observe the some of the properties of the higher-order function in Python –

- A function can be stored in a variable.
- A function can be passed as a parameter to another function.
- A high order functions can be stored in the form of lists, hash tables, etc.
- Function can be returned from a function.

To create higher-order function in Python, you can use nested scopes or callable objects. Below we will discuss about them briefly.

## Creating Higher Order Function with Nested Scopes

One way to defining a higher-order function in Python is by using nested scopes. This involves defining a function within another function and returns the inner function.

### Example

Let's observe following example for creating a higher order function in Python. In this example, the `multiplier` function takes one argument, `a`, and returns another function `multiply`, which calculates the value `a * b`

```
def multiplier(a):  
    # Nested function with second number  
  
    def multiply(b):  
        # Multiplication of two numbers  
        return a * b  
  
    return multiply  
  
# Assigning nested multiply function to a variable  
multiply_second_number = multiplier(5)  
  
# Using variable as high order function  
Result = multiply_second_number(10)
```

```
# Printing result
print("Multiplication of Two numbers is: ", Result)
```

**Output**

On executing the above program, you will get the following results –

```
Multiplication of Two numbers is:  50
```

**Creating Higher-Order Functions with Callable Objects**

Another approach to create higher-order functions is by using callable objects. This involves defining a class with a `__call__` method.

**Example**

Here is the another approach to creating higher-order functions is using callable objects.

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, x):
        return self.factor * x

# Create an instance of the Multiplier class
multiply_second_number = Multiplier(2)

# Call the Multiplier object to computes factor * x
Result = multiply_second_number(100)

# Printing result
print("Multiplication of Two numbers is: ", Result)
```

**Output**

On executing the above program, you will get the following results –

```
Multiplication of Two numbers is:  200
```

**Higher-order functions with the 'functools' Module**

The `functools` module provides higher-order functions that act on or return other functions. Any callable object can be treated as a function for the purposes of this module.

**Working with Higher-order functions using the wraps()**

In this example, my\_decorator is a higher-order function that modifies the behavior of invite function using the functools.wraps() function.

```
import functools

def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print("Calling", f.__name__)
        return f(*args, **kwargs)
    return wrapper

@my_decorator
def invite(name):
    print(f"Welcome to, {name}!")

invite("Tutorialspoint")
```

## Output

On executing the above program, you will get the following results –

```
Calling invite
Welcome to, Tutorialspoint!
```

## Working with Higher-order functions using the partial()

The partial() function of the functools module is used to create a callable 'partial' object. This object itself behaves like a function. The partial() function receives another function as argument and freezes some portion of a function's arguments resulting in a new object with a simplified signature.

## Example

In following example, a user defined function myfunction() is used as argument to a partial function by setting default value on one of the arguments of original function.

```
import functools

def myfunction(a,b):
    return a*b

partfunction = functools.partial(myfunction, b = 10)
print(partfunction(10))
```

## Output

On executing the above program, you will get the following results –

100

### Working with Higher-order functions using the reduce()

Similar to the above approach the functools module provides the reduce() function, that receives two arguments, a function and an iterable. And, it returns a single value. The argument function is applied cumulatively two arguments in the list from left to right. Result of the function in first call becomes first argument and third item in list becomes second. This is repeated till list is exhausted.

#### Example

```
import functools

def mult(x,y):
    return x*y

# Define a number to calculate factorial
n = 4

num = functools.reduce(mult, range(1, n+1))

print (f'Factorial of {n}: ',num)
```

#### Output

On executing the above program, you will get the following results –

Factorial of 4: 24

# 210. Python - Object Internals

The internals of Python objects provide deeper insights into how Python manages and manipulates data. This knowledge is essential for writing efficient, optimized code and for effective debugging.

Whether we're handling immutable or mutable objects by managing memory with reference counting and garbage collection or leveraging special methods and slots, grasping these concepts is fundamental to mastering Python programming.

Understanding Python's object internals is crucial for optimizing code and debugging. Following is an overview of the key aspects of Python object internals –

## Object Structure

In Python, every object is a complex data structure that encapsulates various pieces of information. Understanding the object structure helps developers to grasp how Python manages memory and handles data.

Each python object mainly consists of two parts as mentioned below –

- **Object Header:** This is a crucial part of every Python object that contains essential information for the Python interpreter to manage the object effectively. It typically consists of two main components namely Reference count and Type Pointer.
- **Object Data:** This data is the actual data contained within the object which can differ based on the object's type. For example, an integer contains its numeric value while a list contains references to its elements.

## Object Identity

Object Identity is the identity of an object which is a unique integer that represents its memory address. It remains constant during the object's lifetime. Every object in Python has a unique identifier obtained using the `id()` function.

### Example

Following is the example code of getting the Object Identity –

```
a = "Tutorialspoint"  
print(id(a)) # Example of getting the id of an string object
```

On executing the above code, we will get the following output –

```
2366993878000
```

**Note:** The memory address will change on every execution of the code.

## Object Type

Object Type is the type of an object defines the operations that can be performed on it. For example, integers, strings and lists have distinct types. It is defined by its class and can be accessed using the `type()` function.

**Example**

Here is the example of object type –

```
a = "Tutorialspoint"
print(type(a))
```

On executing the above code, we will get the following output –

```
<class 'str'>
```

**Object Value**

Object Value of an object is the actual data it holds. This can be a primitive value like an integer or string, or it can be more complex data structures like lists or dictionaries.

**Example**

Following is the example of the object value –

```
b = "Welcome to Tutorialspoint"
print(b)
```

On executing the above code, we will get the following output –

```
Welcome to Tutorialspoint
```

**Memory Management**

Memory management in Python is a critical aspect of the language's design by ensuring efficient use of resources while handling object lifetimes and garbage collection. Here are the key components of memory management in Python –

- **Reference Counting:** Python uses reference counting to manage memory. Each object keeps track of how many references point to it. When this count drops to zero then the memory can be freed.
- **Garbage Collection:** In addition to reference counting the Python employs a garbage collector to identify and clean up reference cycles.

**Example**

Following is the example of getting the reference counting in memory management –

```
import sys
c = [1, 2, 3]
print(sys.getrefcount(c)) # Shows the reference count
```

On executing the above code, we will get the following output –

```
2
```

**Attributes and Methods**

Python objects can have attributes and methods which are accessed using dot notation. Attributes store data while methods define the behavior.

**Example**

```
class MyClass:  
    def __init__(self, value):  
        self.value = value  
  
    def display(self):  
        print(self.value)  
  
obj = MyClass(10)  
obj.display()
```

On executing the above code, we will get the following output –

```
10
```

Finally, understanding Python's object internals helps optimize performance and debug effectively. By grasping how objects are structured and managed in memory where developers can make informed decisions when writing Python code.

# 211. Python - Memory Management

In Python, memory management is automatic, it involves handling a private heap that contains all Python objects and data structures. The Python memory manager internally ensures the efficient allocation and deallocation of this memory. This tutorial will explore Python's memory management mechanisms, including garbage collection, reference counting, and how variables are stored on the stack and heap.

## Memory Management Components

Python's memory management components provide efficient and effective utilization of memory resources throughout the execution of Python programs. Python has three memory management components –

- **Private Heap:** Acts as the main storage for all Python objects and data. It is managed internally by the Python memory manager.
- **Raw Memory Allocator:** This low-level component directly interacts with the operating system to reserve memory space in Python's private heap. It ensures there's enough room for Python's data structures and objects.
- **Object-Specific Allocators:** On top of the raw memory allocator, several object-specific allocators manage memory for different types of objects, such as integers, strings, tuples, and dictionaries.

## Memory Allocation in Python

Python manages memory allocation in two primary ways – Stack and Heap.

### Stack – Static Memory Allocation

In static memory allocation, memory is allocated at compile time and stored in the stack. This is typical for function call stacks and variable references. The stack is a region of memory used for storing local variables and function call information. It operates on a Last-In-First-Out (LIFO) basis, where the most recently added item is the first to be removed.

The stack is generally used for variables of primitive data types, such as numbers, booleans, and characters. These variables have a fixed memory size, which is known at compile-time.

#### Example

Let us look at an example to illustrate how variables of primitive types are stored on the stack. In the above example, variables named x, y, and z are local variables within the function named `example_function()`. They are stored on the stack, and when the function execution completes, they are automatically removed from the stack.

```
def my_function():
    x = 5
    y = True
    z = 'Hello'
```

```

    return x, y, z

print(my_function())
print(x, y, z)

```

On executing the above program, you will get the following output –

```

(5, True, 'Hello')
Traceback (most recent call last):
  File "/home/cg/root/71937/main.py", line 8, in <module>
    print(x, y, z)
NameError: name 'x' is not defined

```

## Heap – Dynamic Memory Allocation

Dynamic memory allocation occurs at runtime for objects and data structures of non-primitive types. The actual data of these objects is stored in the heap, while references to them are stored on the stack.

### Example

Let's observe an example for creating a list dynamically allocates memory in the heap.

```

a = [0]*10
print(a)

```

### Output

On executing the above program, you will get the following results –

```
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

## Garbage Collection in Python

Garbage Collection in Python is the process of automatically freeing up memory that is no longer in use by objects, making it available for other objects. Python's garbage collector runs during program execution and activates when an object's reference count drops to zero.

## Reference Counting

Python's primary garbage collection mechanism is reference counting. Every object in Python maintains a reference count that tracks how many aliases (or references) point to it. When an object's reference count drops to zero, the garbage collector deallocates the object.

Working of the reference counting as follows –

- **Increasing Reference Count**– It happens when a new reference to an object is created, the reference count increases.
- **Decreasing Reference Count**– When a reference to an object is removed or goes out of scope, the reference count decreases.

**Example**

Here is an example that demonstrates working of reference counting in Python.

```
import sys

# Create a string object
name = "Tutorialspoint"
print("Initial reference count:", sys.getrefcount(name))

# Assign the same string to another variable
other_name = "Tutorialspoint"
print("Reference count after assignment:", sys.getrefcount(name))

# Concatenate the string with another string
string_sum = name + ' Python'
print("Reference count after concatenation:", sys.getrefcount(name))

# Put the name inside a list multiple times
list_of_names = [name, name, name]
print("Reference count after creating a list with 'name' 3 times:",
      sys.getrefcount(name))

# Deleting one more reference to 'name'
del other_name
print("Reference count after deleting 'other_name':", sys.getrefcount(name))

# Deleting the list reference
del list_of_names
print("Reference count after deleting the list:", sys.getrefcount(name))
```

**Output**

On executing the above program, you will get the following results –

```
Initial reference count: 4
Reference count after assignment: 5
Reference count after concatenation: 5
Reference count after creating a list with 'name' 3 times: 8
Reference count after deleting 'other_name': 7
```

Reference count after deleting the list: 4

# 212. Python - Metaclasses

Metaclasses are a powerful feature in Python that allow you to customize class creation. By using metaclasses, you can add specific behaviors, attributes, and methods to classes, and allowing you to create more flexible, efficient programs. These classes provide the ability to work with metaprogramming in Python.

Metaclasses are an OOP concept present in all python code by default. Python provides the functionality to create custom metaclasses by using the keyword type. Type is a metaclass whose instances are classes. Any class created in python is an instance of type metaclass.

## Creating Metaclasses in Python

A metaclass is a class of a class that defines how a class behaves. Every class in Python is an instance of its metaclass. By default, Python uses the type() function to construct the metaclasses. However, you can define your own metaclass to customize class creation and behavior.

When defining a class, if no base classes or metaclass are explicitly specified, then Python uses type() to construct the class. Then its body is executed in a new namespace, resulting class name is locally linked to the output of type(name, bases, namespace).

### Example

Let's observe the result of creating a class object without specifying specific bases or a metaclass

```
class Demo:  
    pass  
  
obj = Demo()  
  
print(obj)
```

### Output

On executing the above program, you will get the following results –

```
<__main__.Demo object at 0x7fe78f43fe80>
```

This example demonstrates the basics of metaprogramming in Python using metaclasses. The above output indicates that obj is an instance of the Demo class, residing in memory location 0x7fe78f43fe80. This is the default behavior of the Python metaclass, allowing us to easily inspect the details of the class.

## Creating Metaclasses Dynamically

The type() function in Python can be used to create classes and metaclasses dynamically.

### Example

In this example, DemoClass will be created using type() function, and an instance of this class is also created and displayed.

```
# Creating a class dynamically using type()
DemoClass = type('DemoClass', (), {})
obj = DemoClass()
print(obj)
```

### Output

Upon executing the above program, you will get the following results –

```
<__main__.DemoClass object at 0x7f9ff6af3ee0>
```

### Example

Here is another example of creating a Metaclass with inheritance which can be done by inheriting one from another class using type() function.

```
class Demo:
    pass

Demo2 = type('Demo2', (Demo,), dict(attribute=10))
obj = Demo2()

print(obj.attribute)
print(obj.__class__)
print(obj.__class__.__bases__)
```

### Output

Following is the output –

```
10
<class '__main__.Demo2'>
((),)
```

## Customizing Metaclass Creation

In Python, you can customize how classes are created and initialized by defining your own metaclass. This customization is useful for various metaprogramming tasks, such as adding specific behavior to all instances of a class or enforcing certain patterns across multiple classes.

Customizing the classes can be done by overriding methods in the metaclass, specifically `__new__` and `__init__`.

### Example

Let's see the example of demonstrating how we can customize class creation using the `__new__` method of a metaclass in python.

```
# Define a custom metaclass

class MyMetaClass(type):

    def __new__(cls, name, bases, dct):
        dct['version'] = 1.0

        # Modify the class name
        name = 'Custom' + name

        return super().__new__(cls, name, bases, dct)

# MetaClass acts as a template for the custom metaclass

class Demo(metaclass=MyMetaClass):

    pass

# Instantiate the class

obj = Demo()

# Print the class name and version attribute

print("Class Name:", type(obj).__name__)
print("Version:", obj.version)
```

**Output**

While executing above code, you will get the following results –

```
Class Name: CustomDemo
Version: 1.0
```

**Example**

Here is another example that demonstrates how to customize the metaclass using the `__init__` in Python.

```
# Define a custom metaclass

class ByCreatingMetaClass(type):

    def __init__(cls, name, bases, dct):
        print('Initializing class', name)

        # Add a class-level attribute
        cls.version = 10
```

```
super().__init__(name, bases, dct)

# Define a class using the custom metaclass
class MyClass(metaclass=MyMetaClass):
    def __init__(self, value):
        self.value = value

    def display(self):
        print(f"Value: {self.value}, Version: {self.__class__.version}")

# Instantiate the class and demonstrate its usage
obj = MyClass(42)
obj.display()
```

## Output

While executing above code, you will get the following results –

```
Initializing class MyClass
Value: 42, Version: 10
```

# 213. Python - Metaprogramming with Metaclasses

In Python, Metaprogramming refers to the practice of writing code that has knowledge of itself and can be manipulated. The metaclasses are a powerful tool for metaprogramming in Python, allowing you to customize how classes are created and behave. Using metaclasses, you can create more flexible and efficient programs through dynamic code generation and reflection.

Metaprogramming in Python involves techniques such as decorators and metaclasses. In this tutorial, you will learn about metaprogramming with metaclasses by exploring dynamic code generation and reflection.

## Defining Metaclasses

Metaprogramming with metaclasses in Python offer advanced features of enabling advanced capabilities to your program. One such feature is the `__prepare__()` method, which allows customization of the namespace where a class body will be executed.

This method is called before any class body code is executed, providing a way to initialize the class namespace with additional attributes or methods. The `__prepare__()` method should be implemented as a class method.

### Example

Here is an example of creating a metaclass with advanced features using the `__prepare__()` method.

```
class MyMetaClass(type):

    @classmethod
    def __prepare__(cls, name, bases, **kwargs):
        print(f'Preparing namespace for {name}')

        # Customize the namespace preparation here
        custom_namespace = super().__prepare__(name, bases, **kwargs)
        custom_namespace['CONSTANT_VALUE'] = 100

        return custom_namespace

    # Define a class using the custom metaclass
    class MyClass(metaclass=MyMetaClass):
        def __init__(self, value):
            self.value = value
```

```

def display(self):
    print(f"Value: {self.value}, Constant: {self.__class__.CONSTANT_VALUE}")

# Instantiate the class
obj = MyClass(42)
obj.display()

```

**Output**

While executing above code, you will get the following results –

```

Preparing namespace for MyClass
Value: 42, Constant: 100

```

**Dynamic Code Generation with Metaclasses**

Metaprogramming with metaclasses enables the creation or modification of code during runtime.

**Example**

This example demonstrates how metaclasses in Python metaprogramming can be used for dynamic code generation.

```

class MyMeta(type):

    def __new__(cls, name, bases, attrs):
        print(f"Defining class: {name}")

        # Dynamic attribute to the class
        attrs['dynamic_attribute'] = 'Added dynamically'

        # Dynamic method to the class
        def dynamic_method(self):
            return f"This is a dynamically added method for {name}"

        attrs['dynamic_method'] = dynamic_method

        return super().__new__(cls, name, bases, attrs)

    # Define a class using the metaclass
    class MyClass(metaclass=MyMeta):
        pass

```

```
obj = MyClass()
print(obj.dynamic_attribute)
print(obj.dynamic_method())
```

**Output**

On executing above code, you will get the following results –

```
Defining class: MyClass
Added dynamically
This is a dynamically added method for MyClass
```

**Reflection and Metaprogramming**

Metaprogramming with metaclasses often involves reflection, allowing for introspection and modification of class attributes and methods at runtime.

**Example**

In this example, the MyMeta metaclass inspects and prints the attributes of the MyClass during its creation, demonstrating how metaclasses can introspect and modify class definitions dynamically.

```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        # Inspect class attributes and print them
        print(f"Class attributes for {name}: {dct}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=MyMeta):
    data = "example"
```

**Output**

On executing above code, you will get the following results –

```
Class attributes for MyClass: {'__module__': '__main__', '__qualname__': 'MyClass', 'data': 'example'}
```

# 214. Python - Mocking and Stubbing

Python mocking and stubbing are important techniques in unit testing that help to isolate the functionality being tested by replacing real objects or methods with controlled substitutes. In this chapter, we are going to understand about Mocking and Stubbing in detail –

## Python Mocking

Mocking is a testing technique in which mock objects are created to simulate the behavior of real objects.

This is useful when testing a piece of code that interacts with complex, unpredictable or slow components such as databases, web services or hardware devices.

The primary purpose of mocking is to isolate the code under test and ensure that its behavior is evaluated independently of its dependencies.

## Key Characteristics of Mocking

The following are the key characteristics of mocking in python –

- **Behavior Simulation:** Mock objects can be programmed to return specific values, raise exceptions or mimic the behavior of real objects under various conditions.
- **Interaction Verification:** Mocks can record how they were used by allowing the tester to verify that specific methods were called with the expected arguments.
- **Test Isolation:** By replacing real objects with mocks, tests can focus on the logic of the code under test without worrying about the complexities or availability of external dependencies.

## Example of Python Mocking

Following is the example of the database.get\_user method, which is mocked to return a predefined user dictionary. The test can then verify that the method was called with the correct arguments –

```
from unittest.mock import Mock

# Create a mock object
database = Mock()

# Simulate a method call
database.get_user.return_value = {"name": "Prasad", "age": 30}

# Use the mock object
user = database.get_user("prasad_id")
print(user)
```

```
# Verify the interaction
database.get_user.assert_called_with("prasad_id")
```

## Output

```
{"name": "Prasad", "age": 30}
```

## Python Stubbing

Stubbing is a related testing technique where certain methods or functions are replaced with "stubs" that return fixed, predetermined responses.

Stubbing is simpler than mocking because it typically does not involve recording or verifying interactions. Instead, stubbing focuses on providing controlled inputs to the code under test by ensuring consistent and repeatable results.

### Key Characteristics of Stubbing

The following are the key characteristics of Stubbing in python –

- **Fixed Responses:** Stubs return specific, predefined values or responses regardless of how they are called.
- **Simplified Dependencies:** By replacing complex methods with stubs, tests can avoid the need to set up or manage intricate dependencies.
- **Focus on Inputs:** Stubbing emphasizes providing known inputs to the code under test by allowing the tester to focus on the logic and output of the tested code.

### Example of Python Stubbing

Following is the example of the get\_user\_from\_db function, which is stubbed to always return a predefined user dictionary. The test does not need to interact with a real database for simplifying the setup and ensuring consistent results –

```
from unittest.mock import patch

# Define the function to be stubbed
def get_user_from_db(user_id):
    # Simulate a complex database operation
    pass

# Test the function with a stub
with patch('__main__.get_user_from_db', return_value={"name": "Prasad", "age": 25}):
    user = get_user_from_db("prasad_id")
    print(user)
```

## Output

```
{'name': 'Prasad', 'age': 25}
```

## Python Mocking Vs. Stubbing

The comparison of the Mocking and Stubbing key features, purposes and use cases gives the clarity on when to use each method. By exploring these distinctions, developers can create more effective and maintainable tests which ultimately leads to higher quality software.

The following table shows the key difference between mocking and stubbing based on the different criteria –

| Criteria                 | Mocking                                                                   | Stubbing                                           |
|--------------------------|---------------------------------------------------------------------------|----------------------------------------------------|
| Purpose                  | Simulate the behavior of real objects                                     | Provide fixed, predetermined responses             |
| Interaction Verification | Can verify method calls and arguments                                     | Typically does not verify interactions             |
| Complexity               | More complex; can simulate various behaviors                              | Simpler; focuses on providing controlled inputs    |
| Use Case                 | Isolate and test code with complex dependencies                           | Simplify tests by providing known responses        |
| Recording Behavior       | Records how methods were called                                           | Does not record interactions                       |
| State Management         | Can maintain state across calls                                           | Usually stateless; returns fixed output            |
| Framework Support        | Primarily uses <b>unittest.mock</b> with features like Mock and MagicMock | Uses unittest.mock's patch for simple replacements |
| Flexibility              | Highly flexible; can simulate exceptions and side effects                 | Limited flexibility; focused on return values      |

# 215. Python - Monkey Patching

Monkey patching in Python refers to the practice of dynamically modifying or extending code at runtime typically replacing or adding new functionalities to existing modules, classes or methods without altering their original source code. This technique is often used for quick fixes, debugging or adding temporary features.

The term "monkey patching" originates from the idea of making changes in a way that is ad-hoc or temporary, akin to how a monkey might patch something up using whatever materials are at hand.

## Steps to Perform Monkey Patching

Following are the steps that shows how we can perform monkey patching –

- First, to apply a monkey patch, we have to import the module or class we want to modify.
- In the second step we have to define a new function or method with the desired behavior.
- Replace the original function or method with the new implementation by assigning it to the attribute of the class or module.

## Example of Monkey Patching

Now let's understand the Monkey patching with the help of an example –

### Define a Class or Module to Patch

First we have to define the original class or module that we want to modify. Below is the code –

```
# original_module.py

class MyClass:
    def say_hello(self):
        return "Hello, Welcome to Tutorialspoint!"
```

### Create a Patching Function or Method

Next we have to define a function or method that we will use to monkey patch the original class or module. This function will contain the new behavior or functionality we want to add –

```
# patch_module.py

from original_module import MyClass
```

```
# Define a new function to be patched

def new_say_hello(self):
    return "Greetings!"

# Monkey patching MyClass with new_say_hello method
MyClass.say_hello = new_say_hello
```

## Test the Monkey Patch

Now we can test the patched functionality. Ensure that the patching is done before we create an instance of MyClass with the patched method –

```
# test_patch.py

from original_module import MyClass
import patch_module

# Create an instance of MyClass
obj = MyClass()

# Test the patched method
print(obj.say_hello()) # Output: Greetings!
```

## Drawbacks of Monkey Patching

Following are the draw backs of monkey patching –

- **Overuse:** Excessive monkey patching can make the code hard to understand and maintain. We have to use it judiciously and consider alternative design patterns if possible.
- **Compatibility:** Monkey patching may introduce unexpected behavior especially in complex systems or with large code-bases.

# 216. Python - Signal Handling

Signal handling in Python allows you to define custom handlers for managing asynchronous events such as interrupts or termination requests from keyboard, alarms, and even system signals. You can control how your program responds to various signals by defining custom handlers. The signal module in Python provides mechanisms to set and manage signal handlers.

A signal handler is a function that gets executed when a specific signal is received. The `signal.signal()` function allows defining custom handlers for signals. The signal module offers a way to define custom handlers that will be executed when a specific signal is received. Some default handlers are already installed in Python, which are –

- `SIGPIPE` is ignored.
- `SIGINT` is translated into a `KeyboardInterrupt` exception.

## Commonly Used Signals

Python signal handlers are executed in the main Python thread of the main interpreter, even if the signal is received in another thread. Signals can't be used for inter-thread communication.

Following are the list of some common signals and their default actions –

- **SIGINT** – Interrupt from keyboard (Ctrl+C), which raises a `KeyboardInterrupt`.
- **SIGTERM** – Termination signal.
- **SIGALRM** – Timer signal from `alarm()`.
- **SIGCHLD** – Child process stopped or terminated.
- **SIGUSR1** and **SIGUSR2** – User-defined signals.

## Setting a Signal Handler

To set a signal handler, we can use the `signal.signal()` function. It allows you to define custom handlers for signals. A handler remains installed until explicitly reset, except for `SIGCHLD`.

### Example

Here is an example of setting a signal handler using the `signal.signal()` function with the `SIGINT` handler.

```
import signal
import time

def handle_signal(signum, frame):
    print(f"Signal {signum} received")

# Setting the handler for SIGINT
signal.signal(signal.SIGINT, handle_signal)
```

```
print("Press Ctrl+C to trigger SIGINT")
while True:
    time.sleep(1)
```

**Output**

On executing the above program, you will get the following results –

```
Press Ctrl+C to trigger SIGINT
Signal 2 received
Signal 2 received
Signal 2 received
Signal 2 received
```

**Signal Handling on Windows**

On Windows, the `signal.signal()` function can only handle a limited set of signals. If you try to use a signal not supported on Windows, a `ValueError` will be raised. An `AttributeError` will also be raised if a signal name is not defined as a `SIG*` module level constant.

The supported signals on Windows are follows –

- `SIGABRT`
- `SIGFPE`
- `SIGILL`
- `SIGINT`
- `SIGSEGV`
- `SIGTERM`
- `SIGBREAK`

**Handling Timers and Alarms**

Timers and alarms can be used to schedule signal delivery after a certain amount of time.

**Example**

Let's observe following example of handling alarms.

```
import signal
import time

def handler(signum, stack):
    print('Alarm: ', time.ctime())

signal.signal(signal.SIGALRM, handler)
signal.alarm(2)
```

```
time.sleep(5)

for i in range(5):
    signal.alarm(2)
    time.sleep(5)
    print("interrupted #%d" % i)
```

**Output**

On executing the above program, you will get the following results –

```
Alarm: Wed Jul 17 17:30:11 2024
Alarm: Wed Jul 17 17:30:16 2024
interrupted #0
Alarm: Wed Jul 17 17:30:21 2024
interrupted #1
Alarm: Wed Jul 17 17:30:26 2024
interrupted #2
Alarm: Wed Jul 17 17:30:31 2024
interrupted #3
Alarm: Wed Jul 17 17:30:36 2024
interrupted #4
```

**Getting Signal Names from Numbers**

There is no straightforward way of getting signal names from numbers in Python. You can use the signal module to get all its attributes, filter out those that start with SIG, and store them in a dictionary.

**Example**

This example creates a dictionary where the keys are signal numbers and the values are the corresponding signal names. This is useful for dynamically resolving signal names from their numeric values.

```
import signal

sig_items = reversed(sorted(signal.__dict__.items()))
final = dict((k, v) for v, k in sig_items if v.startswith('SIG') and not
v.startswith('SIG_'))
print(final)
```

**Output**

On executing the above program, you will get the following results –

```
{<Signals.SIGXFSZ: 25>: 'SIGXFSZ', <Signals.SIGXCPU: 24>: 'SIGXCPU',
<Signals.SIGWINCH: 28>: 'SIGWINCH', <Signals.SIGVTALRM: 26>: 'SIGVTALRM',
<Signals.SIGUSR2: 12>: 'SIGUSR2', <Signals.SIGUSR1: 10>: 'SIGUSR1',
<Signals.SIGURG: 23>: 'SIGURG', <Signals.SIGTTOU: 22>: 'SIGTTOU',
<Signals.SIGTTIN: 21>: 'SIGTTIN', <Signals.SIGTSTP: 20>: 'SIGTSTP',
<Signals.SIGTRAP: 5>: 'SIGTRAP', <Signals.SIGTERM: 15>: 'SIGTERM',
<Signals.SIGSYS: 31>: 'SIGSYS', <Signals.SIGSTOP: 19>: 'SIGSTOP',
<Signals.SIGSEGV: 11>: 'SIGSEGV', <Signals.SIGRTMIN: 34>: 'SIGRTMIN',
<Signals.SIGRTMAX: 64>: 'SIGRTMAX', <Signals.SIGQUIT: 3>: 'SIGQUIT',
<Signals.SIGPWR: 30>: 'SIGPWR', <Signals.SIGPROF: 27>: 'SIGPROF',
<Signals.SIGIO: 29>: 'SIGIO', <Signals.SIGPIPE: 13>: 'SIGPIPE',
<Signals.SIGKILL: 9>: 'SIGKILL', <Signals.SIGABRT: 6>: 'SIGABRT',
<Signals.SIGINT: 2>: 'SIGINT', <Signals.SIGILL: 4>: 'SIGILL', <Signals.SIGHUP:
1>: 'SIGHUP', <Signals.SIGFPE: 8>: 'SIGFPE', <Signals.SIGCONT: 18>: 'SIGCONT',
<Signals.SIGCHLD: 17>: 'SIGCHLD', <Signals.SIGBUS: 7>: 'SIGBUS',
<Signals.SIGALRM: 14>: 'SIGALRM'}
```

# 217. Python - Type Hints

Python type hints were introduced in PEP 484 to bring the benefits of static typing to a dynamically typed language. Although type hints do not enforce type checking at runtime, they provide a way to specify the expected types of variables, function parameters, and return values, which can be checked by static analysis tools such as mypy. This enhances code readability, facilitates debugging, and improves the overall maintainability of the code.

Type hints in Python use annotations for function parameters, return values and variable assignments.

Python's type hints can be used to specify a wide variety of types such as basic data types, collections, complex types and custom user-defined types. The typing module provides many built-in types to represent these various types –

- Basic Data Types
- Collections Types
- Optional Types
- Union Types
- Any Type
- Type Aliases
- Generic Types
- Callable Types
- Literal Types
- NewType

Let's see each one, one after another in detail.

## Basic Data Types

In Python, when using type hints to specify basic types, we can simply use the name of the type as the annotation.

### Example

Following is the example of using the basic data types such as integer, float, string etc –

```
from typing import Optional

# Integer type
def calculate_square_area(side_length: int) -> int:
    return side_length ** 2

# Float type
def calculate_circle_area(radius: float) -> float:
    return 3.14 * radius * radius
```

```

# String type

def greet(name: str) -> str:
    return f"Hello, {name}"


# Boolean type

def is_adult(age: int) -> bool:
    return age >= 18


# None type

def no_return_example() -> None:
    print("This function does not return anything")


# Optional type (Union of int or None)

def safe_divide(x: int, y: Optional[int]) -> Optional[float]:
    if y is None or y == 0:
        return None
    else:
        return x / y


# Example usage

print(calculate_square_area(5))
print(calculate_circle_area(3.0))
print(greet("Alice"))
print(is_adult(22))
no_return_example()
print(safe_divide(10, 2))
print(safe_divide(10, 0))
print(safe_divide(10, None))

```

On executing the above code we will get the following output –

```

25
28.25999999999998
Hello, Alice
True
This function does not return anything
5.0

```

```
None
None
```

## Collections Types

In Python when dealing with collections such as lists, tuples, dictionaries, etc. in type hints we typically use the typing module to specify the collection types.

### Example

Below is the example of the Collections using in type hints –

```
from typing import List, Tuple, Dict, Set, Iterable, Generator

# List of integers
def process_numbers(numbers: List[int]) -> List[int]:
    return [num * 2 for num in numbers]

# Tuple of floats
def coordinates() -> Tuple[float, float]:
    return (3.0, 4.0)

# Dictionary with string keys and integer values
def frequency_count(items: List[str]) -> Dict[str, int]:
    freq = {}
    for item in items:
        freq[item] = freq.get(item, 0) + 1
    return freq

# Set of unique characters in a string
def unique_characters(word: str) -> Set[str]:
    return set(word)

# Iterable of integers
def print_items(items: Iterable[int]) -> None:
    for item in items:
        print(item)

# Generator yielding squares of integers up to n
```

```

def squares(n: int) -> Generator[int, None, None]:
    for i in range(n):
        yield i * i

# Example usage
numbers = [1, 2, 3, 4, 5]
print(process_numbers(numbers))

print(coordinates())

items = ["apple", "banana", "apple", "orange"]
print(frequency_count(items))

word = "hello"
print(unique_characters(word))

print_items(range(5))

gen = squares(5)
print(list(gen))

```

On executing the above code we will get the following output –

```

[2, 4, 6, 8, 10]
(3.0, 4.0)
{'apple': 2, 'banana': 1, 'orange': 1}
{'l', 'e', 'h', 'o'}
0
1
2
3
4
[0, 1, 4, 9, 16]

```

## Optional Types

In Python, Optional types are used to indicate that a variable can either be of a specified type or None. This is particularly useful when a function may not always return a value or when a parameter can accept a value or be left unspecified.

### Example

Here is the example of using the optional types in type hints –

```
from typing import Optional

def divide(a: float, b: float) -> Optional[float]:
    if b == 0:
        return None
    else:
        return a / b

result1: Optional[float] = divide(10.0, 2.0)    # result1 will be 5.0
result2: Optional[float] = divide(10.0, 0.0)    # result2 will be None

print(result1)
print(result2)
```

On executing the above code we will get the following output –

```
5.0
None
```

## Union Types

Python uses Union types to allow a variable to accept values of different types. This is useful when a function or data structure can work with various types of inputs or produce different types of outputs.

### Example

Below is the example of this –

```
from typing import Union

def square_root_or_none(number: Union[int, float]) -> Union[float, None]:
    if number >= 0:
        return number ** 0.5
    else:
        return None
```

```

result1: Union[float, None] = square_root_or_none(50)
result2: Union[float, None] = square_root_or_none(-50)

print(result1)
print(result2)

```

On executing the above code we will get the following output –

```

7.0710678118654755
None

```

## Any Type

In Python, Any type is a special type hint that indicates that a variable can be of any type. It essentially disables type checking for that particular variable or expression. This can be useful in situations where the type of a value is not known beforehand or when dealing with dynamic data.

### Example

Following is the example of using Any type in Type hint –

```

from typing import Any

def print_value(value: Any) -> None:
    print(value)

print_value(10)
print_value("hello")
print_value(True)
print_value([1, 2, 3])
print_value({'key': 'value'})

```

On executing the above code we will get the following output –

```

10
hello
True
[1, 2, 3]
{'key': 'value'}

```

## Type Aliases

Type aliases in Python are used to give alternative names to existing types. They can make code easier to read by giving clear names to complicated type annotations or combinations of types. This is especially helpful when working with nested structures or long-type hints.

### Example

Below is the example of using the Type Aliases in the Type hints –

```
from typing import List, Tuple

# Define a type alias for a list of integers
Vector = List[int]

# Define a type alias for a tuple of coordinates
Coordinates = Tuple[float, float]

# Function using the type aliases
def scale_vector(vector: Vector, factor: float) -> Vector:
    return [int(num * factor) for num in vector]

def calculate_distance(coord1: Coordinates, coord2: Coordinates) -> float:
    x1, y1 = coord1
    x2, y2 = coord2
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

# Using the type aliases
v: Vector = [1, 2, 3, 4]
scaled_v: Vector = scale_vector(v, 2.5)
print(scaled_v)

c1: Coordinates = (3.0, 4.0)
c2: Coordinates = (6.0, 8.0)
distance: float = calculate_distance(c1, c2)
print(distance)
```

On executing the above code we will get the following output –

|               |
|---------------|
| [2, 5, 7, 10] |
| 5.0           |

## Generic Types

Generic types create functions, classes or data structures that can handle any type while maintaining type safety. The typing module's TypeVar and Generic constructs make this possible. They are helpful for making reusable components that can work with various types without compromising type checking.

### Example

Here is the example of it –

```
from typing import TypeVar, List

# Define a type variable T
T = TypeVar('T')

# Generic function that returns the first element of a list
def first_element(items: List[T]) -> T:
    return items[0]

# Example usage
int_list = [1, 2, 3, 4, 5]
str_list = ["apple", "banana", "cherry"]

first_int = first_element(int_list)      # first_int will be of type int
first_str = first_element(str_list)      # first_str will be of type str

print(first_int)
print(first_str)
```

On executing the above code, we will get the following output –

```
1
apple
```

## Callable Types

Python's Callable type is utilized to show that a type is a function or a callable object. It is found in the typing module and lets you define the types of the arguments and the return type of a function. This is handy for higher-order functions.

### Example

Following is the example of using Callable type in type hint –

```
from typing import Callable
```

```

# Define a function that takes another function as an argument

def apply_operation(x: int, y: int, operation: Callable[[int, int], int]) ->
    int:

    return operation(x, y)

# Example functions to be passed as arguments

def add(a: int, b: int) -> int:
    return a + b

def multiply(a: int, b: int) -> int:
    return a * b

# Using the apply_operation function with different operations

result1 = apply_operation(5, 3, add)          # result1 will be 8
result2 = apply_operation(5, 3, multiply)     # result2 will be 15

print(result1)
print(result2)

```

On executing the above code, we will get the following output –

8

15

## Literal Types

The Literal type is used to specify that a value must be exactly one of a set of predefined values.

### Example

Below is the example –

```

from typing import Literal

def move(direction: Literal["left", "right", "up", "down"]) -> None:
    print(f"Moving {direction}")

move("left")  # Valid
move("up")    # Valid

```

On executing the above code, we will get the following output –

Moving left

Moving up

## NewType

NewType is a function in the typing module that allows us to create distinct types derived from existing ones. This can be useful for adding type safety to our code by distinguishing between different uses of the same underlying type. For example, we might want to differentiate between user IDs and product IDs even though both are represented as integers.

### Example

Below is the example –

```
from typing import NewType

# Create new types
UserId = NewType('UserId', int)
ProductId = NewType('ProductId', int)

# Define functions that use the new types
def get_user_name(user_id: UserId) -> str:
    return f"User with ID {user_id}"

def get_product_name(product_id: ProductId) -> str:
    return f"Product with ID {product_id}"

# Example usage
user_id = UserId(42)
product_id = ProductId(101)

print(get_user_name(user_id))  # Output: User with ID 42
print(get_product_name(product_id))  # Output: Product with ID 101
```

On executing the above code, we will get the following output –

User with ID 42

Product with ID 101

# 218. Python - Automation Tutorial

## Automation using Python

Automation with Python involves using programming techniques to perform tasks automatically, typically without human intervention. Python provides various libraries to make it a powerful tool for automating different types of repetitive tasks including, task scheduling, web scraping, GUI automation and many more.

Utilizing the Python's extensive libraries can create automation solutions to fit specific needs.

## Python Libraries for Automation

Following are the few of the Python libraries that are commonly used for automation –

- [\*\*unittest\*\*](#) – Unit testing framework in Python.
- [\*\*Selenium\*\*](#) – Web automation framework for testing web applications across different browsers.
- [\*\*Beautiful Soup\*\*](#) – Library for parsing HTML and XML documents, used for web scraping.
- [\*\*pandas\*\*](#) – Data manipulation library, useful for automating data analysis tasks.
- [\*\*requests\*\*](#) – HTTP library for sending HTTP requests and handling responses.
- [\*\*PyAutoGUI\*\*](#) – GUI automation library for simulating mouse and keyboard actions.

In this tutorial, you will learn about the various aspects of automation using Python, from scheduling tasks with schedule module, web scraping with BeautifulSoup, and GUI automation with PyAutoGUI.

Here we are providing practical examples and please ensure that you have the necessary modules installed before executing the provided codes in your local system using the below commands in your command prompt –

```
pip install schedule  
pip install beautifulsoup4  
pip install pyautogui
```

## Python Automation with "schedule" Module

The schedule module in Python allows scheduling tasks to run automatically at specified intervals or times. It's ideal for automating repetitive tasks such as job scheduling, periodic tasks, or time-based events.

### Example

This example demonstrates the application of Python for automating routine tasks such as scheduling weekly meetings, setting alarms, and more, using Python's schedule module.

```
import schedule  
import time
```

```

# Function definitions for scheduled tasks

def alarm():
    print("Time to restart the server    ")

def job():
    print("Starting daily tasks    ")

def meet():
    print('Weekly standup meeting    ')

# Schedule tasks
schedule.every(5).seconds.do(alarm) # Run every 5 seconds
schedule.every().day.at("10:30").do(job) # Run daily at 10:30 AM
schedule.every().monday.at("12:15").do(meet) # Run every Monday at 12:15 PM

# Main loop to execute scheduled tasks
while True:
    schedule.run_pending()
    time.sleep(0.5)

```

## Output

On executing the above code we will get the following output –

```

Time to restart the server
Time to restart the server
Time to restart the server
...

```

## Python Web Scraping

Web scraping is a straightforward example of automation task. It is a process of automatically extracting information from websites. This technique involves fetching web pages and extracting the required data from the HTML content.

Web scraping is widely used for data mining, data analysis, and automated tasks where collecting information from the web is necessary.

### Example

This example demonstrates how to use BeautifulSoup and the urllib.request module to scrape a webpage for URLs containing the keyword "python".

```

from bs4 import BeautifulSoup
from urllib.request import urlopen
import re

html = urlopen("https://www.tutorialspoint.com/python/index.htm")
content = html.read()
soup = BeautifulSoup(content, "html.parser")
for a in soup.findAll('a', href=True):
    if re.findall('python', a['href']):
        print("Python URL:", a['href'])

```

## Output

On executing the above code we will get the following output –

```

Python URL: /machine_learning_with_python/index.htm
Python URL: /python_pandas/index.htm
Python URL: /python/index.htm
Python URL: /python/index.htm
Python URL: /python_pandas/index.htm
Python URL: /python_pillow/index.htm
Python URL: /machine_learning_with_python/index.htm
Python URL: /python_technologies_tutorials.htm
Python URL: /python/index.htm
.....

```

## Automating Mouse Movements with "pyautogui"

The pyautogui module allows your Python scripts to automate mouse movements, keyboard inputs, and window interactions, useful for testing GUI applications or automating repetitive desktop tasks.

### Example

This example demonstrates how the Python's pyautogui module can be used to automate mouse movements in a square pattern.

```

import pyautogui

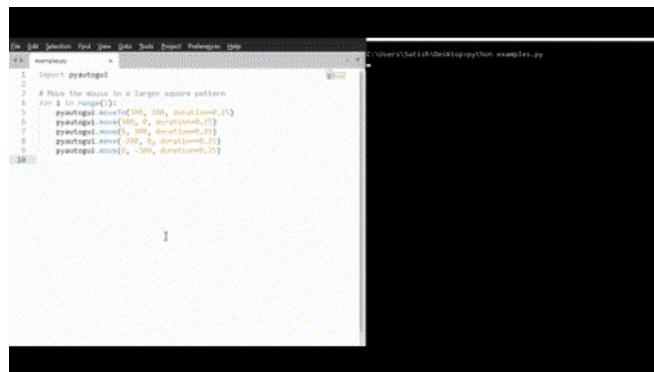
# Move the mouse in a larger square pattern
for i in range(5):
    pyautogui.moveTo(300, 100, duration=0.25)
    pyautogui.move(300, 0, duration=0.25)

```

```
pyautogui.move(0, 300, duration=0.25)
pyautogui.move(-300, 0, duration=0.25)
pyautogui.move(0, -300, duration=0.25)
```

## Output

On executing the above code we will get the following output –



## Automating Unit Testing in Python

The unittest module in Python is used for automated testing. Automated unit testing is the practice of writing tests for individual units of code, such as functions or methods, and running these tests automatically to ensure they behave as expected.

The unittest module in Python is a built-in library that provides a framework for creating and running automated tests.

### Example

This example demonstrates how to use the unittest module in Python to perform automated testing on string methods. Each test case method begins with the prefix `test_`, which is recognized by the unittest framework as a method to run.

```
# importing unittest module
import unittest

class TestingStringMethods(unittest.TestCase):
    # string equal
    def test_string_equality(self):
        # if both arguments are then it's succes
        self.assertEqual('ttp' * 5, 'ttptptptpttp')

    # comparing the two strings
    def test_string_case(self):
        # if both arguments are then it's succes
```

```

        self.assertEqual('Tutorialspoint'.upper(), 'TUTORIALSPOINT')

# checking whether a string is upper or not
def test_is_string_upper(self):
    # used to check whether the statement is True or False
    self.assertTrue('TUTORIALSPOINT'.isupper())
    self.assertFalse('TUTORIALSpoint'.isupper())

def test_string_startswith(self):
    # Used the startswith() to identify the start of the string
    self.assertTrue('tutorialspoint'.startswith('putor'))
    self.assertFalse('tutorialspoint'.startswith('point'))

# running the tests
unittest.main(verbosity=2)

```

**Output**

On executing the above code we will get the following output –

```

test_is_string_upper (__main__.TestingStringMethods) ... ok
test_string_case (__main__.TestingStringMethods) ... ok
test_string_equality (__main__.TestingStringMethods) ... ok
test_string_startswith (__main__.TestingStringMethods) ... FAIL

=====
FAIL: test_string_startswith (__main__.TestingStringMethods)
-----
Traceback (most recent call last):
  File "main.py", line 23, in test_string_startswith
    self.assertTrue('tutorialspoint'.startswith('putor'))
AssertionError: False is not true
-----
Ran 4 tests in 0.000s

FAILED (failures=1)

```



# 219. Python - Humanize Package

The Humanize Package in Python is a library which is specifically designed to convert numerical values, dates, times and file sizes into formats that are more easily understood by humans. The usage of this package are as follows:

- This package is essential for creating user-friendly interfaces and readable reports where data interpretation needs to be quick and intuitive.
- The primary goal of the Humanize Package is to bridge the gap between raw data and human understanding.

Although computers and databases excel at processing raw numerical data but these formats can be challenging for humans to quickly grasp. The Humanize Package tackles this issue by converting these data points into more intuitive and user-friendly formats.

## Installation of Humanize Package

To install the `humanize` package in Python, we can use `pip` which is the standard package manager for Python. Following code has to be run in the command line or terminal to install Humanize Package –

```
pip install humanize
```

After installation we can verify if `humanize` is installed correctly by running a Python interpreter and importing the Humanize package by using the below code –

```
import humanize
```

## Different Utilities in Humanize Package

The Humanize package in Python provides a wide range of utilities that transform data into human-readable formats by enhancing usability and comprehension. Let's explore the different utilities offered by `humanize` package in detail –

### Number Utilities

The Humanize package in Python provides a large number of utilities that enhance the readability and comprehension of numerical data. These utilities convert numbers into formats that are more natural and understandable for humans.

#### Integer Formatting

The Integer Formatting utility converts large integers into strings with commas for improved readability. Following is the example of applying the integer formatting utility –

```
import humanize  
print(humanize.intcomma(123456))
```

Output

```
123,456
```

## Integer Word Representation

The Integer Word Representation converts large integers into their word representation for easier understanding, especially for very large numbers. Below is the example of it –

```
import humanize
print(humanize.intword(12345678908545944))
```

### Output

```
12.3 quadrillion
```

## Ordinal Numbers

The Ordinal numbers converts integers into their ordinal form. For example, 1 will be given as 1st and 2 as 2nd. Below is the example converting 3 as 3rd –

```
import humanize
print(humanize.ordinal(3))
```

### Output

```
3rd
```

## AP Numbers

These converts the integers into their corresponding words. Here is the example of it –

```
import humanize
print(humanize.apnumber(9))
```

### Output

```
nine
```

## Fractional Units

This converts decimal numbers into fractions for more intuitive representation. Following is the example of it –

```
import humanize
print(humanize.fractional(0.5))
```

### Output

```
1/2
```

## File Size Utilities

As we already know that the `humanize` package in Python provides several utilities among them. File Size Utilities is one which is specifically designed to convert raw byte values into human-readable file sizes.

These utilities help to make file sizes more understandable by converting them into formats that are easier to read and interpret. Here is a detailed overview of the file size utilities available in the `humanize` package –

### File Size Formatting using `naturalsize()`

The `naturalsize()` function is the primary utility for converting file sizes into human-readable formats. It automatically selects the appropriate units such as bytes, KB, MB, GB based on the size provided.

#### Syntax

Following is the syntax of the `naturalsize()` function of the File Size Utilities of Python Humanize package –

```
humanize.naturalsize(value, binary, gnu, format)
```

#### Parameter

Below are the parameters of the `naturalsize()` function of the python `humanize` package –

- **value:** The file size in bytes.
- **binary:** A boolean flag to indicate whether to use binary units. The default value is False.
- **gnu:** A boolean flag to indicate whether to use GNU-style output and the default value is False.
- **format:** A string to specify the output format. The default value is "%.1f".

#### Example

Following is the example of using the `naturalsize()` of `humanize` package in python –

```
import humanize

# Default usage with decimal units
file_size = 123456789
print(f"File size: {humanize.naturalsize(file_size)}")

# Using binary units
print(f"File size (binary): {humanize.naturalsize(file_size, binary=True)}")

# Using GNU-style prefixes
print(f"File size (GNU): {humanize.naturalsize(file_size, gnu=True)}")

# Custom format
```

```
print(f"File size (custom format): {humanize.naturalsize(file_size,
format='%.2f')}")
```

Following is the output –

```
File size: 123.5 MB
File size (binary): 117.7 MiB
File size (GNU): 117.7M
File size (custom format): 123.46 MB
```

## Date Time Utilities

The Humanize package in Python provides several utilities for making dates and times more readable. These utilities transform date-time objects into formats that are easier for humans to understand such as relative times, natural dates and more. Following is the detailed overview of the date and time utilities offered by the humanize package –

### Natural Time

The Natural Time converts date-time objects into human-readable relative times such as 2 days ago, 3 hours ago. Following is the example of natural time –

```
import humanize
from datetime import datetime, timedelta

past_date = datetime.now() - timedelta(days=2)
print(humanize.naturaltime(past_date))
```

### Output

```
2 days ago
```

### Natural Date

The Natural Date formats specific dates into a readable format like "July 11, 2024". Here is the example –

```
import humanize
from datetime import datetime
some_date = datetime(2022, 7, 8)
print(humanize.naturaldate(some_date))
```

### Output

```
Jul 08 2022
```

## Natural Day

The Natural Day provides a human-readable representation of a date by considering today's date for contextual relevance, for example "today", "tomorrow", "yesterday" etc. Below is the example of it –

```
import humanize
from datetime import datetime, timedelta
today = datetime.now()
tomorrow = today + timedelta(days=1)
print(humanize.naturalday(today))
print(humanize.naturalday(tomorrow))
```

## Output

```
today
tomorrow
```

## Precise Delta

The Precise Delta converts time duration into human-readable strings by breaking down into days, hours, minutes and second. Here is the example of it –

```
import humanize
from datetime import timedelta
duration = timedelta(days=2, hours=3, minutes=30)
print(humanize.precisedelta(duration))
```

## Output

```
2 days, 3 hours and 30 minutes
```

## Duration Utilities

The humanize package in Python also includes the Duration Utilities for converting duration (time intervals) into human-readable formats. These utilities help to present duration in a way that is understandable and meaningful to users. Here's an overview of the duration utilities provided by humanize package –

### Duration Formatting using naturaldelta()

The naturaldelta() function converts time deltas (duration) into human-readable strings that describe the duration in a natural language format such as "2 hours ago", "3 days from now", etc.

Following is the example of using the naturaldelta() function of the Python Humanize Package –

```
from datetime import timedelta
```

```
import humanize

Using naturaldelta for time durations
delta1 = timedelta(days=3, hours=5)
print(f"Time duration: {humanize.naturaldelta(delta1)} from now")

Example of a future duration (delta2)
delta2 = timedelta(hours=5)
print(f"Future duration: in {humanize.naturaldelta(delta2)}")

Example of a past duration (delta3)
delta3 = timedelta(days=1, hours=3)
print(f"Past duration: {humanize.naturaldelta(delta3)} ago")
```

## Output

```
Time duration: 3 days from now
Future duration: in 5 hours
Past duration: a day ago
```

# 220. Python - Context Managers

**Context managers** in Python provide a powerful way to manage resources efficiently and safely. A context manager in Python is an object that defines a runtime context for use with the **with** statement. It ensures that setup and cleanup operations are performed automatically.

For instance, when working with file operations, context managers handle the opening and closing of files, ensuring that resources are managed correctly.

## How Context Managers Work?

Python context managers work by implementing the `__enter__()` and `__exit__()` methods (or their asynchronous equivalents for `async` operations). These methods ensure that resources are correctly acquired and released. Also, Python's `contextlib` module further simplifies the creation of custom context managers.

### Example

Here's a simple example demonstrating how a context manager works with file operations in Python.

```
with open('example.txt', 'w') as file:  
    file.write('Hello, TutorialsPoint!')
```

In this example, a file is opened in the write mode, and then automatically closed when the block inside the `with` statement is exited.

## Python Context Manager Types

Python supports both synchronous and asynchronous context managers. Each type has specific methods that need to be implemented to manage the life cycle of the context.

### Synchronous Context Managers

A synchronous context managers are implemented using the `__enter__()` and `__exit__()` methods.

#### 1. The `__enter__()` Method

The `__enter__(self)` method is called when execution enters the context of the `with` statement. This method should return the resource to be used within the `with` block.

### Example

Here is a simple example of creating our own context manager using the `__enter__()` and `__exit__()` methods.

```
class MyContextManager:  
    def __enter__(self):  
        print("Entering the context")
```

```

    return self

def __exit__(self, exc_type, exc_value, traceback):
    print("Exiting the context")

with MyContextManager():
    print("body")

```

On executing the above code, you will get the following output –

```

Entering the context
body
Exiting the context

```

## 2. The `__exit__()` Method

The `__exit__(self, exc_type, exc_value, traceback)` method is called when execution leaves the context of the `with` statement. It can handle exceptions if any occur, and it returns a Boolean flag indicating if the exception should be suppressed.

This example demonstrates creating the our own context manager and how the `__exit__()` methods handle exceptions.

```

class MyContextManager:

    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
        if exc_type:
            print("An exception occurred")
        return True # Suppress exception

    with MyContextManager():
        print("body")
        name = "Python"/3 #to raise an exception

```

While executing the above code, you will get the following output –

```

Entering the context
body
Exiting the context

```

An exception occurred

## Asynchronous Context Managers

Similar to the synchronous context managers, Asynchronous context managers are also implemented using the two methods which are `__aenter__()` and `__aexit__()`. These are used within `async with` statements.

**The `__aenter__(self)` Method** – It must return an awaitable that will be awaited when entering the context.

**`__aexit__(self, exc_type, exc_value, traceback)` Method** – It must return an awaitable that will be awaited when exiting the context.

### Example

Following is the example of creating an asynchronous context manager class –

```
import asyncio

class AsyncContextManager:

    async def __aenter__(self):
        print("Entering the async context class")
        return self

    async def __aexit__(self, exc_type, exc_value, traceback):
        print("Exiting the async context class")
        if exc_type:
            print("Exception occurred")
        return True

    async def main():
        async with AsyncContextManager():
            print("Inside the async context")
            name = "Python"/3 #to raise an exception

    asyncio.run(main())
```

On executing the above code, you will get the following output –

```
Entering the async context class
Inside the async context
Exiting the async context class
Exception occurred
```

## Creating Custom Context Managers

The `contextlib` module from the Python standard library provides the utilities to create context managers more easily.

### Using the `contextlib.contextmanager()` Function

The `contextlib.contextmanager()` function is a decorator allows you to create factory functions for with statement context managers. It eliminates the need to define a separate class or implement the `__enter__()` and `__exit__()` methods individually.

#### Example

Here's an example using the `contextlib.contextmanager` to create a context manager function.

```
from contextlib import contextmanager

@contextmanager
def my_context_manager():
    print("Entering the context manager method")
    try:
        yield
    finally:
        print("Exiting the context manager method")

with my_context_manager():
    print("Inside the context")
```

On executing the above code, you will get the following output –

```
Entering the context manager method
Inside the context
Exiting the context manager method
```

### Using the `contextlib.asynccontextmanager()` Function

The `contextlib` module also provides `asynccontextmanager`, specifically designed for creating asynchronous context managers. It is similar to `contextmanager` and eliminates the need to define a separate class or implement the `__aenter__()` and `__aexit__()` methods individually.

#### Example

Here's an example demonstrating the usage of `contextlib.asynccontextmanager()` to create an asynchronous context manager function.

```
import asyncio
from contextlib import asynccontextmanager
```

```
@asynccontextmanager
async def async_context_manager():
    try:
        print("Entering the async context")
        # Perform async setup tasks if needed
        yield
    finally:
        # Perform async cleanup tasks if needed
        print("Exiting the async context")

async def main():
    async with async_context_manager():
        print("Inside the async context")
        await asyncio.sleep(1)  # Simulating an async operation

# Run the asyncio event loop
asyncio.run(main())
```

On executing the above code, you will get the following output –

```
Entering the async context
Inside the async context
Exiting the async context
```

# 221. Python - Coroutines

Python Coroutines are a fundamental concept in programming that extend the capabilities of traditional functions. They are particularly useful for asynchronous programming and complex data processing pipelines.

Coroutines are an extension of the concept of functions and generators. They are designed to perform cooperative multitasking and manage asynchronous operations.

In traditional functions i.e. subroutines which have a single entry and exit point whereas coroutines can pause and resume their execution at various points by making them highly flexible.

## Key Characteristics of Coroutines

Following are the key characteristics of Coroutines in python –

- **Multiple Entry Points:** Coroutines are not constrained to a single entry point like traditional functions. They can pause their execution at certain points, when they hit a yield statement and resume later. This allows coroutines to handle complex workflows that involve waiting for or processing asynchronous data.
- **No Central Coordinator:** When we see traditional functions i.e subroutines, which are often coordinated by a main function but coroutines operate more independently. They can interact with each other in a pipeline fashion where data flows through a series of coroutines, each performing a different task.
- **Cooperative Multitasking:** Coroutines enable cooperative multitasking. This means that instead of relying on the operating system or runtime to switch between tasks the programmer controls when coroutines yield and resume by allowing for more fine-grained control over execution flow.

## Subroutines Vs. Coroutines

Subroutines are the traditional functions with a single entry point and no inherent mechanism for pausing or resuming execution. They are called in a defined sequence and handle tasks with straightforward control flow.

Coroutines are the advanced functions with multiple entry points that can pause and resume their execution. They are useful for tasks that require asynchronous execution, complex control flows and data pipelines. They support cooperative multitasking by allowing the programmer to control when execution switches between tasks.

The following table helps in understanding the key differences and similarities between subroutines and coroutines by making us easier to grasp their respective roles and functionalities in programming.

| Criteria   | Subroutines                                   | Coroutines                                                           |
|------------|-----------------------------------------------|----------------------------------------------------------------------|
| Definition | A sequence of instructions performing a task. | A generalization of subroutines that can pause and resume execution. |

|                   |                                                                                                 |                                                                                                    |
|-------------------|-------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Entry Points      | Single entry point.                                                                             | Multiple entry points; can pause and resume execution.                                             |
| Execution Control | Called by a main function or control structure.                                                 | These can suspend execution and be resumed later and programmer controls switching.                |
| Purpose           | Perform a specific task or computation.                                                         | Manage asynchronous operations, cooperative multitasking and complex workflows.                    |
| Calling Mechanism | Typically called by a main function or other subroutines.                                       | Invoked and controlled using 'next()', 'send()', and 'close()' methods.                            |
| Data Handling     | No built-in mechanism for handling data exchanges; typically uses parameters and return values. | Can receive and process data using 'yield' with 'send()'.                                          |
| State Management  | No inherent mechanism to maintain state between calls.                                          | Maintains execution state between suspensions and can resume from where it left off.               |
| Usage             | These are used for modularizing code into manageable chunks.                                    | These are used for asynchronous programming, managing data pipelines and cooperative multitasking. |
| Concurrency       | Not inherently designed for concurrent execution; typically used in sequential programming.     | Supports cooperative multitasking and can work with asynchronous tasks.                            |
| Example Usage     | Helper functions, utility functions.                                                            | Data pipelines, asynchronous tasks, cooperative multitasking.                                      |
| Control Flow      | Execution follows a linear path through the code.                                               | Execution can jump back and forth between coroutines based on yield points.                        |

## Execution of Coroutines

Coroutines are initiated with the `__next__()` method which starts the coroutine and advances execution to the first yield statement. The coroutine then waits for a value to be sent to it. The `send()` method is used to send values to the coroutine which can then process these values and potentially yield results.

### Example of Basic Coroutine

A coroutine uses the `yield` statement which can both send and receive values. Unlike a generator which yields values for iteration where as a coroutine typically uses `yield` to receive input and perform actions based on that input. Following is the basic example of the Python coroutine –

```
def print_name(prefix):
    print(f"Searching prefix: {prefix}")
    while True:
        name = (yield)
        if prefix in name:
            print(name)

# Instantiate the coroutine
corou = print_name("Welcome to")

# Start the coroutine
corou.__next__()

# Send values to the coroutine
corou.send("Tutorialspoint")
corou.send("Welcome to Tutorialspoint")
```

### Output

```
Searching prefix: Welcome to
Welcome to Tutorialspoint
```

### Closing a Coroutine

Coroutines can run indefinitely so it's important to close them properly when they are no longer needed. The `close()` method terminates the coroutine and handles cleanup. If we attempt to send data to a closed coroutine, it will raise a `StopIteration` exception.

### Example

Following is the example of closing a coroutine in python –

```
def print_name(prefix):
    print(f"Searching prefix: {prefix}")
    try:
        while True:
            name = (yield)
```

```

        if prefix in name:
            print(name)
    except GeneratorExit:
        print("Closing coroutine!!")

# Instantiate and start the coroutine
corou = print_name("Come")
corou.__next__()

# Send values to the coroutine
corou.send("Come back Thank You")
corou.send("Thank you")

# Close the coroutine
corou.close()

```

**Output**

```

Searching prefix: Come
Come back Thank You
Closing coroutine!!

```

**Chaining Coroutines for Pipelines**

Coroutines can be chained together to form a processing pipeline which allows data to flow through a series of stages. This is particularly useful for processing sequences of data in stages where each stage performs a specific task.

**Example**

Below is the example which shows chaining coroutines for pipelines –

```

def producer(sentence, next_coroutine):
    ...
    Splits the input sentence into tokens and sends them to the next coroutine.
    ...
    tokens = sentence.split(" ")
    for token in tokens:
        next_coroutine.send(token)
    next_coroutine.close()

```

```
def pattern_filter(pattern="ing", next_coroutine=None):
    ...
    Filters tokens based on the specified pattern and sends matching tokens to
    the next coroutine.
    ...
    print(f"Searching for {pattern}")
    try:
        while True:
            token = (yield)
            if pattern in token:
                next_coroutine.send(token)
    except GeneratorExit:
        print("Done with filtering!!")
        next_coroutine.close()

def print_token():
    ...
    Receives tokens and prints them.
    ...
    print("I'm the sink, I'll print tokens")
    try:
        while True:
            token = (yield)
            print(token)
    except GeneratorExit:
        print("Done with printing!")

# Setting up the pipeline
pt = print_token()
pt.__next__()

pf = pattern_filter(next_coroutine=pt)
pf.__next__()

sentence = "Tutorialspoint is welcoming you to learn and succeed in Career!!!"
```

```
producer(sentence, pf)
```

**Output**

```
I'm the sink, I'll print tokens
Searching for ing
welcoming
Done with filtering!!
Done with printing!
```

## 222. Python - Descriptors

Python Descriptors are a way to customize the access, assignment and deletion of object attributes. They provide a powerful mechanism for managing the behavior of attributes by defining methods that get, set and delete their values. Descriptors are often used to implement properties, methods, and attribute validation.

A descriptor is any object that implements at least one of the methods such as `__get__`, `__set__` and `__delete__`. These methods control how an attribute's value is accessed and modified.

### How Python Descriptors Work?

When an attribute is accessed on an instance then Python looks up the attribute in the instance's class. If the attribute is found and it is a descriptor, then Python invokes the appropriate descriptor method instead of simply returning the attribute's value. This allows the descriptor to control what happens during attribute access.

The descriptor protocol is a low-level mechanism that is used by many high-level features in Python such as properties, methods, static methods, and class methods. Descriptors can be used to implement patterns like lazy loading, type checking and computed properties.

### Descriptor Methods

Python Descriptors involve three main methods namely `__get__()`, `__set__()` and `__delete__()`. As we already discussed above these methods control the behavior of attribute access, assignment and deletion, respectively.

#### 1. The `__get__()` Method

The `__get__()` method in descriptors is a key part of the descriptor protocol in Python. It is called to retrieve the value of an attribute from an instance or from the class. Understanding how the `__get__()` method works is crucial for creating custom descriptors that can manage attribute access in sophisticated ways.

#### Syntax

The following is the syntax of Python Descriptor `__get__` method –

```
def __get__(self, instance, owner):
    """
    instance: the instance that the attribute is accessed through, or None when
    accessed through the owner class.
    owner: the owner class where the descriptor is defined.
    """
```

#### Parameters

Below are the parameters of this method –

- **self:** The descriptor instance.
- **instance:** The instance of the class where the attribute is accessed. It is None when the attribute is accessed through the class rather than an instance.
- **owner:** The class that owns the descriptor.

### Example

Following is the basic example of `__get__()` method in which it returns the stored value `_value` when `obj.attr` is accessed –

```
class Descriptor:

    def __get__(self, instance, owner):
        if instance is None: return self
        return instance._value


class MyClass:

    attr = Descriptor()

    def __init__(self, value):
        self._value = value

    obj = MyClass(42)
    print(obj.attr)
```

### Output

```
42
```

## 2. The `__set__()` Method

The `__set__()` method is a part of the descriptor protocol in Python and is used to control the behavior of setting an attribute's value. When an attribute managed by a descriptor is assigned a new value, the `__set__()` method is called by allowing the user to customize or enforce rules for the assignment.

### Syntax

The following is the syntax of Python Descriptor `__set__()` method –

```
def __set__(self, instance, value):
    """
    instance: the instance of the class where the attribute is being set.
    value: the value to assign to the attribute.
    """
```

### Parameters

Below are the parameters of this method –

- **self:** The descriptor instance.
- **instance:** The instance of the class where the attribute is being set.
- **value:** The value being assigned to the attribute.

### Example

Following is the basic example of `__set__()` method which ensures that the value assigned to `attr` is an integer –

```
class Descriptor:

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError("Value must be an integer")
        instance._value = value


class MyClass:

    attr = Descriptor()

    def __init__(self, value):
        self.attr = value

obj = MyClass(42)
print(obj.attr)
obj.attr = 100
print(obj.attr)
```

### Output

```
<__main__.Descriptor object at 0x000001E5423ED3D0>
<__main__.Descriptor object at 0x000001E5423ED3D0>
```

## 3. The `__delete__()` Method

The `__delete__()` method in the descriptor protocol allows us to control what happens when an attribute is deleted from an instance. This can be useful for managing resources, cleaning up or enforcing constraints when an attribute is removed.

### Syntax

The following is the syntax of Python Descriptor `__delete__()` method –

```
def __delete__(self, instance):
    """
    instance: the instance of the class from which the attribute is being
    deleted.
    """
```

## Parameters

Below are the parameters of this method –

- **self:** The descriptor instance.
- **instance:** The instance of the class where the attribute is being deleted.

## Example

Following is the basic example of `__set__()` method which ensures that the value assigned to attr is an integer –

```
class LoggedDescriptor:

    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        return instance.__dict__.get(self.name)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        if self.name in instance.__dict__:
            print(f"Deleting {self.name} from {instance}")
            del instance.__dict__[self.name]
        else:
            raise AttributeError(f"{self.name} not found")

class Person:

    name = LoggedDescriptor("name")
    age = LoggedDescriptor("age")

    def __init__(self, name, age):
        self.name = name
        self.age = age

# Example usage
p = Person("Tutorialspoint", 30)
print(p.name)
print(p.age)
```

```
del p.name
print(p.name)

del p.age
print(p.age)
```

**Output**

```
Tutorialspoint
30
Deleting name from <__main__.Person object at 0x0000021A1A67E2D0>
None
Deleting age from <__main__.Person object at 0x0000021A1A67E2D0>
None
```

## Types of Python Descriptors

In Python descriptors can be broadly categorized into two types based on the methods they implement. They are –

- Data Descriptors
- Non-data Descriptors

Let's see about the two types of python descriptors in detail for our better understanding.

### 1. Data Descriptors

Data descriptors are a type of descriptor in Python that define both `__get__()` and `__set__()` methods. These descriptors have precedence over instance attributes which means that the descriptor's `__get__()` and `__set__()` methods are always called, even if an instance attribute with the same name exists.

**Example**

Below is the example of a data descriptor that ensures an attribute is always an integer and logs access and modification operations –

```
class Integer:

    def __get__(self, instance, owner):
        print("Getting value")
        return instance._value

    def __set__(self, instance, value):
        print("Setting value")
```

```

if not isinstance(value, int):
    raise TypeError("Value must be an integer")
instance._value = value

def __delete__(self, instance):
    print("Deleting value")
    del instance._value

class MyClass:
    attr = Integer()

# Usage
obj = MyClass()
obj.attr = 42
print(obj.attr)
obj.attr = 100
print(obj.attr)
del obj.attr

```

**Output**

```

Setting value
Getting value
42
Setting value
Getting value
100
Deleting value

```

**2. Non-data Descriptors**

Non-data descriptors are a type of descriptor in Python that define only the `__get__()` method. Unlike data descriptors, non-data descriptors can be overridden by instance attributes. This means that if an instance attribute with the same name exists, it will take precedence over the non-data descriptor.

**Example**

Following is an example of a non-data descriptor that provides a default value if the attribute is not set on the instance –

```
class Default:
```

```

def __init__(self, default):
    self.default = default

def __get__(self, instance, owner):
    return getattr(instance, '_value', self.default)

class MyClass:
    attr = Default("default_value")

# Usage
obj = MyClass()
print(obj.attr)
obj._value = "Tutorialspoint"
print(obj.attr)

```

**Output**

```

default_value
Tutorialspoint

```

**Data Descriptors Vs. Non-data Descriptors**

Understanding the differences between Data Descriptors and Non-Data Descriptors is crucial for leveraging their capabilities effectively.

| Criteria          | Data Descriptors                                                                                                                                                 | Non-Data Descriptors                                          |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| <b>Definition</b> | Implements both <code>__get__()</code> , <code>__set__()</code> methods, and the <code>__delete__()</code> method optionally.                                    | Implements only <code>__get__()</code> method.                |
| <b>Methods</b>    | <code>__get__(self, instance, owner)</code><br><code>__set__(self, instance, value)</code><br><code>__delete__(self, instance)</code> (optional)                 | <code>__get__(self, instance, owner)</code>                   |
| <b>Precedence</b> | Takes precedence over instance attributes.                                                                                                                       | Overridden by instance attributes.                            |
| <b>Use Cases</b>  | Attribute validation and enforcement,<br>Managed attributes (e.g., properties),<br>Logging attribute access and modification,<br>Enforcing read-only attributes. | Method binding,<br>Caching and,<br>Providing default values.. |

Finally, we can say Descriptors in Python provides a powerful mechanism for managing attribute access and modification. Understanding the differences between data descriptors and non-data descriptors as well as their appropriate use cases is essential for creating robust and maintainable Python code.

By leveraging the descriptor protocol developers can implement advanced behaviors such as type checking, caching and read-only properties.

# 223. Python - Diagnosing and Fixing Memory Leaks

Memory leaks occur when a program incorrectly manages memory allocations resulting in reduced available memory and potentially causing the program to slow down or crash.

In Python, memory management is generally handled by the interpreter but memory leaks can still happen especially in long-running applications. Diagnosing and fixing memory leaks in Python involves understanding how memory is allocated, identifying problematic areas and applying appropriate solutions.

## Causes of Memory Leaks in Python

Memory leaks in Python can arise from several causes, primarily revolving around how objects are referenced and managed. Here are some common causes of memory leaks in Python –

### 1. Unreleased References

When objects are no longer needed but still referenced somewhere in the code then they are not de-allocated which leads to memory leaks. Here is an example –

```
def create_list():
    my_list = [1] * (10**6)
    return my_list

my_list = create_list()
# If my_list is not cleared or reassigned, it continues to consume memory.
print(my_list)
```

### Output

```
[1, 1, 1, 1,
.....
.....
1, 1, 1]
```

### 2. Circular References

Circular references in Python can lead to memory leaks if not managed properly but Python's cyclic garbage collector can handle many cases automatically.

For understanding how to detect and break circular references, we can use the tools such as the `gc` and `weakref` modules. These tools are crucial for efficient memory management in complex Python applications. Following is the example of circular references –

```
class Node:
```

```

def __init__(self, value):
    self.value = value
    self.next = None

a = Node(1)
b = Node(2)
a.next = b
b.next = a
# 'a' and 'b' reference each other, creating a circular reference.

```

### 3. Global Variables

Variables declared at the global scope persist for the lifetime of the program which potentially causing memory leaks if not managed properly. Below is an example –

```

large_data = [1] * (10**6)

def process_data():
    global large_data
    # Use large_data
    pass

# large_data remains in memory as long as the program runs.

```

### 4. Long-Lived Objects

Objects that persist for the lifetime of the application can cause memory issues if they accumulate over time. Here is the example –

```

cache = {}

def cache_data(key, value):
    cache[key] = value

# Cached data remains in memory until explicitly cleared.

```

### 5. Improper Use of Closures

Closures that capture and retain references to large objects can inadvertently cause memory leaks. Below is the example –

```
def create_closure():
```

```

large_object = [1] * (10**6)

def closure():
    return large_object

return closure

my_closure = create_closure()
# The large_object is retained by the closure, causing a memory leak.

```

## Tools for Diagnosing Memory Leaks

Diagnosing memory leaks in Python can be challenging but there are several tools and techniques available to help identify and resolve these issues. Here are some of the most effective tools and methods for diagnosing memory leaks in Python –

### 1. Using the "gc" Module

The gc module can help in identifying objects that are not being collected by the garbage collector. Following is the example of diagnosing the memory leaks using the gc module –

```

import gc

# Enable automatic garbage collection
gc.enable()

# Collect garbage and return unreachable objects
unreachable_objects = gc.collect()
print(f"Unreachable objects: {unreachable_objects}")

# Get a list of all objects tracked by the garbage collector
all_objects = gc.get_objects()
print(f"Number of tracked objects: {len(all_objects)}")

```

#### Output

```

Unreachable objects: 51
Number of tracked objects: 6117

```

### 2. Using "tracemalloc"

The tracemalloc module is used to trace memory allocations in Python. It is helpful for tracking memory usage and identifying where memory is being allocated. Following is the example of diagnosing the memory leaks using the tracemalloc module –

```

import tracemalloc

# Start tracing memory allocations
tracemalloc.start()

# our code here
a = 10
b = 20
c = a+b

# Take a snapshot of current memory usage
snapshot = tracemalloc.take_snapshot()

# Display the top 10 memory-consuming lines
top_stats = snapshot.statistics('lineno')
for stat in top_stats[:10]:
    print(stat)

```

## Output

```
C:\Users\Niharikaa\Desktop\sample.py:7: size=400 B, count=1, average=400 B
```

### 3. Using "memory\_profiler"

The `memory_profiler` is a module for monitoring memory usage of a Python program. It provides a decorator to profile functions and a command-line tool for line-by-line memory usage analysis. In the below example, we are diagnosing the memory leaks using the `memory_profiler` module –

```

from memory_profiler import profile

@profile
def my_function():

    # our code here
    a = 10
    b = 20
    c = a+b

if __name__ == "__main__":
    my_function()

```

## Output

| Line # | Mem  | usage | Increment | Occurrences | Line               |
|--------|------|-------|-----------|-------------|--------------------|
| <hr/>  |      |       |           |             |                    |
| 3      | 49.1 | MiB   | 49.1 MiB  | 1           | @profile           |
| 4      |      |       |           |             | def my_function(): |
| 5      |      |       |           |             | # Your code here   |
| 6      | 49.1 | MiB   | 0.0 MiB   | 1           | a = 10             |
| 7      | 49.1 | MiB   | 0.0 MiB   | 1           | b = 20             |
| 8      | 49.1 | MiB   | 0.0 MiB   | 1           | c = a+b            |

## Fixing Memory Leaks

Once a memory leak is identified, we can fix the memory leaks which involves locating and eliminating unnecessary references to objects.

- **Eliminate Global Variables:** Avoid using global variables unless and until absolutely necessary. Instead we can use local variables or pass objects as arguments to functions.
- **Break Circular References:** Use weak references to break cycles where possible. The weakref module allows us to create weak references that do not prevent garbage collection.
- **Manual Cleanup:** Explicitly delete objects or remove references when they are no longer needed.
- **Use Context Managers:** Ensure resources that are properly cleaned up using context managers i.e. with statement.
- **Optimize Data Structures:** Use appropriate data structures that do not unnecessarily hold onto references.

Finally, we can conclude diagnosing and fixing memory leaks in Python involves identifying lingering references by using tools like gc, memory\_profiler and tracemalloc etc. to track memory usage and implementing fixes such as removing unnecessary references and breaking circular references.

By following these steps, we can ensure our Python programs use memory efficiently and avoid memory leaks.

# 224. Python - Immutable Data Structures

The Python Immutable data structures once created, cannot be changed. This means that any attempt to modify the data structure will result in a new instance being created rather than altering the original. Immutable data structures are useful for ensuring that data remains constant throughout the execution of a program which can help prevent bugs and make code easier to understand and maintain.

Before proceeding deep into this topic let's have a quick recall of what is data structure. The Data structures are specialized formats for organizing, processing, retrieving and storing data. They define how data is arranged in memory and how operations such as accessing, inserting, deleting and updating can be performed efficiently.

## Different Immutable Data Structures in Python

Immutable data structures are essential in Python for their stability, thread-safety and ease of use. Here are the different immutable data structures in Python –

- **Tuples:** These are the ordered collections of items that cannot be changed after their creation. They can contain mixed data types and are useful for representing fixed collections of related items.
- **Strings:** These data structures are sequences of characters and are immutable. Any operation that modifies a string will create a new string.
- **Frozen Sets:** These are immutable versions of sets. Unlike regular sets, frozen sets do not allow modification after creation.
- **Named Tuples:** These are a subclass of tuples with named fields which provide more readable and self-documenting code. They are immutable like regular tuples.

Now, let's proceed about each Immutable data structure in detail.

### Tuples

Tuples in Python are immutable sequences of elements which means once created, they cannot be modified. They are defined using parentheses '()' and can hold a collection of items such as numbers, strings and even other tuples.

#### Creating Tuples

Tuples are created using parentheses '()' and elements separated by commas ','. Even tuples with a single element require a trailing comma to distinguish them from grouped expressions.

Following is the example of creating a tuple by assigning parentheses '()' to a variable –

```
empty_tuple = ()  
single_element_tuple = (5,) # Note the comma after the single element  
print("Single element tuple:", single_element_tuple)  
multi_element_tuple = (1, 2, 'Tutorialspoint', 3.14)  
print("Multi elements tuple:", multi_element_tuple)  
nested_tuple = (1, (2, 3), 'Learning')
```

```
print("Nested tuple:", nested_tuple)
```

On executing the above code, we will get the following output –

```
Single element tuple: (5,)
Multi elements tuple: (1, 2, 'Tutorialspoint', 3.14)
Nested tuple: (1, (2, 3), 'Learning')
```

### **Understanding Tuple Immutability in Python**

Here we are going understand the immutability of the tuples in Python. Below is the example –

```
# Define a tuple
my_tuple = (1, 2, 3, 'hello')

# Attempt to modify an element (which is not possible with tuples)
# This will raise a TypeError

try:
    my_tuple[0] = 10
except TypeError as e:
    print(f"Error: {e}")

# Even trying to append or extend a tuple will result in an error
try:
    my_tuple.append(4)
except AttributeError as e:
    print(f"Error: {e}")

# Trying to reassign the entire tuple to a new value is also not allowed
try:
    my_tuple = (4, 5, 6)
except TypeError as e:
    print(f"Error: {e}")

print("Original tuple:", my_tuple)
```

On executing the above code, we will get the following output –

```
Error: 'tuple' object does not support item assignment
Error: 'tuple' object has no attribute 'append'
Original tuple: (4, 5, 6)
```

## **Strings**

Strings in Python are sequences of characters which are used to represent and manipulate textual data. They are enclosed within either single quotes ' or double quotes " with the option to use triple quotes """ for multi-line strings.

Key characteristics include immutability which means once created those strings cannot be changed, ordered indexing where characters are accessed by position and support for various operations such as concatenation, slicing and iteration.

Strings are fundamental in Python for tasks such as text processing, input/output operations and data representation in applications offering a versatile toolset with built-in methods for efficient manipulation and formatting of textual information.

### **Creating Strings**

Each type of string creation method i.e. ', ", """ has its own use case depending on whether we need to include quotes within the string, handle multi-line text or other specific formatting requirements in our Python code.

Following is the example of creating the string with the help of three types of quotes ', ", """ –

```
# Single line string
single_quoted_string = 'Hello, Welcome to Tutorialspoint'

# Double quoted string
double_quoted_string = "Python Programming"

# Triple quoted string for multi-line strings
multi_line_string = """This is a
multi-line
string"""

print(single_quoted_string)
print(double_quoted_string)
print(multi_line_string)
```

On executing the above code, we will get the following output –

```
Hello, Welcome to Tutorialspoint
Python Programming
This is a
multi-line
string
```

### **Understanding String Immutability in Python**

With the help of following example we are going to understand the immutability of the strings in python.

```
# Example demonstrating string immutability

my_string = "Hello"

# Attempting to modify a string will create a new string instead of modifying
# the original

modified_string = my_string + " Learners"
print(modified_string) # Output: Hello Learners

# Original string remains unchanged
print(my_string) # Output: Hello

# Trying to modify the string directly will raise an error
try:
    my_string[0] = 'h' # TypeError: 'str' object does not support item
    assignment
except TypeError as e:
    print(f"Error: {e}")
```

On executing the above code, we will get the following output –

```
Hello Learners
Hello
Error: 'str' object does not support item assignment
```

## Frozen Sets

A frozen set in Python is an immutable version of a set. Once created its elements cannot be changed, added or removed. Frozen sets are particularly useful in situations where we need a set that remains constant throughout the execution of a program especially when we want to use it as a key in a dictionary or as an element in another set.

### Creating Frozen Sets

We can create a frozen set using the `frozenset()` constructor by passing an iterable such as a list or another set as an argument. Following is the example of creating a frozen set –

```
# Creating a frozen set
fset = frozenset([1, 2, 3, 4])

# Printing the frozen set
print(fset)
```

On executing the above code, we will get the following output –

```
frozenset({1, 2, 3, 4})
```

### **Understanding Frozen Sets Immutability in Python**

Here's an example shows how frozen sets become immutable and do not allow modifications after creation.

```
# Creating a frozenset
frozen_set = frozenset([1, 2, 3, 4])

# Attempting to add an element to the frozenset will raise an error
try:
    frozen_set.add(5)
except AttributeError as e:
    print(f"Error: {e}")

# Attempting to remove an element from the frozenset will also raise an error
try:
    frozen_set.remove(2)
except AttributeError as e:
    print(f"Error: {e}")

# The original frozenset remains unchanged
print("Original frozenset:", frozen_set)
```

On executing the above code, we will get the following output –

```
Error: 'frozenset' object has no attribute 'add'
Error: 'frozenset' object has no attribute 'remove'
Original frozenset: frozenset({1, 2, 3, 4})
```

### **Named Tuples**

A named tuple in Python is a lightweight data structure available in the collections module that behaves same as a tuple but allows us to access its elements using named attributes as well as indices.

It combines the advantages of tuples such as immutable, memory-efficient with the ability to refer to elements by name, enhancing readability and maintainability of code.

#### **Creating Named Tuples**

We can define a named tuple using the namedtuple() factory function from the collections module. It takes two arguments such as a name for the named tuple type and a sequence i.e. string of field names or iterable of strings which specifies the names of its fields.

```

from collections import namedtuple

# Define a named tuple type 'Point' with fields 'x' and 'y'
Point = namedtuple('Point', ['x', 'y'])

# Create an instance of Point
p1 = Point(1, 2)

# Access elements by index (like a tuple)
print(p1[0])

# Access elements by name
print(p1.x)
print(p1.y)

```

On executing the above code, we will get the following output –

```

1
1
2

```

### **Understanding Named Tuples Immutability in Python**

The Named tuples in Python are provided by the `collections.namedtuple` factory functions are indeed immutable. They behave similarly to regular tuples but have named fields by making them more readable and self-documenting.

```

from collections import namedtuple

# Define a named tuple called Point with fields 'x' and 'y'
Point = namedtuple('Point', ['x', 'y'])

# Create an instance of Point
p = Point(x=1, y=2)
print(p)

# Attempt to modify the named tuple
# This will raise an AttributeError since named tuples are immutable
try:
    p.x = 10
except AttributeError as e:

```

```
print(f"Error occurred: {e}")

# Accessing elements in a named tuple is similar to accessing elements in a
regular tuple

print(p.x)
print(p.y)
```

On executing the above code, we will get the following output –

```
Point(x=1, y=2)
Error occurred: can't set attribute
1
2
```