What is Python

Python is a general purpose, dynamic, high-level, and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

Python is *easy to learn* yet powerful and versatile scripting language, which makes it attractive for Application Development.

Python's syntax and *dynamic typing* with its interpreted nature make it an ideal language for scripting and rapid application development.

Python supports *multiple programming pattern*, including object-oriented, imperative, and functional or procedural programming styles.

Python is not intended to work in a particular area, such as web programming. That is why it is known as *multipurpose* programming language because it can be used with web, enterprise, 3D CAD, etc.

We don't need to use data types to declare variable because it is *dynamically typed* so we can write a=10 to assign an integer value in an integer variable.

Python makes the development and debugging *fast* because there is no compilation step included in Python development, and edit-test-debug cycle is very fast.

Python 2 vs. Python 3

In most of the programming languages, whenever a new version releases, it supports the features and syntax of the existing version of the language, therefore, it is easier for the projects to switch in the newer version. However, in the case of Python, the two versions Python 2 and Python 3 are very much different from each other.

A list of differences between Python 2 and Python 3 are given below:

1. Python 2 uses **print** as a statement and used as print "something" to print some string on the console. On the other hand, Python 3 uses **print** as a function and used as print("something") to print something on the console.

2. Python 2 uses the function raw_input() to accept the user's input. It returns the string representing the value, which is typed by the user. To convert it into the integer, we need to use the int() function in Python. On the other hand, Python 3 uses input() function which automatically interpreted the type of input entered by the user. However, we can cast this value to any type by using primitive functions (int(), str(), etc.).
3. In Python 2, the implicit string type is ASCII, whereas, in Python 3, the implicit string type is Unicode.
4. Python 3 doesn't contain the xrange() function of Python 2. The xrange() is the variant of range() function which returns a xrange object that works similar to Java iterator. The range() returns a list for example the function range(0,3) contains 0, 1, 2.
5. There is also a small change made in Exception handling in Python 3. It defines a keyword **as** which is necessary to be used. We will discuss it in Exception handling section of Python programming tutorial.

Python Features

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

1) Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

2) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Java or C takes multiple lines.

3) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

5) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

6) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

8) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

10) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

11. Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

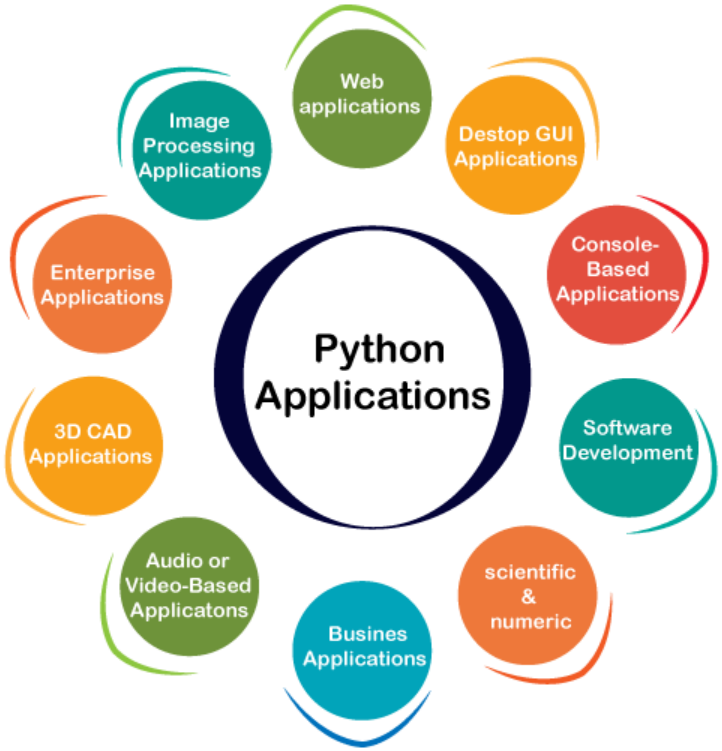
12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x,** then we don't need to write **int x = 15.** Just write x = 15.

Python Applications

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Here, we are specifying application areas where Python can be applied.



1) Web Applications

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on **Instagram**. Python provides many useful frameworks, and these are given below:

- Django and Pyramid framework(Use for heavy applications)
- Flask and Bottle (Micro-framework)
- Plone and Django CMS (Advance Content management)

2) Desktop GUI Applications

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a **Tk GUI library** to develop a user interface. Some popular GUI libraries are given below.

- Tkinter or Tk
- wxWidgetM
- Kivy (used for writing multitouch applications)
- PyQt or Pyside

3) Console-based Application

Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. This kind of application was more popular in the old generation of computers. Python can develop this kind of application very effectively. It is famous for having REPL, which means **the Read-Eval-Print Loop** that makes it the most suitable language for the command-line applications.

Python provides many free library or module which helps to build the command-line apps. The necessary **IO** libraries are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

4) Software Development

Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

- **SCons** is used to build control.
- **Buildbot** and **Apache** Gumps are used for automated continuous compilation and testing.
- **Round** or **Trac** for bug tracking and project management.

5) Scientific and Numeric

This is the era of Artificial intelligence where the machine can perform the task the same as the human. Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

Implementing machine learning algorithms require complex mathematical calculation. Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, etc. If you have some basic knowledge of Python, you need to import libraries on the top of the code. Few popular frameworks of machine libraries are given below.

- SciPy
- Scikit-learn
- NumPy
- Pandas
- Matplotlib

6) Business Applications

Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.

Oddo is an example of the all-in-one Python-based application which offers a range of business applications. Python provides a **Tryton** platform which is used to develop the business application.

7) Audio or Video-based Applications

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are **TimPlayer, cplay,** etc. The few multimedia libraries are given below.

- Gstreamer
- Pyglet
- QT Phonon

8) 3D CAD Applications

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using the following functionalities.

- Fandango (Popular)
- CAMVOX
- HeeksCNC
- AnyCAD
- RCAM

9) Enterprise Applications

Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

10) Image Processing Application

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are given below.

- OpenCV
- Pillow
- SimpleITK

In this topic, we have described all types of applications where Python plays an essential role in the development of these applications. In the next tutorial, we will learn more concepts about Python.

Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.

Identifier Naming

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
- Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- Identifier name must not be similar to any keyword defined in the language.
- Identifier names are case sensitive; for example, my name, and MyName is not the same.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Object References

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class. Consider the following example.

```
1. print("John")
```

Output:

```
John
```

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type()** function.

```
1. type("John")
```

Output:

```
<class 'str'>
```

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

Let's understand the following example

1. a = 50

In the above image, the variable **a** refers to an integer object.

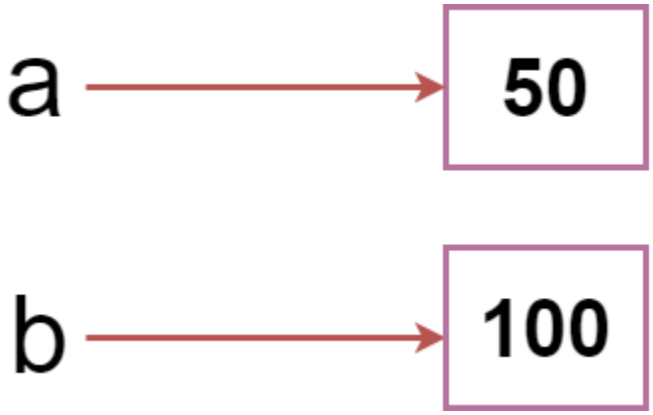
Suppose we assign the integer value 50 to a new variable b.

```
a = 50

b = a
```

The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

```
a = 50

b =100
```



Python manages memory efficiently if we assign the same variable to two different values.

Object Identity

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in **id()** function, is used to identify the object identifier. Consider the following example.

```
1. a = 50
2. b = a
3. print(id(a))
4. print(id(b))
5. # Reassigned variable a
6. a = 500
7. print(id(a))
```

Output:

```
140734982691168
140734982691168
2822056960944
```

We assigned the **b = a, a** and **b** both point to the same object. When we checked by the **id()** function it returned the same number. We reassign **a** to 500; then it referred to the new object identifier.

Variable Names

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character(_). Consider the following example of valid variables names.

```
1. name = "Devansh"
2. age = 20
3. marks = 80.50
4.
5. print(name)
6. print(age)
7. print(marks)
```

Output:

```
Devansh
20
80.5
```

Consider the following valid variables name.

```
1. name = "A"
2. Name = "B"
3. naMe = "C"
4. NAME = "D"
5. n_a_m_e = "E"
6. _name = "F"
7. name_ = "G"
```

```
_name_ = "H"
na56me = "I"

print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)
```

Output:

```
A B C D E D E F G F I
```

In the above example, we have declared a few valid variable names such as name, _name_ , etc. But it is not recommended because when we try to read code, it may create confusion. The variable name should be descriptive to make code more readable.

The multi-word keywords can be created by the following method.

- **Camel Case -** In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
- **Pascal Case -** It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
- **Snake Case -** In the snake case, Words are separated by the underscore. For example - name_of_student, etc.

Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

1. Assigning single value to multiple variables

Eg:

```
x=y=z=50
```

2. print(x)
3. print(y)
4. print(z)

Output:

```
50
50
50
```

2. Assigning multiple values to multiple variables:

Eg:

```
1. a,b,c=5,10,15
2. print a
3. print b
4. print c
```

Output:

```
5
10
15
```

The values will be assigned in the order in which variables appear.

Basic Fundamentals:

This section contains the fundamentals of Python, such as:

i)Tokens and their types.

ii) Comments

a)Tokens:

- The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
- The token is the smallest unit inside the given program.

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- Operators.

Python Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

1. a = 5

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

```
1. a=10
2. b="Hi Python"
3. c = 10.5
4. print(type(a))
5. print(type(b))
6. print(type(c))
```

Output:

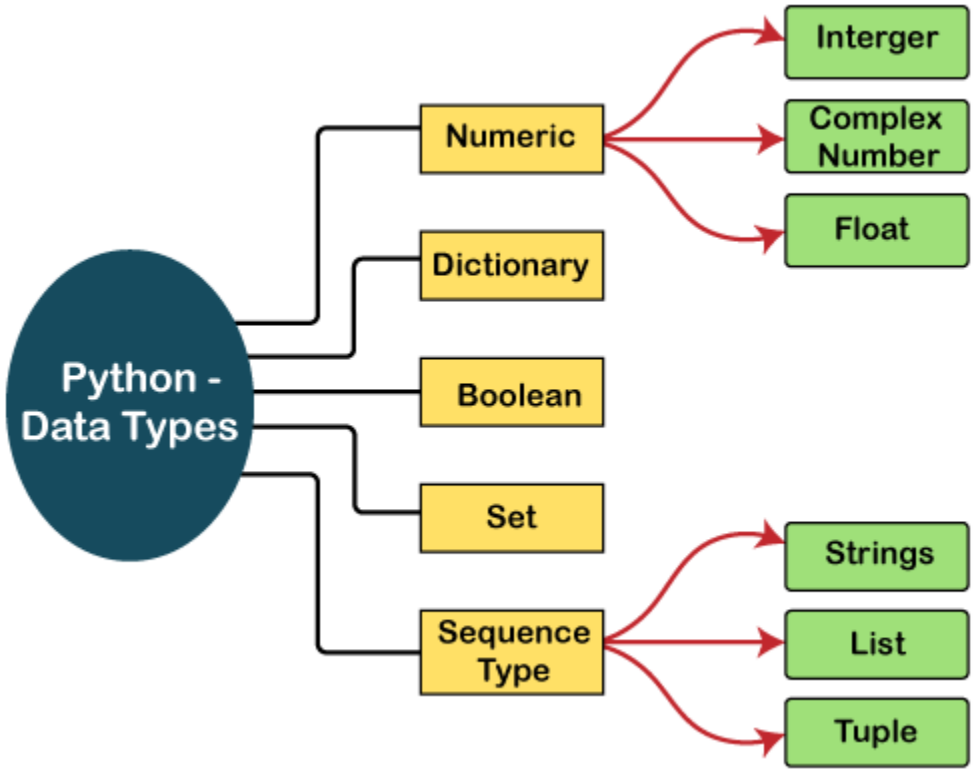
```
<type 'int'>
<type 'str'>
<type 'float'>
```

Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. Numbers
2. Sequence Type
3. Boolean
4. Set
5. Dictionary



In this section of the tutorial, we will give a brief introduction of the above data-types. We will discuss each one of them in detail later in this tutorial.

Numbers

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable. For example;

```
1. a = 5
2. print("The type of a", type(a))
3.
4. b = 40.5
5. print("The type of b", type(b))
6.
7. c = 1+3j
8. print("The type of c", type(c))
9. print(" c is a complex number", isinstance(1+3j,complex))
```

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.

3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Sequence Type

String

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator * is known as a repetition operator as the operation "Python" *2 returns 'Python Python'.

The following example illustrates the string in Python.

Example - 1

```
1. str = "string using double quotes"
2. print(str)
3. s = '''A multiline
4. string'''
5. print(s)
```

Output:

```
string using double quotes
A multiline
string
```

Consider the following example of string handling.

Example - 2

```
str1 = 'hello javatpoint' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

Output:

```
he
o
hello javatpointhello javatpoint
hello javatpoint how are you
```

List

Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
list1  = [1, "hi", "Python", 2]
#Checking type of given list
print(type(list1))

#Printing the list1
```

```
print (list1)

# List slicing
print (list1[3:])

# List slicing
print (list1[0:2])

# List Concatenation using + operator
print (list1 + list1)

# List repetation using * operator
print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```
tup  = ("hi", "Python", 2)
```

```
# Checking type of tup
print (type(tup))

#Printing the tuple
print (tup)

# Tuple slicing
print (tup[1:])
print (tup[0:1])

# Tuple concatenation using + operator
print (tup + tup)

# Tuple repatation using * operator
print (tup * 3)

# Adding value to tup. It will throw an error.
t[2] = "hi"
```

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):
  File "main.py", line 14, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Consider the following example.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}

# Printing dictionary
print (d)

# Accesing value using keys
print("1st name is "+d[1])
print("2nd name is "+ d[4])

print (d.keys())
print (d.values())
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

```
1. # Python program to check the boolean type
2. print(type(True))
3. print(type(False))
4. print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

Set

Python Set is the unordered collection of the data type. It is iterable, mutable(can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function **set(),** or a sequence of elements is passed in the curly braces and separated by the comma. It can contain various types of values. Consider the following example.

```
1. # Creating Empty set
2. set1 = set()
3.
4. set2 = {'James', 2, 3,'Python'}
5.
6. #Printing Set value
7. print(set2)
8.
9. # Adding element to the set
10.
```

```
11. set2.add(10)
12. print(set2)
13. 
14. #Removing element from the set
15. set2.remove(2)
16. print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```

Python Keywords

Python Keywords are special reserved words that convey a special meaning to the compiler/interpreter. Each keyword has a special meaning and a specific operation. These keywords can't be used as a variable. Following is the List of Python Keywords.

True	False	None	and	as
asset	def	class	continue	break
else	finally	elif	del	except
global	for	if	from	import
raise	try	or	return	pass

nonlocal	in	not	is	lambda

Consider the following explanation of keywords.

1. **True -** It represents the Boolean true, if the given condition is true, then it returns "True". Non-zero values are treated as true.
2. **False -** It represents the Boolean false; if the given condition is false, then it returns "False". Zero value is treated as false
3. **None -** It denotes the null value or void. An empty list or Zero can't be treated as **None**.
4. **and -** It is a logical operator. It is used to check the multiple conditions. It returns true if both conditions are true. Consider the following truth table.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

5. or - It is a logical operator in Python. It returns true if one of the conditions is true. Consider the following truth table.

A	B	A and B
True	True	True
True	False	True
False	True	True
False	False	False

6. not - It is a logical operator and inverts the truth value. Consider the following truth table.

A	Not A
True	False
False	True

7. assert - This keyword is used as the debugging tool in Python. It checks the correctness of the code. It raises an **AssertionError** if found any error in the code and also prints the message with an error.

Example:

```
a = 10
b = 0
print('a is dividing by Zero')
assert b != 0
print(a / b)
```

Output:

```
a is dividing by Zero
Runtime Exception:
Traceback (most recent call last):
  File "/home/40545678b342ce3b70beb1224bed345f.py", line 4, in
    assert b != 0, "Divide by 0 error"
AssertionError: Divide by 0 error
```

8. def - This keyword is used to declare the function in Python. If followed by the function name.

```
def my_func(a,b):
    c = a+b
    print(c)
my_func(10,20)
```

Output:

```
30
```

9. class - It is used to represents the class in Python. The class is the blueprint of the objects. It is the collection of the variable and methods. Consider the following class.

```
class Myclass:
  #Variables……..
  def function_name(self):
    #statements……
```

10. continue - It is used to stop the execution of the current iteration. Consider the following example.

```
a = 0
while a < 4:
  a += 1
  if a == 2:
    continue
  print(a)
```

Output:

```
1
3
4
```

11. break - It is used to terminate the loop execution and control transfer to the end of the loop. Consider the following example.

Example

```
for i in range(5):
    if(i==3):
        break
    print(i)
print("End of execution")
```

Output:

```
0
1
2
End of execution
```

12. If - It is used to represent the conditional statement. The execution of a particular block is decided by if statement. Consider the following example.

Example

```
1. i = 18
2. if (1 < 12):
3. print("I am less than 18")
```

Output:

```
I am less than 18
```

13. else - The else statement is used with the if statement. When if statement returns false, then else block is executed. Consider the following example.

Example:

```
1. n = 11
2. if(n%2 == 0):
3.     print("Even")
4. else:
5.     print("odd")
```

Output:

```
Odd
```

14. elif - This Keyword is used to check the multiple conditions. It is short for **else-if**. If the previous condition is false, then check until the true condition is found. Condition the following example.

Example:

```
1. marks = int(input("Enter the marks:"))
```

```
if(marks>=90):
    print("Excellent")
elif(marks<90 and marks>=75):
    print("Very Good")
elif(marks<75 and marks>=60):
    print("Good")
else:
    print("Average")
```

Output:

```
Enter the marks:85
Very Good
```

15. del - It is used to delete the reference of the object. Consider the following example.

Example:

```
a=10
b=12
del a
print(b)
# a is no longer exist
print(a)
```

Output:

```
12
NameError: name 'a' is not defined
```

16. try, except - The try-except is used to handle the exceptions. The exceptions are run-time errors. Consider the following example.

Example:

```
a = 0
try:
    b = 1/a
except Exception as e:
    print(e)
```

Output:

```
division by zero
```

17. raise - The raise keyword is used to through the exception forcefully. Consider the following example.

Example

```
a = 5
if (a>2):
    raise Exception('a should not exceed 2 ')
```

Output:

```
Exception: a should not exceed 2
```

18. finally - The **finally** keyword is used to create a block of code that will always be executed no matter the else block raises an error or not. Consider the following example.

Example:

```
a=0
b=5
try:
    c = b/a
```

```
    print(c)
except Exception as e:
    print(e)
finally:
    print('Finally always executed')
```

Output:

```
division by zero
Finally always executed
```

19. for, while - Both keywords are used for iteration. The **for** keyword is used to iterate over the sequences (list, tuple, dictionary, string). A while loop is executed until the condition returns false. Consider the following example.

Example: For loop

```
list = [1,2,3,4,5]
for i in list:
    print(i)
```

Output:

```
1
2
3
4
5
```

Example: While loop

```
a = 0
while(a<5):
    print(a)
    a = a+1
```

Output:

```
0
1
2
3
4
```

20. import - The import keyword is used to import modules in the current Python script. The module contains a runnable Python code.

Example:

```
import math
print(math.sqrt(25))
```

Output:

```
5
```

21. from - This keyword is used to import the specific function or attributes in the current Python script.

Example:

```
from math import sqrt
print(sqrt(25))
```

Output:

```
5
```

22. as - It is used to create a name alias. It provides the user-define name while importing a module.

Example:

```
import calendar as cal
print(cal.month_name[5])
```

Output:

```
May
```

23. pass - The **pass** keyword is used to execute nothing or create a placeholder for future code. If we declare an empty class or function, it will through an error, so we use the pass keyword to declare an empty class or function.

Example:

```
class my_class:
    pass

def my_func():
    pass
```

24. return - The **return** keyword is used to return the result value or none to called function.

Example:

```
def sum(a,b):
    c = a+b
    return c

print("The sum is:",sum(25,15))
```

Output:

```
The sum is: 40
```

25. is - This keyword is used to check if the two-variable refers to the same object. It returns the true if they refer to the same object otherwise false. Consider the following example.

Example

```
x = 5
y = 5

a = []
b = []
print(x is y)
print(a is b)
```

Output:

```
True
False
```

Note: A mutable data-types do not refer to the same object.

26. global - The global keyword is used to create a global variable inside the function. Any function can access the global. Consider the following example.

Example

```
def my_func():
    global a
    a = 10
    b = 20
    c = a+b
    print(c)

my_func()
```

```
9.
10. def func():
11.     print(a)
12.
13. func()
```

Output:

```
30
10
```

27. nonlocal - The **nonlocal** is similar to the **global** and used to work with a variable inside the nested function(function inside a function). Consider the following example.

Example

```
1.  def outside_function():
2.      a = 20
3.      def inside_function():
4.          nonlocal a
5.          a = 30
6.          print("Inner function: ",a)
7.      inside_function()
8.      print("Outer function: ",a)
9.  outside_function()
```

Output:

```
Inner function:  50
Outer function:  50
```

28. lambda - The lambda keyword is used to create the anonymous function in Python. It is an inline function without a name. Consider the following example.

Example

```
a = lambda x: x**2
for i in range(1,6):
  print(a(i))
```

Output:

```
1
4
9
16
25
```

29. yield - The **yield** keyword is used with the Python generator. It stops the function's execution and returns value to the caller. Consider the following example.

Example

```
def fun_Generator():
  yield 1
  yield 2
  yield 3


# Driver code to check above generator function
for value in fun_Generator():
  print(value)
```

Output:

```
1
2
3
```

30. with - The **with** keyword is used in the exception handling. It makes code cleaner and more readable. The advantage of using **with**, we don't need to call **close()**. Consider the following example.

Example

```
with open('file_path', 'w') as file:
    file.write('hello world !')
```

31. None - The None keyword is used to define the null value. It is remembered that **None** does not indicate 0, false, or any empty data-types. It is an object of its data type, which is Consider the following example.

Example:

```
def return_none():
  a = 10
  b = 20
  c = a + b

x = return_none()
print(x)
```

Output:

```
None
```

Python Literals

Python Literals can be defined as data that is given in a variable or constant.

Python supports the following literals:

1. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Example:

1. "Aman" , '12345'

Types of Strings:

There are two types of Strings supported in Python:

a) Single-line String- Strings that are terminated within a single-line are known as Single line Strings.

Example:

1. text1='hello'

b) Multi-line String - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

1) Adding black slash at the end of each line.

Example:

```
1. text1='hello\
2. user'
3. print(text1)
```

```
'hellouser'
```

2) Using triple quotation marks:-

Example:

```
1. str2='''welcome
2. to
3. SSSIT'''
4. print str2
```

Output:

```
welcome
to
SSSIT
```

II. Numeric literals:

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

Int(signed integers)	Long(long integers)	float(floating point)	Complex(complex)
Numbers(can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: 3.14j

Example - Numeric Literals

```
x = 0b10100 #Binary Literals
y = 100 #Decimal Literal
z = 0o215 #Octal Literal
u = 0x12d #Hexadecimal Literal

#Float Literal
float_1 = 100.5
float_2 = 1.5e2

#Complex Literal
a = 5+3.14j

print(x, y, z, u)
print(float_1, float_2)
print(a, a.imag, a.real)
```

Output:

```
20 100 141 301
100.5 150.0
(5+3.14j) 3.14 5.0
```

III. Boolean literals:

A Boolean literal can have any of the two values: True or False.

Example - Boolean Literals

```
x = (1 == True)
y = (2 == False)
```

```
3. z = (3 == True)
4. a = True + 10
5. b = False + 10
6.
7. print("x is", x)
8. print("y is", y)
9. print("z is", z)
10. print("a:", a)
11. print("b:", b)
```

Output:

```
x is True
y is False
z is False
a: 11
b: 10
```

IV. Special literals.

Python contains one special literal i.e., **None.**

None is used to specify to that field that is not created. It is also used for the end of lists in Python.

Example - Special Literals

```
1. val1=10
2. val2=None
3. print(val1)
4. print(val2)
```

Output:

```
10
None
```

V. Literal Collections.

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

List:

- List contains items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

Example - List literals

```
1. list=['John',678,20.4,'Peter']
2. list1=[456,'Andrew']
3. print(list)
4. print(list + list1)
```

Output:

```
['John', 678, 20.4, 'Peter']
['John', 678, 20.4, 'Peter', 456, 'Andrew']
```

Dictionary:

- Python dictionary stores the data in the key-value pair.
- It is enclosed by curly-braces {} and each pair is separated by the commas(,).

Example

```
dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}
print(dict)
```

Output:

```
{'name': 'Pater', 'Age': 18, 'Roll_nu': 101}
```

Tuple:

- Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.
- It is enclosed by the parentheses () and each element is separated by the comma(,).

Example

```
tup = (10,20,"Dev",[2,3,4])
print(tup)
```

Output:

```
(10, 20, 'Dev', [2, 3, 4])
```

Set:

- Python set is the collection of the unordered dataset.
- It is enclosed by the {} and each element is separated by the comma(,).

Example: - Set Literals

```
set = {'apple','grapes','guava','papaya'}
print(set)
```

Output:

```
{'guava', 'apple', 'papaya', 'grapes'}
```

Python Operators

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators, which are described as follows.

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), *(multiplication), /(divide), %(reminder), //(floor division), and exponent (**) operators.

Consider the following table for a detailed explanation of arithmetic operators.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10

/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to the second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

Comparison operator

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.

>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

Assignment Operators

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	Description
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.

//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Bitwise Operators

The bitwise operators perform bit by bit operation on the values of the two operands. Consider the following example.

For example,

```
if a = 7
   b = 6
then, binary (a) = 0111
     binary (b) = 0011

hence, a & b = 0011
       a | b = 0111
            a ^ b = 0100
        ~ a = 1000
```

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice

	versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Logical Operators

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression **a** is true, then not (a) will be false and vice versa.

Membership Operators

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

Identity Operators

The identity operators are used to decide whether an element certain class or type.

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both sides do not point to the same object.

Operator Precedence

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in Python is given below.

Operator	Description
**	The exponent operator is given priority over all the others used in the expression.
~ + -	The negation, unary plus, and minus.

* / % //	The multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus, and minus
>> <<	Left shift. and right shift
&	Binary and.
^ |	Binary xor, and or
<= < > >=	Comparison operators (less than, less than equal to, greater than, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Python Comments

Python Comment is an essential tool for the programmers. Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation. A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.

In the other programming language such as C++, It provides the // for single-lined comment and /*.... */ for multiple-lined comment, but Python provides the single-lined Python comment. To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

Let's understand the following example.

```
# This is the print statement
print("Hello Python")
```

Here we have written comment over the print statement using the hash(#). It will not affect our print statement.

Multiline Python Comment

We must use the hash(#) at the beginning of every line of code to apply the multiline Python comment. Consider the following example.

```
# First line of the comment
# Second line of the comment
# Third line of the comment
```

Example:

```
# Variable a holds value 5
# Variable b holds value 10
# Variable c holds sum of a and b
# Print the result
a = 5
b = 10
c = a+b
print("The sum is:", c)
```

Output:

```
The sum is: 15
```

The above code is very readable even the absolute beginners can under that what is happening in each line of the code. This is the advantage of using comments in code.

We can also use the triple quotes (''''''') for multiline comment. The triple quotes are also used to string formatting. Consider the following example.

Docstrings Python Comment

The docstring comment is mostly used in the module, function, class or method. It is a documentation Python string. We will explain the class/method in further tutorials.

Example:

```
def intro():
  """
  This function prints Hello Joseph
  """
  print("Hi Joseph")
intro()
```

Output:

```
Hello Joseph
```

We can check a function's docstring by using the __doc__ attribute.

Generally, four whitespaces are used as the indentation. The amount of indentation depends on user, but it must be consistent throughout that block.

```
def intro():
  """
```

```
3.    This function prints Hello Joseph
4.    """
5.    print("Hello Joseph")
6.  intro.__doc__
```

Output:

```
Output:
'\n    This function prints Hello Joseph\n    '
```

Note: The docstring must be the first thing in the function; otherwise, Python interpreter cannot get the docstring.

Python Indenta

Python indentation uses to define the block of the code. The other programming languages such as C, C++, and Java use curly braces {}, whereas Python uses an indentation. Whitespaces are used as indentation in Python.

Indentation uses at the beginning of the code and ends with the unintended line. That same line indentation defines the block of the code (body of a function, loop, etc.)

Generally, four whitespaces are used as the indentation. The amount of indentation depends on user, but it must be consistent throughout that block.

```
1.  for i in range(5):
2.      print(i)
3.      if(i == 3):
4.          break
```

To indicate a block of code we indented each line of the block by the same whitespaces.

Consider the following example.

```
1.  dn = int(input("Enter the number:"))
```

```
if(n%2 == 0):
    print("Even Number")
else:
    print("Odd Number")

print("Task Complete")
```

Output:

```
Enter the number: 10
Even Number
Task Complete
```

The above code, **if** and **else** are two separate code blocks. Both code blocks are indented four spaces. The print("Task Complete") statement is not indented four whitespaces and it is out of the **if-else** block.

If the indentation is not used properly, then that will result in **IndentationError**.

Python If-else statements

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

Indentation in Python

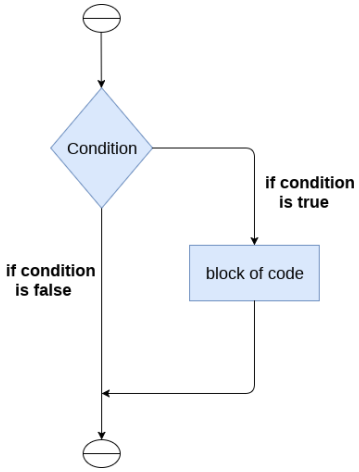
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

```
1. if expression:
2.     statement
```

Example 1

```
1. num = int(input("enter the number?"))
2. if num%2 == 0:
```

```
    print("Number is even")
```

Output:

```
enter the number?10
Number is even
```

Example 2 : Program to print the largest of the three numbers.

```
a = int(input("Enter a? "));
b = int(input("Enter b? "));
c = int(input("Enter c? "));
if a>b and a>c:
    print("a is largest");
if b>a and b>c:
    print("b is largest");
if c>a and c>b:
    print("c is largest");
```

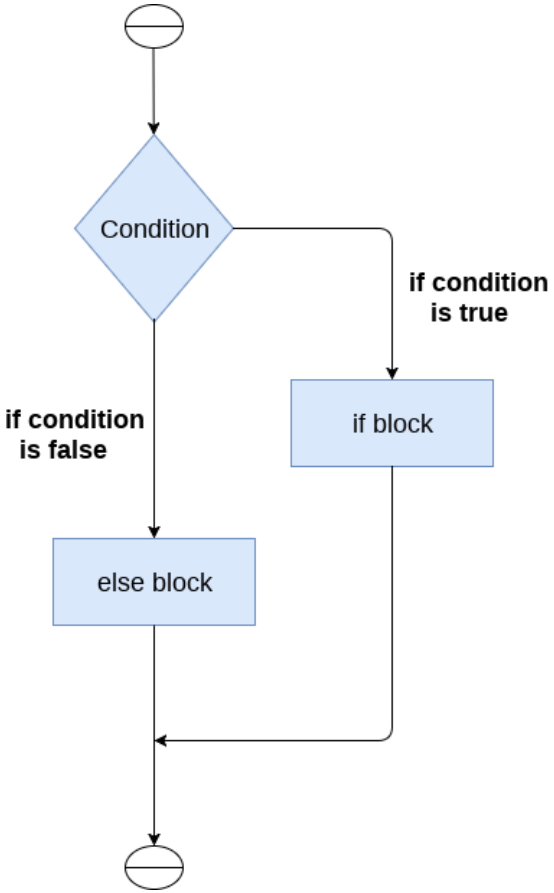
Output:

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```

Example 1 : Program to check whether a person is eligible to vote or not.

```
1. age = int (input("Enter your age? "))
2. if age>=18:
3.     print("You are eligible to vote !!");
4. else:
5.     print("Sorry! you have to wait !!");
```

Output:

```
Enter your age? 90
You are eligible to vote !!
```

Example 2: Program to check whether a number is even or not.

```
1. num = int(input("enter the number?"))
2. if num%2 == 0:
3.     print("Number is even...")
4. else:
5.     print("Number is odd...")
```

Output:

```
enter the number?10
Number is even
```

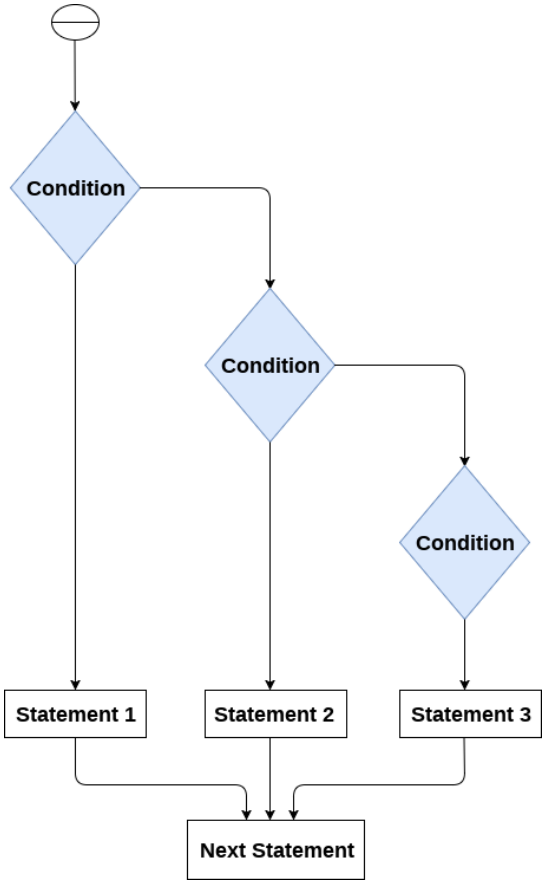
The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below.

```
1. if expression 1:
2.     # block of statements
3.
4. elif expression 2:
5.     # block of statements
6.
7. elif expression 3:
8.     # block of statements
9.
10. else:
11.     # block of statements
```



Example 1

```
1. number = int(input("Enter the number?"))
2. if number==10:
3.     print("number is equals to 10")
4. elif number==50:
5.     print("number is equal to 50");
```

```
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

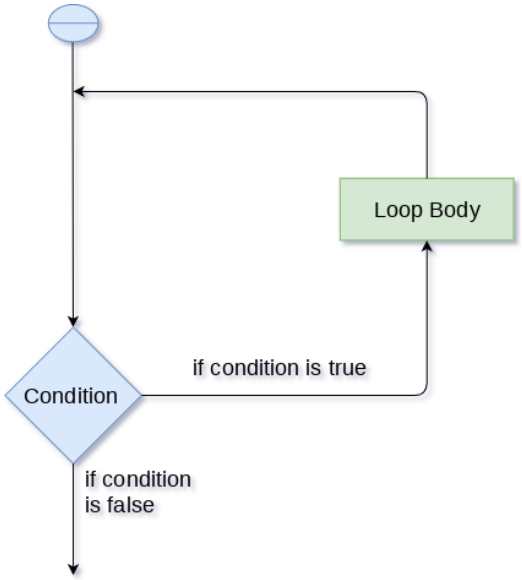
Example 2

```
marks = int(input("Enter the marks? "))
f marks > 85 and marks <= 100:
   print("Congrats ! you scored grade A ...")
lif marks > 60 and marks <= 85:
   print("You scored grade B + ...")
lif marks > 40 and marks <= 60:
   print("You scored grade B ...")
lif (marks > 30 and marks <= 40):
   print("You scored grade C ...")
else:
   print("Sorry you are fail ?")
```

Python Loops

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantages of loops

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Using loops, we do not need to write the same code again and again.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

Loop Statement	Description
for loop	The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
do-while loop	The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

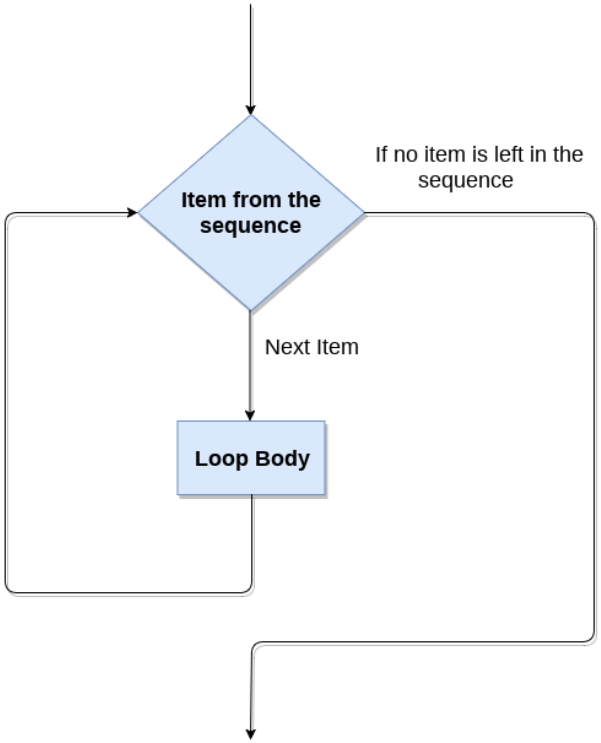
Python for loop

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

```
1. for iterating_var in sequence:
2.     statement(s)
```

The for loop flowchart



For loop Using Sequence

Example-1: Iterating string using for loop

```
1. str = "Python"
2. for i in str:
3.     print(i)
```

Output:

```
P
y
t
h
o
n
```

Example- 2: Program to print the table of the given number .

```
1. list = [1,2,3,4,5,6,7,8,9,10]
2. n = 5
3. for i in list:
4.     c = n*i
5.     print(c)
```

Output:

```
5
10
15
20
25
30
35
40
```

```
45
50s
```

Example-4: Program to print the sum of the given list.

```
1. list = [10,30,23,43,65,12]
2. sum = 0
3. for i in list:
4.     sum = sum+i
5. print("The sum is:",sum)
```

Output:

```
The sum is: 183
```

For loop Using range() function

The range() function

The **range()** function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9. The syntax of the range() function is given below.

Syntax:

```
1. range(start,stop,step size)
```

- The start represents the beginning of the iteration.
- The stop represents that the loop will iterate till stop-1. The **range(1,5)** will generate numbers 1 to 4 iterations. It is optional.
- The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

Consider the following examples:

Example-1: Program to print numbers in sequence.

```
for i in range(10):
    print(i,end = ' ')
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

Example - 2: Program to print table of given number.

```
n = int(input("Enter the number "))
for i in range(1,11):
    c = n*i
    print(n,"*",i,"=",c)
```

Output:

```
Enter the number 10
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

Example-3: Program to print even number using step size in range().

```
n = int(input("Enter the number "))
for i in range(2,n,2):
    print(i)
```

Output:

```
Enter the number 20
2
4
6
8
10
12
14
16
18
```

We can also use the **range()** function with sequence of numbers. The **len()** function is combined with range() function which iterate through a sequence using indexing. Consider the following example.

```
1. list = ['Peter','Joseph','Ricky','Devansh']
2. for i in range(len(list)):
3.     print("Hello",list[i])
```

Output:

```
Hello Peter
Hello Joseph
Hello Ricky
Hello Devansh
```

Nested for loop in python

Python allows us to nest any number of for loops inside a **for** loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

Syntax

```
1. for iterating_var1 in sequence:   #outer loop
```

```
    for iterating_var2 in sequence:  #inner loop
        #block of statements
#Other statements
```

Example- 1: Nested for loop

```
# User input for number of rows
rows = int(input("Enter the rows:"))
# Outer loop will print number of rows
for i in range(0,rows+1):
# Inner loop will print number of Astrisk
    for j in range(i):
        print("*",end = '')
    print()
```

Output:

```
Enter the rows:5
*
**
***
****
*****
```

Example-2: Program to number pyramid.

```
rows = int(input("Enter the rows"))
for i in range(0,rows+1):
    for j in range(i):
        print(i,end = '')
    print()
```

Output:

```
1
22
333
4444
55555
```

Using else statement with for loop

Unlike other languages like C, C++, or Java, Python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

Example 1

```
for i in range(0,5):
    print(i)
else:
    print("for loop completely exhausted, since there is no break.")
```

Output:

```
0
1
2
3
4
for loop completely exhausted, since there is no break.
```

The for loop completely exhausted, since there is no break.

Example 2

```
for i in range(0,5):
    print(i)
    break;
else:
     print("for loop is exhausted");
print("The loop is broken due to break statement...came out of the loop")
```

In the above example, the loop is broken due to the break statement; therefore, the else statement will not be executed. The statement present immediate next to else block will be executed.

Output:

```
0
```

Python While loop

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.

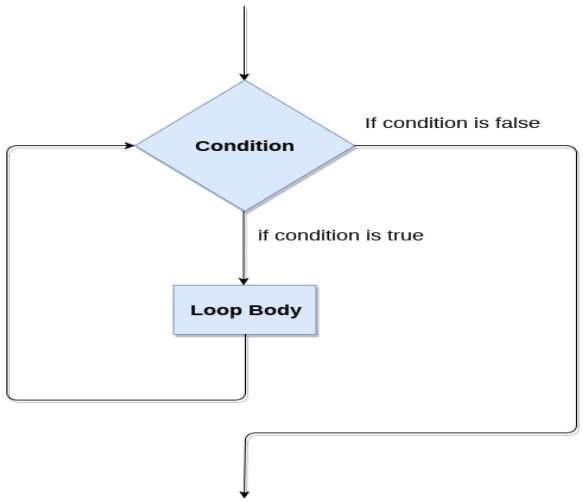
It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

The syntax is given below.

```
1. while expression:
2.     statements
```

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

While loop Flowchart



Example-1: Program to print 1 to 10 using while loop

```
i=1
#The while loop will iterate until condition becomes false.
While(i<=10):
    print(i)
    i=i+1
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Example -2: Program to print table of given numbers.

```
i=1
number=0
b=9
number = int(input("Enter the number:"))
while i<=10:
    print("%d X %d = %d \n"%(number,i,number*i))
    i = i+1
```

Output:

```
Enter the number:10
10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100
```

Infinite while loop

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the **infinite while loop.**

Any **non-zero** value in the while loop indicates an **always-true** condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Example 1

```
1. while (1):
2.     print("Hi! we are inside the infinite while loop")
```

Output:

```
Hi! we are inside the infinite while loop
Hi! we are inside the infinite while loop
```

Example 2

```
var = 1
while(var != 2):
    i = int(input("Enter the number:"))
    print("Entered value is %d"%(i))
```

Output:

```
Enter the number:10
Entered value is 10
Enter the number:10
Entered value is 10
Enter the number:10
Entered value is 10
Infinite time
```

Using else with while loop

Python allows us to use the else statement with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed, and the statement present after else block will be executed. The else statement is optional to use with the while loop. Consider the following example.

Example 1

```
i=1
while(i<=5):
    print(i)
    i=i+1
```

```
else:
    print("The while loop exhausted")
```

Example 2

```
i=1
while(i<=5):
    print(i)
    i=i+1
    if(i==3):
        break
else:
    print("The while loop exhausted")
```

Output:

```
1
2
```

In the above code, when the break statement encountered, then while loop stopped its execution and skipped the else statement.

Example-3 Program to print Fibonacci numbers to given limit

```
terms = int(input("Enter the terms "))
# first two intial terms
a = 0
b = 1
count = 0

# check if the number of terms is Zero or negative
```

```
8.  if (terms <= 0):
9.     print("Please enter a valid integer")
10. elif (terms == 1):
11.    print("Fibonacci sequence upto",limit,":")
12.    print(a)
13. else:
14.    print("Fibonacci sequence:")
15.    while (count < terms) :
16.        print(a, end = ' ')
17.        c = a + b
18.        # updateing values
19.        a = b
20.        b = c
21.
22.     count += 1
```

Output:

```
Enter the terms 10
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34
```

Python break statement

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break is given below.

```
#loop statements
break;
```

Example 1

```
list =[1,2,3,4]
count = 1;
for i in list:
    if i == 4:
        print("item matched")
        count = count + 1;
        break
print("found at",count,"location");
```

Output:

```
item matched
found at 2 location
```

Example 2

```
str = "python"
for i in str:
    if i == 'o':
        break
    print(i);
```

Output:

```
p
y
t
h
```

Example 3: break statement with while loop

```
i = 0;
while 1:
    print(i," ",end=""),
    i=i+1;
    if i == 10:
        break;
print("came out of while loop");
```

Output:

```
0  1  2  3  4  5  6  7  8  9  came out of while loop
```

Example 3

```
n=2
while 1:
```

```
3.    i=1;
4.    while i<=10:
5.        print("%d X %d = %d\n"%(n,i,n*i));
6.        i = i+1;
7.    choice = int(input("Do you want to continue printing the table, press 0 for no?"))
8.    if choice == 0:
9.        break;
10.   n=n+1
```

Output:

```
2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1

3 X 1 = 3

3 X 2 = 6
```

```
3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0
```

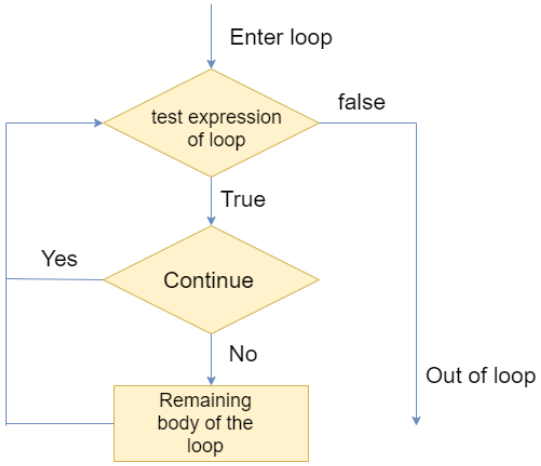
Python continue Statement

The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

Syntax

```
1. #loop statements
2. continue
3. #the code to be skipped
```

Flow Diagram



Consider the following examples.

Example 1

```
1. i = 0
2. while(i < 10):
3.     i = i+1
4.     if(i == 5):
5.         continue
6.     print(i)
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Observe the output of above code, the value 5 is skipped because we have provided the **if condition** using with **continue statement** in while loop. When it matched with the given condition then control transferred to the beginning of the while loop and it skipped the value 5 from the code.

Let's have a look at another example:

Example 2

```
1. str = "JavaTpoint"
2. for i in str:
```

```
3.      if(i == 'T'):
4.          continue
5.      print(i)
```

Output:

```
J
a
v
a
p
o
i
n
t
```

Pass Statement

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.

For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.Pass is also used where the code will be written somewhere but not yet written in the program file. Consider the following example.

Example

```
1. list = [1,2,3,4,5]
2. flag = 0
3. for i in list:
4.     print("Current element:",i,end=" ");
5.     if i==3:
6.         pass
7.         print("\nWe are inside pass block\n");
8.         flag = 1
9.     if flag==1:
10.        print("\nCame out of pass\n");
11.        flag=0
```

Output:

```
Current element: 1 Current element: 2 Current element: 3
We are inside pass block


Came out of pass

Current element: 4 Current element: 5
```

Python Pass

In Python, the pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty. It is the same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.

It is beneficial when a statement is required syntactically, but we want we don't want to execute or execute it later. The difference between the comments and pass is that, comments are entirely ignored by the Python interpreter, where the pass statement is not ignored.

Suppose we have a loop, and we do not want to execute right this moment, but we will execute in the future. Here we can use the pass.

Consider the following example.

Example - Pass statement

```
# pass is just a placeholder for
# we will adde functionality later.
values = {'P', 'y', 't', 'h','o','n'}
for val in values:
    pass
```

Example - 2:

```
for i in [1,2,3,4,5]:
    if(i==4):
        pass
        print("This is pass block",i)
    print(i)
```

Output:

```
1
2
3
This is pass block 4
4
5
```

We can create empty class or function using the pass statement.

```
# Empty Function
def function_name(args):
    pass
#Empty Class
class Python:
    pass
```

Python String

Till now, we have discussed numbers as the standard data-types in Python. In this section of the tutorial, we will discuss the most popular data type in Python, i.e., string.

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.

Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in Python to create a string.

Syntax:

1. str = "Hi Python !"

Here, if we check the type of the variable **str** using a Python script

1. **print**(type(str)), then it will **print** a string (str).

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or **docstrings**.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
str3 = '''Triple quotes are generally used for
        represent the multiline or
        docstring'''
      print(str3)
```

Output:

```
Hello Python
Hello Python
Triple quotes are generally used for
    represent the multiline or
    docstring
```

Strings indexing and splitting

Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.

str = "HELLO"

H	E	L	L	O
0	1	2	3	4

str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

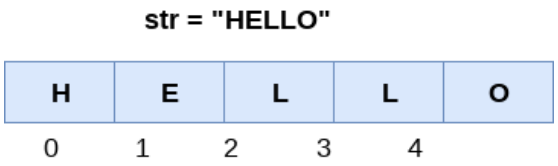
Consider the following example:

```
1. str = "HELLO"
2. print(str[0])
3. print(str[1])
4. print(str[2])
5. print(str[3])
6. print(str[4])
7. # It returns the IndexError because 6th index doesn't exist
8. print(str[6])
```

Output:

```
H
E
L
L
O
IndexError: string index out of range
```

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the : (colon) operator in Python to access the substring from the given string. Consider the following example.



str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

str[:] = 'HELLO'

str[0:] = 'HELLO'

str[:5] = 'HELLO'

str[:3] = 'HEL'

str[0:2] = 'HE'

str[1:4] = 'ELL'

Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.

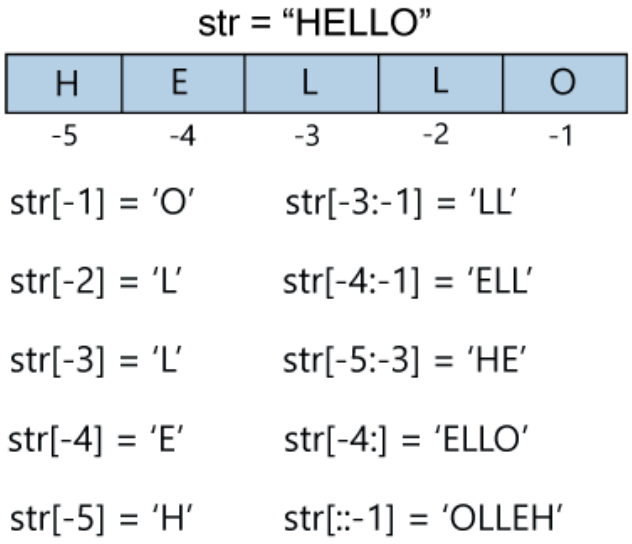
Consider the following example:

```
# Given String
str = "JAVATPOINT"
# Start Oth index to end
print(str[0:])
# Starts 1th index to 4th index
print(str[1:5])
# Starts 2nd index to 3rd index
print(str[2:4])
# Starts 0th to 2nd index
        print(str[:3])
        #Starts 4th to 6th index
        print(str[4:7])
```

Output:

```
JAVATPOINT
AVAT
VA
JAV
TPO
```

We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on. Consider the following image.



Consider the following example

```
1. str = 'JAVATPOINT'
2. print(str[-1])
3. print(str[-3])
4. print(str[-2:])
5. print(str[-4:-1])
6. print(str[-7:-2])
7. # Reversing the given string
8. print(str[::-1])
9. print(str[-12])
```

Output:

```
T
I
```

```
NT
OIN
ATPOI
TNIOPTAVAJ
IndexError: string index out of range
```

Reassigning Strings

Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. Strings are immutable in Python.

Consider the following example.

Example 1

```
1. str = "HELLO"
2. str[0] = "h"
3. print(str)
```

Output:

```
Traceback (most recent call last):
  File "12.py", line 2, in <module>
    str[0] = "h";
TypeError: 'str' object does not support item assignment
```

However, in example 1, the string **str** can be assigned completely to a new content as specified in the following example.

Example 2

```
1. str = "HELLO"
2. print(str)
3. str = "hello"
```

4. **print**(str)

Output:

```
HELLO
hello
```

Deleting the String

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the **del** keyword.

1. str = "JAVATPOINT"
2. **del** str[1]

Output:

```
TypeError: 'str' object doesn't support item deletion
```

Now we are deleting entire string.

1. str1 = "JAVATPOINT"
2. **del** str1
3. **print**(str1)

Output:

```
NameError: name 'str1' is not defined
```

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

Example

Consider the following example to understand the real use of Python operators.

```
1. str = "Hello"
2. str1 = " world"
```

```
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
print(r'C://python37') # prints C://python37 as it is written
print("The string str : %s"%(str)) # prints The string str : Hello
```

Output:

```
HelloHelloHello
Hello world
o
ll
False
False
C://python37
The string str : Hello
```

Python String Formatting

Escape Sequence

Let's suppose we need to write the text as - They said, "Hello what's going on?"- the given statement can be written in single quotes or double quotes but it will raise the **SyntaxError** as it contains both single and double-quotes.

Example

Consider the following example to understand the real use of Python operators.

```
str = "They said, "Hello what's going on?""
```

2. print(str)

Output:

```
SyntaxError: invalid syntax
```

We can use the triple quotes to accomplish this problem but Python provides the escape sequence.

The backslash(/) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently. The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

Example -

```
# using triple quotes
print('''They said, "What's there?"''')

# escaping single quotes
print('They said, "What\'s going on?"')

# escaping double quotes
print("They said, \"What's going on?\"")
```

Output:

```
They said, "What's there?"
They said, "What's going on?"
They said, "What's going on?"
```

The list of an escape sequence is given below:

Sr.	Escape Sequence	Description	Example

1.	\newline	It ignores the new line.	`print("Python1 \`
			`Python2 \`
			`Python3")`
			Output:
			`Python1 Python2 Python3`

2.	\\	Backslash	`print("\\")`
			Output:
			`\`

3.	\'	Single Quotes	`print('\'')`
			Output:
			`'`

4.	\\"	Double Quotes	`print("\"")`
			Output:
			`"`

5.	\a	ASCII Bell	`print("\a")`

6.	\b	ASCII Backspace(BS)	`print("Hello \b World")`
			Output:
			`Hello World`

7.	\f	ASCII Formfeed	`print("Hello \f World!")` `Hello  World!`

8.	\n	ASCII Linefeed	`print("Hello \n World!")`
			Output:
			`Hello`
			`World!`

9.	\r	ASCII Carriege Return(CR)	`print("Hello \r World!")`
			Output:
			`World!`

10.	\t	ASCII Horizontal Tab	`print("Hello \t World!")`
			Output:
			`Hello World!`

11.	\v	ASCII Vertical Tab	`print("Hello \v World!")`
			Output:
			`Hello`
			`World!`

12.	\ooo	Character with octal value	`print("\110\145\154\154\157")`
			Output:
			`Hello`

13	\xHH	Character with hex value.	`print("\x48\x65\x6c\x6c\x6f")`
			Output:
			`Hello`

Here is the simple example of escape sequence.

```
print("C:\\Users\\DEVANSH SHARMA\\Python32\\Lib")
print("This is the \n multiline quotes")
print("This is \x48\x45\x58 representation")
```

Output:

```
C:\Users\DEVANSH SHARMA\Python32\Lib
This is the
 multiline quotes
This is HEX representation
```

We can ignore the escape sequence from the given string by using the raw string. We can do this by writing **r** or **R** in front of the string. Consider the following example.

```
print(r"C:\\Users\\DEVANSH SHARMA\\Python32")
```

Output:

```
C:\\Users\\DEVANSH SHARMA\\Python32
```

The format() method

The **format()** method is the most flexible and useful method in formatting strings. The curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument. Let's have a look at the given an example:

```
# Using Curly braces
print("{} and {} both are the best friend".format("Devansh","Abhishek"))

#Positional Argument
print("{1} and {0} best players ".format("Virat","Rohit"))

#Keyword Argument
```

8. `print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))`

Output:

```
Devansh and Abhishek both are the best friend
Rohit and Virat best players
James,Peter,Ricky
```

Python String Formatting Using % Operator

Python allows us to use the format specifiers used in C's printf statement. The format specifiers in Python are treated in the same way as they are treated in C. However, Python provides an additional operator %, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

Consider the following example.

1. `Integer = 10;`
2. `Float = 1.290`
3. `String = "Devansh"`
4. `print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%(Integer,Float,String))`

Output:

```
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
Hi I am string ... My value is Devansh
```

Python String functions

Python provides various in-built functions that are used for string handling. Many String fun

Method	Description
capitalize()	It capitalizes the first character of the String. This function is deprecated in python3
casefold()	It returns a version of s suitable for case-less comparisons.
center(width ,fillchar)	It returns a space padded string with the original string centred with equal number of left and right spaces.
count(string,begin,end)	It counts the number of occurrences of a substring in a String between begin and end index.
decode(encoding = 'UTF8', errors = 'strict')	Decodes the string using codec registered for encoding.
encode()	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
endswith(suffix ,begin=0,end=len(string))	It returns a Boolean value if the string terminates with given suffix between begin and end.
expandtabs(tabsize = 8)	It defines tabs in string to multiple spaces. The default space value is 8.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
format(value)	It returns a formatted version of S, using the passed value.

index(subsring, beginIndex, endIndex)	It throws an exception if string is not found. It works same as find() method.
isalnum()	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.
isalpha()	It returns true if all the characters are alphabets and there is at least one character, otherwise False.
isdecimal()	It returns true if all the characters of the string are decimals.
isdigit()	It returns true if all the characters are digits and there is at least one character, otherwise False.
isidentifier()	It returns true if the string is the valid identifier.
islower()	It returns true if the characters of a string are in lower case, otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.
isprintable()	It returns true if all the characters of s are printable or s is empty, false otherwise.
isupper()	It returns false if characters of a string are in Upper case, otherwise False.
isspace()	It returns true if the characters of a string are white-space, otherwise false.
istitle()	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are

	lower-case.
isupper()	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
join(seq)	It merges the strings representation of the given sequence.
len(string)	It returns the length of a string.
ljust(width[,fillchar])	It returns the space padded strings with the original string left justified to the given width.
lower()	It converts all the characters of a string to Lower case.
lstrip()	It removes all leading whitespaces of a string and can also be used to remove particular character from leading.
partition()	It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
maketrans()	It returns a translation table to be used in translate function.
replace(old,new[,count])	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
rfind(str,beg=0,end=len(str))	It is similar to find but it traverses the string in backward direction.

rindex(str,beg=0,end=len(str))	It is same as index but it traverses the string in backward direction.
rjust(width,[,fillchar])	Returns a space padded string having original string right justified to the number of characters specified.
rstrip()	It removes all trailing whitespace of a string and can also be used to remove particular character from trailing.
rsplit(sep=None, maxsplit = -1)	It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space.
split(str,num=string.count(str))	Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.
splitlines(num=string.count('\n'))	It returns the list of strings at each line with newline removed.
startswith(str,beg=0,end=len(str))	It returns a Boolean value if the string starts with given str between begin and end.
strip([chars])	It is used to perform lstrip() and rstrip() on the string.
swapcase()	It inverts case of all characters in a string.
title()	It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut.

translate(table,deletechars = '')	It translates the string according to the translation table passed in the function .
upper()	It converts all the characters of a string to Upper Case.
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).
rpartition()	

Python List

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below

```
1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]
```

IIf we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

```
1. print(type(L1))
2. print(type(L2))
```

Output:

```
<class 'list'>
<class 'list'>
```

Characteristics of Lists

The list has the following characteristics:

- The lists are ordered.
- The element of the list can access by index.
- The lists are the mutable type.
- The lists are mutable types.

- A list can store the number of various elements.

Let's check the first statement that lists are the ordered.

```
a = [1,2,"Peter",4.50,"Ricky",5,6]
b = [1,2,5,"Peter",4.50,"Ricky",6]
a ==b
```

Output:

```
False
```

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

```
a = [1, 2,"Peter", 4.50,"Ricky",5, 6]
b = [1, 2,"Peter", 4.50,"Ricky",5, 6]
a == b
```

Output:

```
True
```

Let's have a look at the list example in detail.

```
emp = ["John", 102, "USA"]
Dep1 = ["CS",10]
Dep2 = ["IT",11]
HOD_CS = [10,"Mr. Holding"]
HOD_IT = [11, "Mr. Bewon"]
```

```
print("printing employee data...")
print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
print("printing departments...")
print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))
print("HOD Details ....")
print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]))
print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]))
print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT))
```

Output:

```
printing employee data...
Name : John, ID: 102, Country: USA
printing departments...
Department 1:
Name: CS, ID: 11
Department 2:
Name: IT, ID: 11
HOD Details ....
CS HOD Name: Mr. Holding, Id: 10
IT HOD Name: Mr. Bewon, Id: 11
<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>
```

In the above example, we have created the lists which consist of the employee and department details and printed the corresponding details. Observe the above code to understand the concept of the list better.

List indexing and splitting

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5

List[0] = 0

List[1] = 1

List[2] = 2

List[3] = 3

List[4] = 4

List[5] = 5

List[0:] = [0,1,2,3,4,5]

List[:] = [0,1,2,3,4,5]

List[2:4] = [2, 3]

List[1:3] = [1, 2]

List[:4] = [0, 1, 2, 3]

We can get the sub-list of the list using the following syntax.

1. list_varible(start:stop:step)
    - The **start** denotes the starting index position of the list.
    - The **stop** denotes the last index position of the list.
    - The **step** is used to skip the nth element within a **start:stop**

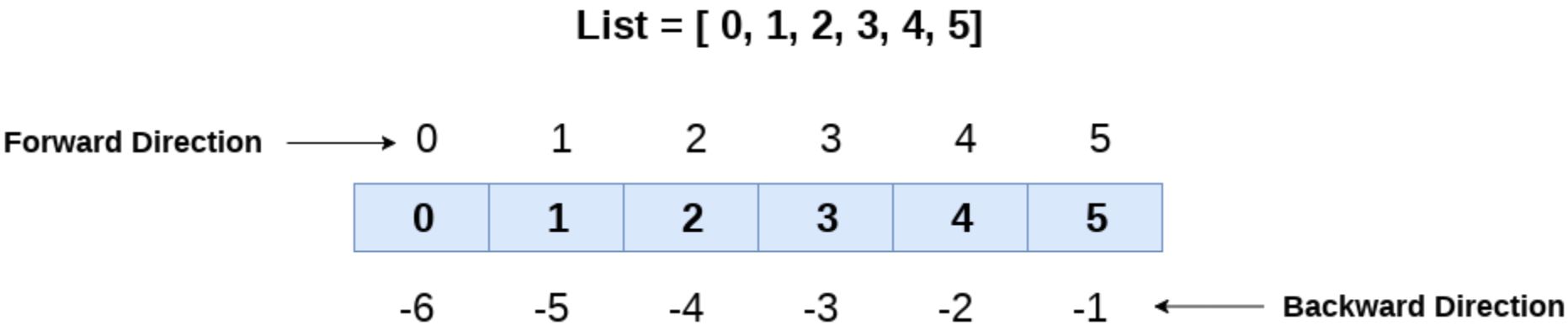
Consider the following example:

```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
# By default the index value is 0 so its starts from the 0th element and go for index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

Output:

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.



Let's have a look at the following example where we will use negative indexing to access the elements of the list.

```
1. list = [1,2,3,4,5]
2. print(list[-1])
3. print(list[-3:])
4. print(list[:-1])
5. print(list[-3:-1])
```

Output:

```
5
[3, 4, 5]
[1, 2, 3, 4]
[3, 4]
```

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

Updating List values

Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the slice and assignment operator.

Python also provides append() and insert() methods, which can be used to add values to the list.

Consider the following example to update the values inside the list.

```
1. list = [1, 2, 3, 4, 5, 6]
2. print(list)
3. # It will assign value to the value to the second index
4. list[2] = 10
5. print(list)
6. # Adding multiple-element
7. list[1:3] = [89, 78]
8. print(list)
9. # It will add value at the end of the list
10. list[-1] = 25
11. print(list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove()** method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

```
1. list = [1, 2, 3, 4, 5, 6]
2. print(list)
3. # It will assign value to the value to second index
4. list[2] = 10
```

```
print(list)
# Adding multiple element
list[1:3] = [89, 78]
print(list)
# It will add value at the end of the list
list[-1] = 25
print(list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

Python List Operations

The concatenation (+) and repetition (*) operators work in the same way as they were working with the strings.

Let's see how the list responds to various operators.

1. Consider a Lists l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8] to perform operation.

Operator	Description	Example
Repetition	The repetition operator enables the list elements to be repeated multiple times.	`L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]`
Concatenation	It concatenates the list mentioned on either side of the operator.	`l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8]`

Membership	It returns true if a particular item exists in a particular list otherwise false.	`print(2 in l1) prints True.`

Iteration	The for loop is used to iterate over the list elements.	`for i in l1:`
		`  print(i)`
		Output
		`1`
		`2`
		`3`
		`4`

Length	It is used to get the length of the list	`len(l1) = 4`

Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

```
1. list = ["John", "David", "James", "Jonathan"]
2. for i in list:
3.     # The i variable will iterate over the elements of the List and contains each element in each iteration.
4.     print(i)
```

Output:

```
John
David
James
Jonathan
```

Adding elements to the list

Python provides append() function which is used to add an element to the list. However, the append() function can only add value to the end of the list.

Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

```
1. #Declaring the empty list
2. l =[]
3. #Number of elements will be entered by the user
4. n = int(input("Enter the number of elements in the list:"))
5. # for loop to take the input
6. for i in range(0,n):
7.     # The input is taken from the user and added to the list as the item
8.     l.append(input("Enter the item:"))
9. print("printing the list items..")
10. # traversal loop to print the list items
11. for i in l:
12.     print(i, end = "  ")
```

Output:

```
Enter the number of elements in the list:5
Enter the item:25
Enter the item:46
Enter the item:12
Enter the item:75
Enter the item:42
printing the list items
25  46  12  75  42
```

Removing elements from the list

Python provides the **remove()** function which is used to remove the element from the list. Consider the following example to understand this concept.

Example -

```
1. list = [0,1,2,3,4]
2. print("printing original list: ");
3. for i in list:
4.     print(i,end=" ")
5. list.remove(2)
6. print("\nprinting the list after the removal of first element...")
7. for i in list:
8.     print(i,end=" ")
```

Output:

```
printing original list:
0 1 2 3 4
printing the list after the removal of first element...
0 1 3 4
```

Python List Built-in functions

Python provides the following built-in functions, which can be used with the lists.

SN	Function	Description	Example
1	cmp(list1,	It compares the elements of both the	This method is not used in the Python 3 and the above

	list2)	lists.	versions.

2	len(list)	It is used to calculate the length of the list.	`L1 = [1,2,3,4,5,6,7,8]`
			`print(len(L1))`
			`8`

3	max(list)	It returns the maximum element of the list.	`L1 = [12,34,26,48,72]`
			`print(max(L1))`
			`72`

4	min(list)	It returns the minimum element of the list.	`L1 = [12,34,26,48,72]`
			`print(min(L1))`
			`12`

5	list(seq)	It converts any sequence to the list.	`str = "Johnson"`
			`s = list(str)`
			`print(type(s))`
			`<class list>`

Let's have a look at the few list examples.

Example: 1- Write the program to remove the duplicate element of the list.

```
list1 = [1,2,2,3,55,98,65,65,13,29]
# Declare an empty list that will store unique values
list2 = []
for i in list1:
    if i not in list2:
```

```
6.          list2.append(i)
7.  print(list2)
```

Output:

```
[1, 2, 3, 55, 98, 65, 13, 29]
```

Example:2- Write a program to find the sum of the element in the list.

```
1.  list1 = [3,4,5,9,10,12,24]
2.  sum = 0
3.  for i in list1:
4.      sum = sum+i
5.  print("The sum is:",sum)
```

Output:

```
The sum is: 67
```

Example: 3- Write the program to find the lists consist of at least one common element.

```
1.  list1 = [1,2,3,4,5,6]
2.  list2 = [7,8,9,2,10]
3.  for x in list1:
4.      for y in list2:
5.          if x == y:
6.              print("The common element is:",x)
```

Output:

```
The common element is: 2
```

Python Tuple

Python Tuple is used to store the sequence of immutable Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

Creating a tuple

A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use. A tuple can be defined as follows.

```
T1 = (101, "Peter", 22)
T2 = ("Apple", "Banana", "Orange")
T3 = 10,20,30,40,50

print(type(T1))
print(type(T2))
print(type(T3))
```

Output:

```
<class 'tuple'>
<class 'tuple'>
<class 'tuple'>
```

Note: The tuple which is created without using parentheses is also known as tuple packing.

An empty tuple can be created as follows.

```
T4 = ()
```

Creating a tuple with single element is slightly different. We will need to put comma after the element to declare the tuple.

```
tup1 = ("JavaTpoint")
print(type(tup1))
#Creating a tuple with single element
tup2 = ("JavaTpoint",)
print(type(tup2))
```

Output:

```
<class 'str'>
<class 'tuple'>
```

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

Consider the following example of tuple:

Example - 1

```
tuple1 = (10, 20, 30, 40, 50, 60)
print(tuple1)
count = 0
for i in tuple1:
    print("tuple1[%d] = %d"%(count, i))
```

```
    count = count+1
```

Output:

```
(10, 20, 30, 40, 50, 60)
tuple1[0] = 10
tuple1[1] = 20
tuple1[2] = 30
tuple1[3] = 40
tuple1[4] = 50
tuple1[5] = 60
```

Example - 2

```
tuple1 = tuple(input("Enter the tuple elements ..."))
print(tuple1)
count = 0
for i in tuple1:
    print("tuple1[%d] = %s"%(count, i))
    count = count+1
```

Output:

```
Enter the tuple elements ...123456
('1', '2', '3', '4', '5', '6')
tuple1[0] = 1
tuple1[1] = 2
tuple1[2] = 3
```

```
tuple1[3] = 4
tuple1[4] = 5
tuple1[5] = 6
```

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

We will see all these aspects of tuple in this section of the tutorial.

Tuple indexing and slicing

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.

Tuple = (0, 1, 2, 3, 4, 5)

0	1	2	3	4	5

Tuple[0] = 0

Tuple[1] = 1

Tuple[2] = 2

Tuple[3] = 3

Tuple[4] = 4

Tuple[5] = 5

Tuple[0:] = (0, 1, 2, 3, 4, 5)

Tuple[:] = (0, 1, 2, 3, 4, 5)

Tuple[2:4] = (2, 3)

Tuple[1:3] = (1, 2)

Tuple[:4] = (0, 1, 2, 3)

Consider the following example:

```
1. tup = (1,2,3,4,5,6,7)
2. print(tup[0])
3. print(tup[1])
4. print(tup[2])
5. # It will give the IndexError
6. print(tup[8])
```

Output:

```
1
2
3
tuple index out of range
```

In the above code, the tuple has 7 elements which denote 0 to 6. We tried to access an element outside of tuple that raised an **IndexError**.

```
1. tuple = (1,2,3,4,5,6,7)
2. #element 1 to end
3. print(tuple[1:])
4. #element 0 to 3 element
5. print(tuple[:4])
6. #element 1 to 4 element
7. print(tuple[1:5])
8. # element 0 to 6 and take step of 2
9. print(tuple[0:6:2])
```

Output:

```
(2, 3, 4, 5, 6, 7)
(1, 2, 3, 4)
(1, 2, 3, 4)
(1, 3, 5)
```

Negative Indexing

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second

last item and so on.

The elements from left to right are traversed using the negative indexing. Consider the following example:

```
1. tuple1 = (1, 2, 3, 4, 5)
2. print(tuple1[-1])
3. print(tuple1[-4])
4. print(tuple1[-3:-1])
5. print(tuple1[:-1])
6. print(tuple1[-2:])
```

Output:

```
5
2
(3, 4)
(1, 2, 3, 4)
(4, 5)
```

Deleting Tuple

Unlike lists, the tuple items cannot be deleted by using the **del** keyword as tuples are immutable. To delete an entire tuple, we can use the **del** keyword with the tuple name.

Consider the following example.

```
1. tuple1 = (1, 2, 3, 4, 5, 6)
2. print(tuple1)
```

```
del tuple1[0]
print(tuple1)
del tuple1
print(tuple1)
```

Output:

```
(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "tuple.py", line 4, in <module>
    print(tuple1)
NameError: name 'tuple1' is not defined
```

Basic Tuple operations

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	`T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)`

Concatenation	It concatenates the tuple mentioned on either side of the operator.	`T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)`
Membership	It returns true if a particular item exists in the tuple otherwise false	`print (2 in T1) prints True.`

Iteration	The for loop is used to iterate over the tuple elements.	`for i in T1:`
		`  print(i)`
		Output
		`1`
		`2`
		`3`
		`4`
		`5`

Length	It is used to get the length of the tuple.	`len(T1) = 5`

Python Tuple inbuilt functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.

2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

Where use tuple?

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.

2. Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.

1. [(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]

List vs. Tuple

SN	List	Tuple
1	The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ().
2	The List is mutable.	The tuple is immutable.
3	The List has the a variable length.	The tuple has the fixed length.
4	The list provides more functionality than a tuple.	The tuple provides less functionality than the list.
5	The list is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items cannot be changed. It can be used as the key inside the dictionary.
6	The lists are less memory efficient than a tuple.	The tuples are more memory efficient because of its immutability.

Python Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

Creating a set

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example 1: Using curly braces

```
1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. print(Days)
3. print(type(Days))
4. print("looping through the set elements ... ")
5. for i in Days:
6.     print(i)
```

Output:

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}
<class 'set'>
looping through the set elements ...
Friday
Tuesday
Monday
Saturday
Thursday
```

```
Sunday
Wednesday
```

Example 2: Using set() method

```
1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. print(Days)
3. print(type(Days))
4. print("looping through the set elements ... ")
5. for i in Days:
6.     print(i)
```

Output:

```
{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}
<class 'set'>
looping through the set elements ...
Friday
Wednesday
Thursday
Saturday
Monday
Tuesday
Sunday
```

It can contain any type of element such as integer, float, tuple etc. But mutable elements (list, dictionary, set) can't be a member of set. Consider the following example.

```
1. # Creating a set which have immutable elements
2. set1 = {1,2,3, "JavaTpoint", 20.5, 14}
3. print(type(set1))
4. #Creating a set which have mutable element
5. set2 = {1,2,3,["Javatpoint",4]}
6. print(type(set2))
```

Output:

```
<class 'set'>

Traceback (most recent call last)
<ipython-input-5-9605bb6fbc68> in <module>
      4
      5 #Creating a set which holds mutable elements
----> 6 set2 = {1,2,3,["Javatpoint",4]}
      7 print(type(set2))

TypeError: unhashable type: 'list'
```

In the above code, we have created two sets, the set **set1** have immutable elements and set2 have one mutable element as a list. While checking the type of set2, it raised an error, which means set can contain only immutable elements.

Creating an empty set is a bit different because empty curly {} braces are also used to create a dictionary as well. So Python provides the set() method used without an argument to create an empty set.

```
# Empty curly braces will create dictionary
set3 = {}
print(type(set3))

# Empty set using set() function
set4 = set()
print(type(set4))
```

Output:

```
<class 'dict'>
<class 'set'>
```

Let's see what happened if we provide the duplicate element to the set.

```
set5 = {1,2,4,4,5,8,9,9,10}
```

```
2. print("Return set with unique elements:",set5)
```

Output:

```
Return set with unique elements: {1, 2, 4, 5, 8, 9, 10}
```

In the above code, we can see that **set5** consisted of multiple duplicate elements when we printed it remove the duplicity from the set.

Adding items to the set

Python provides the **add()** method and **update()** method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

Example: 1 - Using add() method

```
1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(months)
4. print("\nAdding other months to the set...");
5. Months.add("July");
6. Months.add ("August");
7. print("\nPrinting the modified set...");
8. print(Months)
9. print("\nlooping through the set elements ... ")
10. for i in Months:
11.     print(i)
```

Output:

```
printing the original set ...
```

```
{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...
{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...
February
July
May
April
March
August
June
January
```

To add more than one item in the set, Python provides the **update()** method. It accepts iterable as an argument.

Consider the following example.

Example - 2 Using update() function

```
1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nupdating the original set ... ")
5. Months.update(["July","August","September","October"]);
6. print("\nprinting the modified set ... ")
7. print(Months);
```

Output:

```
printing the original set ...
{'January', 'February', 'April', 'May', 'June', 'March'}
```

```
updating the original set ...
printing the modified set ...
{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}
```

Removing items from the set

Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these function, using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error.

Consider the following example.

Example-1 Using discard() method

```
1. months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(months)
4. print("\nRemoving some months from the set...");
5. months.discard("January");
6. months.discard("May");
7. print("\nPrinting the modified set...");
8. print(months)
9. print("\nlooping through the set elements ... ")
10. for i in months:
11.     print(i)
```

Output:

```
printing the original set ...
{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...
```

```
Printing the modified set...
{'February', 'March', 'April', 'June'}

looping through the set elements ...
February
March
April
June
```

Python provides also the **remove()** method to remove the item from the set. Consider the following example to remove the items using **remove()** method.

Example-2 Using remove() function

```
1. months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(months)
4. print("\nRemoving some months from the set...");
5. months.remove("January");
6. months.remove("May");
7. print("\nPrinting the modified set...");
8. print(months)
```

Output:

```
printing the original set ...
{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...
{'February', 'June', 'April', 'March'}
```

We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set.

Consider the following example to remove the item from the set using pop() method.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving some months from the set...");
Months.pop();
Months.pop();
print("\nPrinting the modified set...");
print(Months)
```

Output:

```
printing the original set ...
{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

Printing the modified set...
{'May', 'April', 'February', 'March'}
```

In the above code, the last element of the **Month** set is **March** but the pop() method removed the **June and January** because the set is unordered and the pop() method could not determine the last element of the set.

Python provides the clear() method to remove all the items from the set.

Consider the following example.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving all the items from the set...");
Months.clear()
```

```
6. print("\nPrinting the modified set...")
7. print(Months)
```

Output:

```
printing the original set ...
{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...
set()
```

Difference between discard() and remove()

Despite the fact that **discard()** and **remove()** method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the Python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the Python will raise an error.

Consider the following example.

Example-

```
1. Months = set(["January","February", "March", "April", "May", "June"])
2. print("\nprinting the original set ... ")
3. print(Months)
4. print("\nRemoving items through discard() method...");
5. Months.discard("Feb"); #will not give an error although the key feb is not available in the set
6. print("\nprinting the modified set...")
```

```
7. print(Months)
8. print("\nRemoving items through remove() method...");
9. Months.remove("Jan") #will give an error as the key jan is not available in the set.
10. print("\nPrinting the modified set...")
11. print(Months)
```

Output:

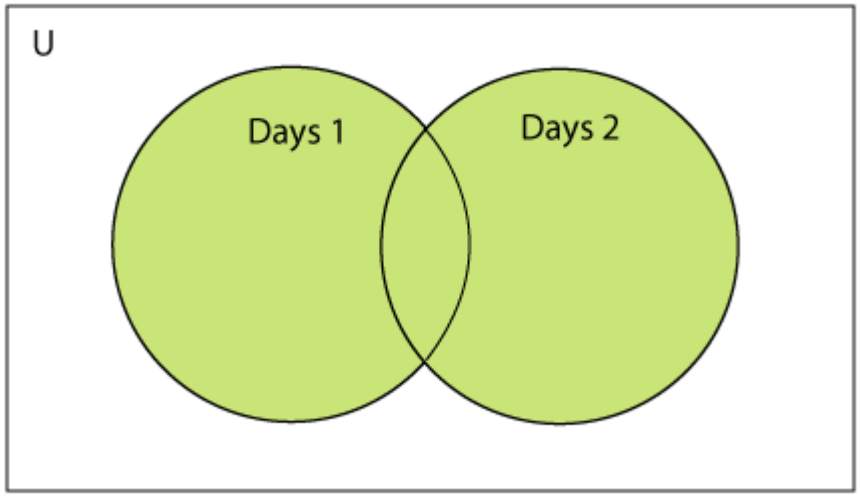
```
printing the original set ...
{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...
{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...
Traceback (most recent call last):
  File "set.py", line 9, in
    Months.remove("Jan")
KeyError: 'Jan'
```

Python Set Operations

Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

Union of two Sets

The union of two sets is calculated by using the pipe (|) operator. The union of the two sets contains all the items that are present in both the sets.



Consider the following example to calculate the union of two sets.

Example 1: using union | operator

```
1. Days1 = {"Monday","Tuesday","Wednesday","Thursday", "Sunday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. print(Days1|Days2) #printing the union of the sets
```

Output:

```
{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}
```

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

Example 2: using union() method

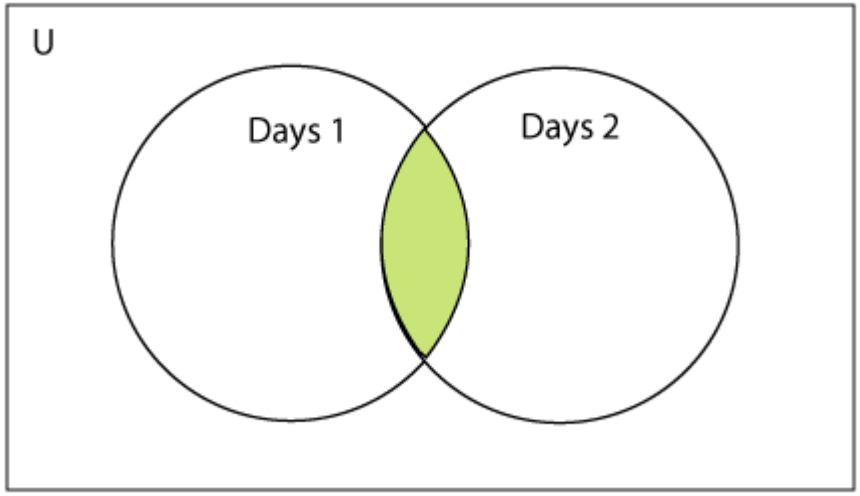
```
1. Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. print(Days1.union(Days2)) #printing the union of the sets
```

Output:

```
{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}
```

Intersection of two sets

The intersection of two sets can be performed by the **and &** operator or the **intersection() function**. The intersection of the two sets is given as the set of the elements that common in both sets.



Consider the following example.

Example 1: Using & operator

```
Days1 = {"Monday","Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday","Tuesday","Sunday", "Friday"}
print(Days1&Days2) #prints the intersection of the two sets
```

Output:

```
{'Monday', 'Tuesday'}
```

Example 2: Using intersection() method

```
set1 = {"Devansh","John", "David", "Martin"}
set2 = {"Steve", "Milan", "David", "Martin"}
print(set1.intersection(set2)) #prints the intersection of the two sets
```

Output:

```
{'Martin', 'David'}
```

Example 3:

```
set1 = {1,2,3,4,5,6,7}
set2 = {1,2,20,32,5,9}
set3 = set1.intersection(set2)
print(set3)
```

Output:

```
{1,2,5}
```

The intersection_update() method

The **intersection_update()** method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

The **intersection_update()** method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.

Consider the following example.

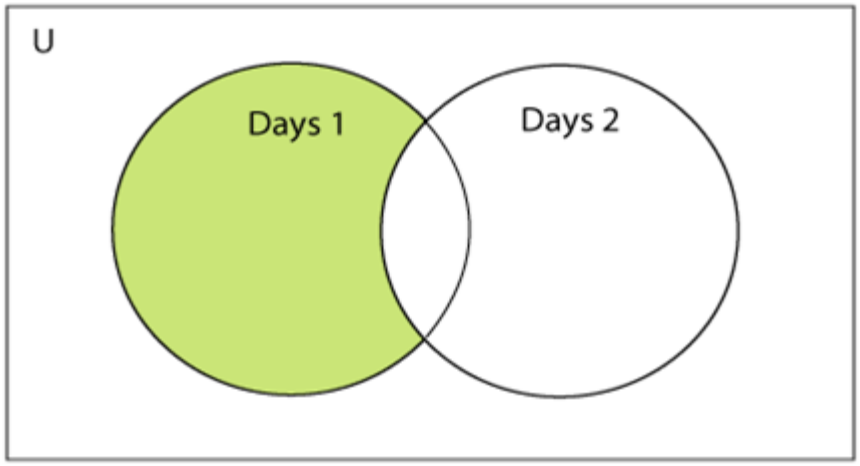
```
a = {"Devansh", "bob", "castle"}
b = {"castle", "dude", "emyway"}
c = {"fuson", "gaurav", "castle"}

a.intersection_update(b, c)

print(a)
```

Output:

```
{'castle'}
```

Difference between the two sets

The difference of two sets can be calculated by using the subtraction (-) operator or **intersection()** method. Suppose there are two sets A and B, and the difference is A-B that denotes the resulting set will be obtained that element of A, which is not present in the set B.



Consider the following example.

Example 1 : Using subtraction (-) operator

```
1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. print(Days1-Days2) #{"Wednesday", "Thursday" will be printed}
```

Output:

```
{'Thursday', 'Wednesday'}
```

Example 2 : Using difference() method

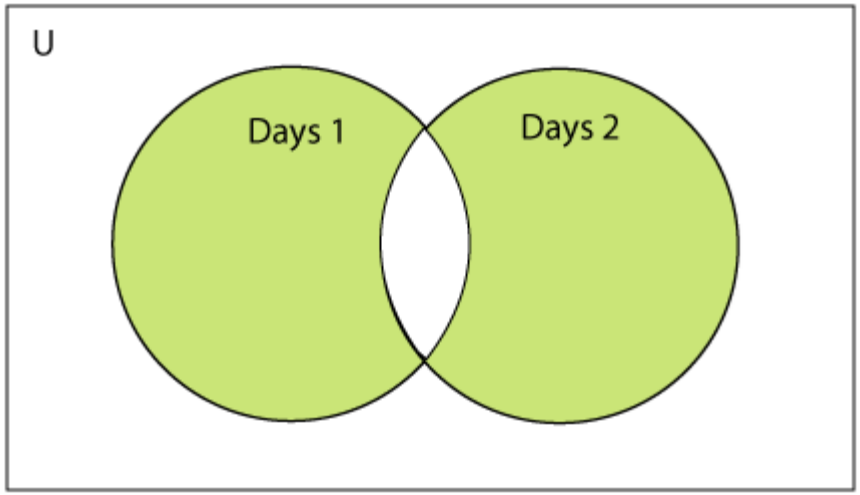
```
1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. print(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days2
```

Output:

```
{'Thursday', 'Wednesday'}
```

Symmetric Difference of two sets

The symmetric difference of two sets is calculated by ^ operator or **symmetric_difference()** method. Symmetric difference of sets, it removes that element which is present in both sets. Consider the following example:



Example - 1: Using ^ operator

```
a = {1,2,3,4,5,6}
b = {1,2,9,8,10}
c = a^b
print(c)
```

Output:

```
{3, 4, 5, 6, 8, 9, 10}
```

Example - 2: Using symmetric_difference() method

```
a = {1,2,3,4,5,6}
b = {1,2,9,8,10}
c = a.symmetric_difference(b)
print(c)
```

Output:

```
{3, 4, 5, 6, 8, 9, 10}
```

Set comparisons

Python allows us to use the comparison operators i.e., <, >, <=, >= , == with the sets by using which we can check whether a set is a subset, superset, or equivalent to other set. The boolean true or false is returned depending upon the items present inside the sets.

Consider the following example.

```
Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday", "Tuesday"}
Days3 = {"Monday", "Tuesday", "Friday"}

#Days1 is the superset of Days2 hence it will print true.
print (Days1>Days2)

#prints false since Days1 is not the subset of Days2
print (Days1<Days2)
```

```
10. 
11. #prints false since Days2 and Days3 are not equivalent
12. print (Days2 == Days3)
```

Output:

```
True
False
False
```

FrozenSets

The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set cannot be changed and therefore it can be used as a key in the dictionary.

The elements of the frozen set cannot be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Consider the following example to create the frozen set.

```
1. Frozenset = frozenset([1,2,3,4,5])
2. print(type(Frozenset))
3. print("\nprinting the content of frozen set...")
4. for i in Frozenset:
5.     print(i);
6. Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation
```

Output:

```
<class 'frozenset'>
```

```
printing the content of frozen set...
1
2
3
4
5
Traceback (most recent call last):
  File "set.py", line 6, in <module>
    Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation
AttributeError: 'frozenset' object has no attribute 'add'
```

Frozenset for the dictionary

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

Consider the following example.

```
1. Dictionary = {"Name":"John", "Country":"USA", "ID":101}
2. print(type(Dictionary))
3. Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary
4. print(type(Frozenset))
5. for i in Frozenset:
6.     print(i)
```

Output:

```
<class 'dict'>
<class 'frozenset'>
Name
Country
ID
```

Set Programming Example

Example - 1: Write a program to remove the given number from the set.

```
1. my_set = {1,2,3,4,5,6,12,24}
2. n = int(input("Enter the number you want to remove"))
3. my_set.discard(n)
4. print("After Removing:",my_set)
```

Output:

```
Enter the number you want to remove:12
After Removing: {1, 2, 3, 4, 5, 6, 24}
```

Example - 2: Write a program to add multiple elements to the set.

```
1. set1 = set([1,2,4,"John","CS"])
2. set1.update(["Apple","Mango","Grapes"])
3. print(set1)
```

Output:

```
{1, 2, 4, 'Apple', 'John', 'CS', 'Mango', 'Grapes'}
```

Example - 3: Write a program to find the union between two set.

```
1. set1 = set(["Peter","Joseph", 65,59,96])
2. set2  = set(["Peter",1,2,"Joseph"])
3. set3 = set1.union(set2)
4. print(set3)
```

Output:

```
{96, 65, 2, 'Joseph', 1, 'Peter', 59}
```

Example- 4: Write a program to find the intersection between two sets.

```
set1 = {23,44,56,67,90,45,"Javatpoint"}
set2 = {13,23,56,76,"Sachin"}
set3 = set1.intersection(set2)
print(set3)
```

Output:

```
{56, 23}
```

Example - 5: Write the program to add element to the frozenset.

```
set1 = {23,44,56,67,90,45,"Javatpoint"}
set2 = {13,23,56,76,"Sachin"}
set3 = set1.intersection(set2)
print(set3)
```

Output:

```
TypeError: 'frozenset' object does not support item assignment
```

Above code raised an error because frozensets are immutable and can't be changed after creation.

Example - 6: Write the program to find the issuperset, issubset and superset.

```
set1 = set(["Peter","James","Camroon","Ricky","Donald"])
set2 = set(["Camroon","Washington","Peter"])
set3 = set(["Peter"])

issubset = set1 >= set2
print(issubset)
```

```
issuperset = set1 <= set2
print(issuperset)
issubset = set3 <= set2
print(issubset)
issuperset = set2 >= set3
print(issuperset)
```

Output:

```
False
False
True
True
```

Python Built-in set methods

Python contains the following methods to be used with the sets.

SN	Method	Description
1	add(item)	It adds an item to the set. It has no effect if the item is already present in the set.
2	clear()	It deletes all the items from the set.
3	copy()	It returns a shallow copy of the set.
4	difference_update(....)	It modifies this set by removing all the items that are also present in the specified sets.
5	discard(item)	It removes the specified item from the set.
6	intersection()	It returns a new set that contains only the common elements of both the sets.

		(all the sets if more than two are specified).
7	intersection_update(....)	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
8	Isdisjoint(....)	Return True if two sets have a null intersection.
9	Issubset(....)	Report whether another set contains this set.
10	Issuperset(....)	Report whether this set contains another set.
11	pop()	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
12	remove(item)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
13	symmetric_difference(....)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
14	symmetric_difference_update(....)	Update a set with the symmetric difference of itself and another.
15	union(....)	Return the union of sets as a new set. (i.e. all elements that are in either set.)
16	update()	Update a set with the union of itself and others.

Python Dictionary

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

Syntax:

```
Dict = {"Name": "Tom", "Age": 22}
```

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

Let's see an example to create a dictionary and print its content.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
```

Output

```
<class 'dict'>
Printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
print("\nCreate Dictionary by using  dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Create Dictionary by using dict():
{1: 'Java', 2: 'T', 3: 'Point'}

Dictionary with each item as a pair:
```

```
{1: 'Devansh', 2: 'Sharma'}
```

Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print("Name : %s" %Employee["Name"])
5. print("Age : %d" %Employee["Age"])
6. print("Salary : %d" %Employee["salary"])
7. print("Company : %s" %Employee["Company"])
```

Output:

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Adding dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

Let's see an example to update the dictionary values.

Example - 1:

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Adding elements to dictionary one at a time
Dict[0] = 'Peter'
Dict[2] = 'Joseph'
Dict[3] = 'Ricky'
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# with a single Key
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[3] = 'JavaTpoint'
```

```
print("\nUpdated key value: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Example - 2:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....");
Employee["Name"] = input("Name: ");
Employee["Age"] = int(input("Age: "));
Employee["salary"] = int(input("Salary: "));
Employee["Company"] = input("Company:");
print("printing the new data");
print(Employee)
```

Output:

```
Empty Dictionary:
```

```
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp_ages': (20, 33, 24)}
```

Deleting elements using del keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

```
1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. print(type(Employee))
3. print("printing Employee data .... ")
4. print(Employee)
5. print("Deleting some of the employee data")
6. del Employee["Name"]
7. del Employee["Company"]
8. print("printing the modified information ")
9. print(Employee)
10. print("Deleting the dictionary: Employee");
11. del Employee
12. print("Lets try to print it again ");
13. print(Employee)
```

Output:

```
<class 'dict'>
printing Employee data ....
```

```
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

- **Using pop() method**

The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.

```
# Creating a Dictionary
Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
# Deleting a key
# using pop() method
pop_ele = Dict.pop(3)
print(Dict)
```

Output:

```
{1: 'JavaTpoint', 2: 'Peter'}
```

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

Iterating Dictionary

A dictionary can be iterated using for loop as given below.

Example 1

 # for loop to print all the keys of a dictionary

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
    print(x)
```

Output:

```
Name
Age
salary
Company
```

Example 2

#for loop to print all the values of the dictionary

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
    print(Employee[x])
```

Output:

```
John
29
25000
GOOGLE
```

Example - 3

#for loop to print the values of the dictionary by using values() method.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.values():
    print(x)
```

Output:

```
John
29
25000
GOOGLE
```

Example 4

#for loop to print the items of the dictionary by using items() method.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.items():
    print(x)
```

Output:

```
('Name', 'John')
('Age', 29)
('salary', 25000)
('Company', 'GOOGLE')
```

Properties of Dictionary keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

```
Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}
for x,y in Employee.items():
    print(x,y)
```

Output:

```
Name John
Age 29
Salary 25000
Company GOOGLE
```

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Consider the following example.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
for x,y in Employee.items():
    print(x,y)
```

Output:

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in
    Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

Built-in Dictionary functions

The built-in python dictionary methods along with the description are given below.

SN	Function	Description

1	cmp(dict1, dict2)	It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false.
2	len(dict)	It is used to calculate the length of the dictionary.
3	str(dict)	It converts the dictionary into the printable string representation.
4	type(variable)	It is used to print the type of the passed variable.

Built-in Dictionary methods

The built-in python dictionary methods along with the description are given below.

SN	Method	Description
1	dic.clear()	It is used to delete all the items of the dictionary.
2	dict.copy()	It returns a shallow copy of the dictionary.
3	dict.fromkeys(iterable, value = None, /)	Create a new dictionary from the iterable with the values equal to value.
4	dict.get(key, default = "None")	It is used to get the value specified for the passed key.
5	dict.has_key(key)	It returns true if the dictionary contains the specified key.
6	dict.items()	It returns all the key-value pairs as a tuple.

7	dict.keys()	It returns all the keys of the dictionary.
8	dict.setdefault(key,default= "None")	It is used to set the key to the default value if the key is not specified in the dictionary
9	dict.update(dict2)	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.
10	dict.values()	It returns all the values of the dictionary.
11	len()	
12	popItem()	
13	pop()	
14	count()	
15	index()	

Python Function

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

Python provide us various inbuilt functions like **range()** or **print()**. Although, the user can create its functions, which can be called user-defined functions.

There are mainly two types of functions.

- **User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.
- **Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

In this tutorial, we will discuss the user define functions.

Advantage of Functions in Python

There are the following advantages of Python functions.

- Using functions, we can avoid rewriting the same logic/code again and again in a program.
- We can call Python functions multiple times in a program and anywhere in a program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.

Creating a Function

Python provides the **def** keyword to define the function. The syntax of the define function is given below.

Syntax:

```
def my_function(parameters):
    function_block
return expression
```

Let's understand the syntax of functions definition.

- The **def** keyword, along with the function name is used to define the function.
- The identifier rule must follow the function name.
- A function accepts the parameter (argument), and they can be optional.
- The function block is started with the colon (:), and block statements must be at the same indentation.
- The **return** statement is used to return the value. A function can have only one **return**

Function Calling

In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

Consider the following example of a simple example that prints the message "Hello World".

```
#function definition
def hello_world():
    print("hello world")
# function calling
hello_world()
```

Output:

```
hello world
```

The return statement

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

Syntax

1. **return** [expression_list]

It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

Consider the following example:

Example 1

```
1. # Defining function
2. def sum():
3.     a = 10
4.     b = 20
5.     c = a+b
6.     return c
7. # calling sum() function in print statement
8. print("The sum is:",sum())
```

Output:

```
The sum is: 30
```

In the above code, we have defined the function named **sum,** and it has a statement **c = a+b,** which computes the given values, and the result is returned by the return statement to the caller function.

Example 2 Creating function without return statement

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
# calling sum() function in print statement
print(sum())
```

Output:

```
None
```

In the above code, we have defined the same function without the return statement as we can see that the **sum()** function returned the **None** object to the caller function.

Arguments in function

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

Consider the following example, which contains a function that accepts a string as the argument.

Example 1

```
#defining the function
def func (name):
```

```
    print("Hi ",name)
#calling the function
func("Devansh")
```

Output:

```
Hi Devansh
```

Example 2

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 20
Sum =  30
```

Call by reference in Python

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

Example 1 Passing Immutable Object (List)

```
1. #defining the function
2. def change_list(list1):
3.     list1.append(20)
4.     list1.append(30)
5.     print("list inside function = ",list1)
6.
7. #defining the list
8. list1 = [10,30,40,50]
9.
10. #calling the function
11. change_list(list1)
12. print("list outside function = ",list1)
```

Output:

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

Example 2 Passing Mutable Object (String)

```
1. #defining the function
2. def change_string (str):
3.     str = str + " Hows you "
4.     print("printing the string inside function :",str)
```

```
5.
6. string1 = "Hi I am there"
7.
8. #calling the function
9. change_string(string1)
10.
11. print("printing the string outside function :",string1)
```

Output:

```
printing the string inside function : Hi I am there Hows you
printing the string outside function : Hi I am there
```

Types of arguments

There may be several types of arguments which can be passed at the time of function call.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Required Arguments

Till now, we have learned about function calling in Python. However, we can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.

Consider the following example.

Example 1

```
def func(name):
    message = "Hi "+name
    return message
name = input("Enter the name:")
print(func(name))
```

Output:

```
Enter the name: John
Hi John
```

Example 2

```
#the function simple_interest accepts three arguments and returns the simple interest accordingly
def simple_interest(p,t,r):
    return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

Output:

```
Enter the principle amount: 5000
Enter the rate of interest: 5
Enter the time in years: 3
Simple Interest:  750.0
```

Example 3

```
#the function calculate returns the sum of two arguments a and b
def calculate(a,b):
    return a+b
calculate(10) # this causes an error as we are missing a required arguments b.
```

Output:

```
TypeError: calculate() missing 1 required positional argument: 'b'
```

Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example 1

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
printme(name = "john")
```

Output:

```
My name is John and age is 22
```

Example 2

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in th
e function
printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age
```

Output:

```
My name is john and age is 22
My name is David and age is 10
```

Variable-length Arguments (*args)

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

However, at the function definition, we define the variable-length argument using the ***args** (star) as *<variable - name >.

Consider the following example.

Example

```
1. def printme(*names):
2.     print("type of passed argument is ",type(names))
3.     print("printing the passed arguments...")
4.     for name in names:
5.         print(name)
6. printme("john","David","smith","nick")
```

Output:

```
type of passed argument is  <class 'tuple'>
printing the passed arguments...
john
David
smith
nick
```

In the above code, we passed ***names** as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated ***arg names** using for loop.

Keyword arguments(**kwargs)

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

Example 1

```
1. #function func is called with the name and message as the keyword arguments
2. def func(name,message):
3.     print("printing the message with",name,"and ",message)
4.
5.     #name and message is copied with the values John and hello respectively
6.     func(name = "John",message="hello")
```

Output:

```
printing the message with John and  hello
```

Example 2 providing the values in different order at the calling

```
1. #The function simple_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case


2. def simple_interest(p,t,r):
3.     return (p*t*r)/100
```

```
4. print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

Output:

```
Simple Interest:  1900.0
```

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

Example 3

```
1. #The function simple_interest(p, t, r) is called with the keyword arguments.
2. def simple_interest(p,t,r):
3.     return (p*t*r)/100
4.
5. # doesn't find the exact match of the name of the arguments (keywords)
6. print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900))
```

Output:

```
TypeError: simple_interest() got an unexpected keyword argument 'time'
```

The Python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

Example 4

```
1. def func(name1,message,name2):
2.     print("printing the message with",name1,",",message,",and",name2)
```

```
3. #the first argument is not the keyword argument
4. func("John",message="hello",name2="David")
```

Output:

```
printing the message with John , hello ,and David
```

The following example will cause an error due to an in-proper mix of keyword and required arguments being passed in the function call.

Example 5

```
1. def func(name1,message,name2):
2.     print("printing the message with",name1,",",message,",and",name2)
3. func("John",message="hello","David")
```

Output:

```
SyntaxError: positional argument follows keyword argument
```

Python provides the facility to pass the multiple keyword arguments which can be represented as ****kwargs**. It is similar as the ***args** but it stores the argument in the dictionary format.

This type of arguments is useful when we do not know the number of arguments in advance.

Consider the following example:

Example 6: Many arguments using Keyword argument

```
1. def food(**kwargs):
2.     print(kwargs)
3. food(a="Apple")
4. food(fruits="Orange", Vagitables="Carrot")
```

Output:

```
{'a': 'Apple'}
{'fruits': 'Orange', 'Vagitables': 'Carrot'}
```

Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

Example 1 Local Variable

```
1. def print_message():
2.     message = "hello !! I am going to print a message." # the variable message is local to the function itself
3.     print(message)
4. print_message()
5. print(message) # this will cause an error since a local variable cannot be accessible here.
```

Output:

```
hello !! I am going to print a message.
  File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in
```

```
    print(message)
NameError: name 'message' is not defined
```

Example 2 Global Variable

```
def calculate(*args):
    sum=0
    for arg in args:
        sum = sum +arg
    print("The sum is",sum)
sum=0
calculate(10,20,30) #60 will be printed as the sum
print("Value of sum outside the function:",sum) # 0 will be printed  Output:
```

Output:

```
The sum is 60
Value of sum outside the function: 0
```

Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

Python abs() Function Example

```
#  integer number
integer = -20
print('Absolute value of -40 is:', abs(integer))

#  floating number
floating = -20.83
print('Absolute value of -40.83 is:', abs(floating))
```

Output:

```
Absolute value of -20 is: 20
Absolute value of -20.83 is: 20.83
```

Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

Python all() Function Example

```
# all values true
k = [1, 3, 4, 6]
print(all(k))

# all values false
k = [0, False]
print(all(k))

# one false value
k = [1, 3, 7, 0]
print(all(k))

# one true value
k = [0, False, 5]
print(all(k))

# empty iterable
k = []
print(all(k))
```

Output:

```
True
False
False
False
True
```

Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Python bin() Function Example

```
1. x =  10
2. y =  bin(x)
3. print (y)
```

Output:

```
0b1010
```

Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

Python bool() Example

```
1. test1 = []
2. print(test1,'is',bool(test1))
3. test1 = [0]
4. print(test1,'is',bool(test1))
5. test1 = 0.0
6. print(test1,'is',bool(test1))
7. test1 = None
8. print(test1,'is',bool(test1))
9. test1 = True
```

```
print(test1,'is',bool(test1))
test1 = 'Easy string'
print(test1,'is',bool(test1))
```

Output:

```
[] is False
[0] is True
0.0 is False
None is False
True is True
Easy string is True
```

Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

Python bytes() Example

```
string = "Hello World."
array = bytes(string, 'utf-8')
print(array)
```

Output:

```
b ' Hello World.'
```

Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

Python callable() Function Example

```
x = 8
print(callable(x))
```

Output:

```
False
```

Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

Python compile() Function Example

```
# compile string source to code
code_str = 'x=5\ny=10\nprint("sum =",x+y)'
code = compile(code_str, 'sum.py', 'exec')
print(type(code))
exec(code)
exec(x)
```

Output:

```
<class 'code'>
sum = 15
```

Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

Python exec() Function Example

```
x = 8
exec('print(x==8)')
exec('print(x+4)')
```

Output:

```
True
12
```

Python sum() Function

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

Python sum() Function Example

```
s = sum([1, 2,4 ])
print(s)

s = sum([1, 2, 4], 10)
print(s)
```

Output:

```
7
```

```
17
```

Python any() Function

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.

Python any() Function Example

```
l = [4, 3, 2, 0]
print(any(l))

l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

Output:

```
True
False
True
False
```

Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

Python ascii() Function Example

```
normalText = 'Python is interesting'
print(ascii(normalText))

otherText = 'Pythön is interesting'
print(ascii(otherText))

print('Pyth\xf6n is interesting')
```

Output:

```
'Python is interesting'
'Pyth\xf6n is interesting'
Pythön is interesting
```

Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

Python bytearray() Example

```
string = "Python is a programming language."

# string with encoding 'utf-8'
arr = bytearray(string, 'utf-8')
print(arr)
```

Output:

```
bytearray(b'Python is a programming language.')
```

Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

Python eval() Function Example

```
x = 8
print(eval('x + 1'))
```

Output:

```
9
```

Python float()

The python **float()** function returns a floating-point number from a number or string.

Python float() Example

```
# for integers
print(float(9))

# for floats
print(float(8.19))

# for string floats
```

```
print(float("-24.27"))

# for string floats with whitespaces
print(float("    -17.19\n"))

# string float error
print(float("xyz"))
```

Output:

```
9.0
8.19
-24.27
-17.19
ValueError: could not convert string to float: 'xyz'
```

Python format() Function

The python **format()** function returns a formatted representation of the given value.

Python format() Function Example

```
# d, f and b are a type

# integer
print(format(123, "d"))

# float arguments
print(format(123.4567898, "f"))

```

```
# binary format
print(format(12, "b"))
```

Output:

```
123
123.456790
1100
```

Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

Python frozenset() Example

```
# tuple of letters
letters = ('m', 'r', 'o', 't', 's')

fSet = frozenset(letters)
print('Frozen set is:', fSet)
print('Empty frozen set is:', frozenset())
```

Output:

```
Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})
Empty frozen set is: frozenset()
```

Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Python getattr() Function Example

```
class Details:
    age = 22
    name = "Phill"

details = Details()
print('The age is:', getattr(details, "age"))
print('The age is:', details.age)
```

Output:

```
The age is: 22
The age is: 22
```

Python globals() Function

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python globals() Function Example

```
age = 22

globals()['age'] = 22
print('The age is:', age)
```

Output:

```
The age is: 22
```

Python hasattr() Function

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

Python hasattr() Function Example

```
l = [4, 3, 2, 0]
print(any(l))

l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

Output:

```
True
False
True
False
```

Python iter() Function

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

Python iter() Function Example

```
# list of numbers
list = [1,2,3,4,5]

listIter = iter(list)

# prints '1'
print(next(listIter))

# prints '2'
print(next(listIter))

# prints '3'
print(next(listIter))

# prints '4'
print(next(listIter))

# prints '5'
print(next(listIter))
```

Output:

```
1
2
3
4
5
```

Python len() Function

The python **len()** function is used to return the length (the number of items) of an object.

Python len() Function Example

```
strA = 'Python'
print(len(strA))
```

Output:

```
6
```

Python list()

The python **list()** creates a list in python.

Python list() Example

```
# empty list
print(list())

# string
String = 'abcde'
print(list(String))

# tuple
Tuple = (1,2,3,4,5)
print(list(Tuple))
```

```
# list
List = [1,2,3,4,5]
print(list(List))
```

Output:

```
[]
['a', 'b', 'c', 'd', 'e']
[1,2,3,4,5]
[1,2,3,4,5]
```

Python locals() Function

The python **locals()** method updates and returns the dictionary of the current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python locals() Function Example

```
def localsAbsent():
    return locals()

def localsPresent():
    present = True
    return locals()

print('localsNotPresent:', localsAbsent())
print('localsPresent:', localsPresent())
```

Output:

```
localsAbsent: {}
localsPresent: {'present': True}
```

Python map() Function

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

Python map() Function Example

```
1. def calculateAddition(n):
2.   return n+n
3.
4. numbers = (1, 2, 3, 4)
5. result = map(calculateAddition, numbers)
6. print(result)
7.
8. # converting map object to set
9. numbersAddition = set(result)
10. print(numbersAddition)
```

Output:

```
<map object at 0x7fb04a6bec18>
{8, 2, 4, 6}
```

Python memoryview() Function

The python **memoryview()** function returns a memoryview object of the given argument.

Python memoryview () Function Example

```
#A random bytearray
randomByteArray = bytearray('ABC', 'utf-8')

mv = memoryview(randomByteArray)

# access the memory view's zeroth index
print(mv[0])

# It create byte from memory view
print(bytes(mv[0:2]))

# It create list from memory view
print(list(mv[0:3]))
```

Output:

```
65
b'AB'
[65, 66, 67]
```

Python object()

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

Python object() Example

```
python = object()
```

2.
3. **print**(type(python))
4. **print**(dir(python))

Output:

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

Python open() Function

The python **open()** function opens the file and returns a corresponding file object.

Python open() Function Example

```
# opens python.text file of the current directory
f = open("python.txt")
# specifying full path
f = open("C:/Python33/README.txt")
```

Output:

```
Since the mode is omitted, the file is opened in 'r' mode; opens for reading.
```

Python chr() Function

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

Python chr() Function Example

```
# Calling function
result = chr(102) # It returns string representation of a char
result2 = chr(112)
# Displaying result
print(result)
print(result2)
# Verify, is it string type?
print("is it string type:", type(result) is str)
```

Output:

```
ValueError: chr() arg not in range(0x110000)
```

Python complex()

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

Python complex() Example

```
# Python complex() function example
# Calling function
a = complex(1) # Passing single parameter
b = complex(1,2) # Passing both parameters
# Displaying result
```

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

Output:

```
(1.5+0j)
(1.5+2.2j)
```

Python delattr() Function

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Python delattr() Function Example

```
class Student:
    id = 101
    name = "Pranshu"
    email = "pranshu@abc.com"
# Declaring function
    def getinfo(self):
        print(self.id, self.name, self.email)
s = Student()
s.getinfo()
delattr(Student,'course') # Removing attribute which is not available
s.getinfo() # error: throws an error
```

Output:

```
101 Pranshu pranshu@abc.com
```

```
AttributeError: course
```

Python dir() Function

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named __dir__(), this method will be called and must return the list of attributes. It takes a single object type argument.

Python dir() Function Example

```
# Calling function
att = dir()
# Displaying result
print(att)
```

Output:

```
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__']
```

Python divmod() Function

Python **divmod()** function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

Python divmod() Function Example

```
# Python divmod() function example
# Calling function
result = divmod(10,2)
# Displaying result
```

```
5. print(result)
```

Output:

```
(5, 0)
```

Python enumerate() Function

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

Python enumerate() Function Example

```
1. # Calling function
2. result = enumerate([1,2,3])
3. # Displaying result
4. print(result)
5. print(list(result))
```

Output:

```
<enumerate object at 0x7ff641093d80>
[(0, 1), (1, 2), (2, 3)]
```

Python dict()

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

- If no argument is passed, it creates an empty dictionary.

- If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
- If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

Python dict() Example

```
# Calling function
result = dict() # returns an empty dictionary
result2 = dict(a=1,b=2)
# Displaying result
print(result)
print(result2)
```

Output:

```
{}
{'a': 1, 'b': 2}
```

Python filter() Function

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

Python filter() Function Example

```
# Python filter() function example
def filterdata(x):
```

```
    if x>5:
        return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
print(list(result))
```

Output:

```
[6]
```

Python hash() Function

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

Hashable types: * bool * int * long * float * string * Unicode * tuple * code object.

Python hash() Function Example

```
# Calling function
result = hash(21) # integer value
result2 = hash(22.2) # decimal value
# Displaying result
print(result)
print(result2)
```

Output:

```
21
```

```
461168601842737174
```

Python help() Function

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

Python help() Function Example

```
# Calling function
info = help() # No argument
# Displaying result
print(info)
```

Output:

```
Welcome to Python 3.5's help utility!
```

Python min() Function

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

Python min() Function Example

```
# Calling function
small = min(2225,325,2025) # returns smallest element
small2 = min(1000.25,2025.35,5625.36,10052.50)
# Displaying result
print(small)
```

```
print(small2)
```

Output:

```
325
1000.25
```

Python set() Function

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

Python set() Function Example

```
# Calling function
result = set() # empty set
result2 = set('12')
result3 = set('javatpoint')
# Displaying result
print(result)
print(result2)
print(result3)
```

Output:

```
set()
{'1', '2'}
{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}
```

Python hex() Function

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

Python hex() Function Example

```
# Calling function
result = hex(1)
# integer value
result2 = hex(342)
# Displaying result
print(result)
print(result2)
```

Output:

```
0x1
0x156
```

Python id() Function

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

Python id() Function Example

```
# Calling function
val = id("Javatpoint") # string object
val2 = id(1200) # integer object
val3 = id([25,336,95,236,92,3225]) # List object
```

```
# Displaying result
print(val)
print(val2)
print(val3)
```

Output:

```
139963782059696
139963805666864
139963781994504
```

Python setattr() Function

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

Python setattr() Function Example

```
class Student:
    id = 0
    name = ""

    def __init__(self, id, name):
        self.id = id
        self.name = name

student = Student(102,"Sohan")
print(student.id)
print(student.name)
#print(student.email) product error
```

```
setattr(student, 'email','sohan@abc.com') # adding new attribute
print(student.email)
```

Output:

```
102
Sohan
sohan@abc.com
```

Python slice() Function

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

Python slice() Function Example

```
# Calling function
result = slice(5) # returns slice object
result2 = slice(0,5,3) # returns slice object
# Displaying result
print(result)
print(result2)
```

Output:

```
slice(None, 5, None)
slice(0, 5, 3)
```

Python sorted() Function

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

Python sorted() Function Example

```
str = "javatpoint" # declaring string
# Calling function
sorted1 = sorted(str) # sorting string
# Displaying result
print(sorted1)
```

Output:

```
['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']
```

Python next() Function

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

Python next() Function Example

```
number = iter([256, 32, 82]) # Creating iterator
# Calling function
item = next(number)
# Displaying result
print(item)
# second item
```

```
item = next(number)
print(item)
# third item
item = next(number)
print(item)
```

Output:

```
256
32
82
```

Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

Python input() Function Example

```
# Calling function
val = input("Enter a value: ")
# Displaying result
print("You entered:",val)
```

Output:

```
Enter a value: 45
You entered: 45
```

Python int() Function

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

Python int() Function Example

```
# Calling function
val = int(10) # integer value
val2 = int(10.52) # float value
val3 = int('10') # string value
# Displaying result
print("integer values :",val, val2, val3)
```

Output:

```
integer values : 10 10 10
```

Python isinstance() Function

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

Python isinstance() function Example

```
class Student:
    id = 101
    name = "John"
```

```
    def __init__(self, id, name):
        self.id=id
        self.name=name

student = Student(1010,"John")
lst = [12,34,5,6,767]
# Calling function
print(isinstance(student, Student)) # isinstance of Student class
print(isinstance(lst, Student))
```

Output:

```
True
False
```

Python oct() Function

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

Python oct() function Example

```
# Calling function
val = oct(10)
# Displaying result
print("Octal value of 10:",val)
```

Output:

```
Octal value of 10: 0o12
```

Python ord() Function

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

Python ord() function Example

```
# Code point of an integer
print(ord('8'))

# Code point of an alphabet
print(ord('R'))

# Code point of a character
print(ord('&'))
```

Output:

```
56
82
38
```

Python pow() Function

The python **pow()** function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

Python pow() function Example

```
# positive x, positive y (x**y)
print(pow(4, 2))

```

```
# negative x, positive y
print(pow(-4, 2))

# positive x, negative y (x**-y)
print(pow(4, -2))

# negative x, negative y
print(pow(-4, -2))
```

Output:

```
16
16
0.0625
0.0625
```

Python print() Function

The python **print()** function prints the given object to the screen or other standard output devices.

Python print() function Example

```
print("Python is programming language.")

x = 7
# Two objects passed
print("x =", x)

y = x
# Three objects passed
```

```
9. print('x =', x, '= y')
```

Output:

```
Python is programming language.
x = 7
x = 7 = y
```

Python range() Function

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

Python range() function Example

```
1. # empty range
2. print(list(range(0)))
3.
4. # using the range(stop)
5. print(list(range(4)))
6.
7. # using the range(start, stop)
8. print(list(range(1,7 )))
```

Output:

```
[]
[0, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

Python reversed() Function

The python **reversed()** function returns the reversed iterator of the given sequence.

Python reversed() function Example

```
# for string
String = 'Java'
print(list(reversed(String)))

# for tuple
Tuple = ('J', 'a', 'v', 'a')
print(list(reversed(Tuple)))

# for range
Range = range(8, 12)
print(list(reversed(Range)))

# for list
List = [1, 2, 7, 5]
print(list(reversed(List)))
```

Output:

```
['a', 'v', 'a', 'J']
['a', 'v', 'a', 'J']
[11, 10, 9, 8]
[5, 7, 2, 1]
```

Python round() Function

The python **round()** function rounds off the digits of a number and returns the floating point number.

Python round() Function Example

```
#  for integers
print(round(10))

#  for floating point
print(round(10.8))

#  even choice
print(round(6.6))
```

Output:

```
10
11
7
```

Python issubclass() Function

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

Python issubclass() Function Example

```
class Rectangle:
  def __init__(rectangleType):
    print('Rectangle is a ', rectangleType)

class Square(Rectangle):
```

```
    def __init__(self):
        Rectangle.__init__('square')

print(issubclass(Square, Rectangle))
print(issubclass(Square, list))
print(issubclass(Square, (list, Rectangle)))
print(issubclass(Rectangle, (list, Rectangle)))
```

Output:

```
True
False
True
True
```

Python str

The python **str()** converts a specified value into a string.

Python str() Function Example

```
str('4')
```

Output:

```
'4'
```

Python tuple() Function

The python **tuple()** function is used to create a tuple object.

Python tuple() Function Example

```
t1 = tuple()
print('t1=', t1)

# creating a tuple from a list
t2 = tuple([1, 6, 9])
print('t2=', t2)

# creating a tuple from a string
t1 = tuple('Java')
print('t1=',t1)

# creating a tuple from a dictionary
t1 = tuple({4: 'four', 5: 'five'})
print('t1=',t1)
```

Output:

```
t1= ()
t2= (1, 6, 9)
t1= ('J', 'a', 'v', 'a')
t1= (4, 5)
```

Python type()

The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

Python type() Function Example

```
List = [4, 5]
print(type(List))

Dict = {4: 'four', 5: 'five'}
print(type(Dict))

class Python:
    a = 0

InstanceOfPython = Python()
print(type(InstanceOfPython))
```

Output:

```
<class 'list'>
<class 'dict'>
<class '__main__.Python'>
```

Python vars() function

The python **vars()** function returns the __dict__ attribute of the given object.

Python vars() Function Example

```
class Python:
  def __init__(self, x = 7, y = 9):
    self.x = x
    self.y = y

InstanceOfPython = Python()
```

```
print(vars(InstanceOfPython))
```

Output:

```
{'y': 9, 'x': 7}
```

Python zip() Function

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

Python zip() Function Example

```
numList = [4,5, 6]
strList = ['four', 'five', 'six']

# No iterables are passed
result = zip()

# Converting itertor to list
resultList = list(result)
print(resultList)

# Two iterables are passed
result = zip(numList, strList)

# Converting itertor to set
resultSet = set(result)
print(resultSet)
```

Output:

```
[]
{(5, 'five'), (4, 'four'), (6, 'six')}
```

Python Lambda Functions

Python Lambda function is known as the anonymous function that is defined without a name. Python allows us to not declare the function in the standard manner, i.e., by using the **def** keyword. Rather, the anonymous functions are declared by using the **lambda** keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The syntax to define an anonymous function is given below.

Syntax

1. **lambda** arguments: expression

   It can accept any number of arguments and has only one expression. It is useful when the function objects are required.

   Consider the following example of the lambda function.

Example 1

```
1. # a is an argument and a+10 is an expression which got evaluated and returned.
2. x = lambda a:a+10
3. # Here we are printing the function object
```

```
print(x)
print("sum = ",x(20))
```

Output:

```
<function <lambda> at 0x0000019E285D16A8>
sum =  30
```

In the above example, we have defined the **lambda a: a+10** anonymous function where **a** is an argument and **a+10** is an expression. The given expression gets evaluated and returned the result. The above lambda function is same as the normal function.

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

Python File Handling

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

1. file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

SN	Access mode	Description
1	r	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It

		creates a new file if no file exists with the same name.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	ab+	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example

```
1. #opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
3.
4. if fileptr:
5.     print("file is opened successfully")
```

Output:

```
<class '_io.TextIOWrapper'>
file is opened successfully
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

The close() method

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

Syntax

```
1. fileobject.close()
```

Consider the following example.

```
1. # opens the file file.txt in read mode
2. fileptr = open("file.txt","r")
3.
4. if fileptr:
5.     print("file is opened successfully")
6.
7. #closes the opened file
8. fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

```
1. try:
2.    fileptr = open("file.txt")
3.    # perform file operations
4. finally:
```

```
5.    fileptr.close()
```

The with statement

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

```
1. with open(<file name>, <access mode>) as <file-pointer>:
2.     #statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

Example

```
1. with open("file.txt",'r') as f:
2.     content = f.read();
3.     print(content)
```

Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

Example

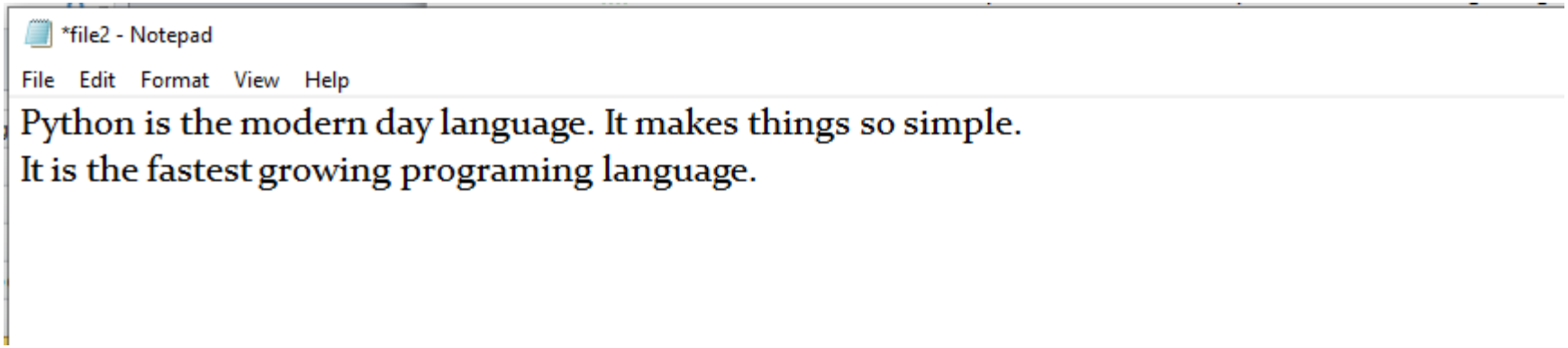
```
# open the file.txt in append mode. Create a new file if no such file exists.
fileptr = open("file2.txt", "w")

# appending the content to the file
fileptr.write('''Python is the modern day language. It makes things so simple.
It is the fastest-growing programing language''')

# closing the opened the file
fileptr.close()
```

Output:

File2.txt

```
Python is the modern-day language. It makes things so simple. It is the fastest growing programming
language.
```

Snapshot of the file2.txt



We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

Example 2

```
1. #open the file.txt in write mode.
2. fileptr = open("file2.txt","a")
3.
4. #overwriting the content of the file
5. fileptr.write(" Python has an easy syntax and user-friendly interaction.")
6.
7. #closing the opened file
8. fileptr.close()
```

Output:

```
Python is the modern day language. It makes things so simple.
It is the fastest growing programing language Python has an easy syntax and user-friendly interaction.
```

Snapshot of the file2.txt

We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax:

```
fileobj.read(<count>)
```

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

Example

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r")
```

```
3. #stores all the data of the file into the variable content
4. content = fileptr.read(10)
5. # prints the type of the data stored in the file
6. print(type(content))
7. #prints the content of the file
8. print(content)
9. #closes the opened file
10. fileptr.close()
```

Output:

```
<class 'str'>
Python is
```

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

```
1. content = fileptr.read()
2. print(content)
```

Output:

```
Python is the modern-day language. It makes things so simple.
It is the fastest-growing programing language Python has easy an syntax and user-friendly interaction.
```

Read file through for loop

We can read the file using for loop. Consider the following example.

```
1. #open the file.txt in read mode. causes an error if no such file exists.
```

```
2. fileptr = open("file2.txt","r");
3. #running a for loop
4. for i in fileptr:
5.     print(i) # i contains each line of the file
```

Output:

```
Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.
```

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

```
1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #stores all the data of the file into the variable content
4. content = fileptr.readline()
5. content1 = fileptr.readline()
6. #prints the content of the file
7. print(content)
8. print(content1)
```

```
#closes the opened file
fileptr.close()
```

Output:

```
Python is the modern day language.


It makes things so simple.
```

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readlines()

#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

Output:

```
['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-
friendly interaction.']
```

Creating a new file

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example 1

```
1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","x")
3. print(fileptr)
4. if fileptr:
5.     print("File created successfully")
```

Output:

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
File created successfully
```

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

```
1. # open the file file2.txt in read mode
```

```
2. fileptr = open("file2.txt","r")
3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte :",fileptr.tell())
6.
7. #reading the content of the file
8. content = fileptr.read();
9.
10. #after the read operation file pointer modifies. tell() returns the location of the fileptr.
11.
12. print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

The syntax to use the seek() method is given below.

Syntax:

```
1. <file-ptr>.seek(offset[, from)
```

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

Example

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.
fileptr.seek(10);

#tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

Python OS module

Renaming the file

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

Syntax:

```
1. rename(current-name, new-name)
```

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

Example 1:

```
1. import os
2.
3. #rename file2.txt to file3.txt
4. os.rename("file2.txt","file3.txt")
```

Output:

The above code renamed current **file2.txt** to **file3.txt**

Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

```
1. remove(file-name)
```

Example 1

```
import os;
#deleting the file named file3.txt
os.remove("file3.txt")
```

Creating the new directory

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax:

```
mkdir(directory name)
```

Example 1

```
import os

#creating a new directory with the name new
os.mkdir("new")
```

The getcwd() method

This method returns the current working directory.

The syntax to use the getcwd() method is given below.

Syntax

```
os.getcwd()
```

Example

```
1. import os
2. os.getcwd()
```

Output:

```
'C:\\Users\\DEVANSH SHARMA'
```

Changing the current working directory

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

Syntax

```
1. chdir("new-directory")
```

Example

```
1. import os
2. # Changing current directory with the new directiory
3. os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")
4. #It will display the current working directory
5. os.getcwd()
```

Output:

```
'C:\\Users\\DEVANSH SHARMA\\Documents'
```

Deleting directory

The rmdir() method is used to delete the specified directory.

The syntax to use the rmdir() method is given below.

Syntax

```
os.rmdir(directory name)
```

Example 1

```
import os
#removing the new directory
os.rmdir("directory_name")
```

It will remove the specified directory.

Writing Python output to the files

In Python, there are the requirements to write the output of a Python script to a file.

The **check_call()** method of module **subprocess** is used to execute a Python script and write the output of that script to a file.

The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file **output.txt.**

Example

file.py

```
temperatures=[10,-20,-289,100]
def c_to_f(c):
    if c< -273.15:
        return "That temperature doesn't make sense!"
```

```
    else:
        f=c*9/5+32
        return f
for t in temperatures:
    print(c_to_f(t))
```

file.py

```
import subprocess

with open("output.txt", "wb") as f:
    subprocess.check_call(["python", "file.py"], stdout=f)
```

The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Method	Description
1	file.close()	It closes the opened file. The file once closed, it can't be read or write anymore.
2	File.fush()	It flushes the internal buffer.
3	File.fileno()	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	File.isatty()	It returns true if the file is connected to a TTY device, otherwise returns false.
5	File.next()	It returns the next line from the file.

6	File.read([size])	It reads the file for the specified size.
7	File.readline([size])	It reads one line from the file and places the file pointer to the beginning of the new line.
8	File.readlines([sizehint])	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function.
9	File.seek(offset[,from)	It modifies the position of the file pointer to a specified offset with the specified reference.
10	File.tell()	It returns the current position of the file pointer within the file.
11	File.truncate([size])	It truncates the file to the optional specified size.
12	File.write(str)	It writes the specified string to a file
13	File.writelines(seq)	It writes a sequence of the strings to a file.

Python Modules

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Example

In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as **file.py.**

```
1. #displayMsg prints a message to the name being passed.
2. def displayMsg(name)
3.     print("Hi "+name);
```

Here, we need to include this module into our main module to call the method displayMsg() defined in the module named file.

Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is given below.

1. **import** module1,module2,..... module n

Hence, if we need to call the function displayMsg() defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

Example:

```
1. import file;
2. name = input("Enter the name?")
3. file.displayMsg(name)
```

Output:

```
Enter the name?John
Hi John
```

The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

1. **from** < module-name> **import** <name 1>, <name 2>..,<name n>

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

calculation.py:

```
1. #place the code in the calculation.py
2. def summation(a,b):
3.     return a+b
4. def multiplication(a,b):
5.     return a*b;
6. def divide(a,b):
7.     return a/b;
```

Main.py:

```
1. from calculation import summation
2. #it will import only the summation() from calculation.py
3. a = int(input("Enter the first number"))
4. b = int(input("Enter the second number"))
5. print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing summation()
```

Output:

```
Enter the first number10
Enter the second number20
Sum =  30
```

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *.

Consider the following syntax.

```
1. from <module> import *
```

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

1. **import** <module-name> as <specific-name>

Example

```
1. #the module calculation of previous example is imported in this example as cal.
2. import calculation as cal;
3. a = int(input("Enter a?"));
4. b = int(input("Enter b?"));
5. print("Sum = ",cal.summation(a,b))
```

Output:

```
Enter a?10
Enter b?20
Sum =  30
```

Python Exception

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

The problem without handling exceptions

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

Example

```
1. a = int(input("Enter a:"))
2. b = int(input("Enter b:"))
3. c = a/b
4. print("a/b = %d" %c)
5. 
6. #other code:
7. print("Hi I am other part of the program")
```

Output:

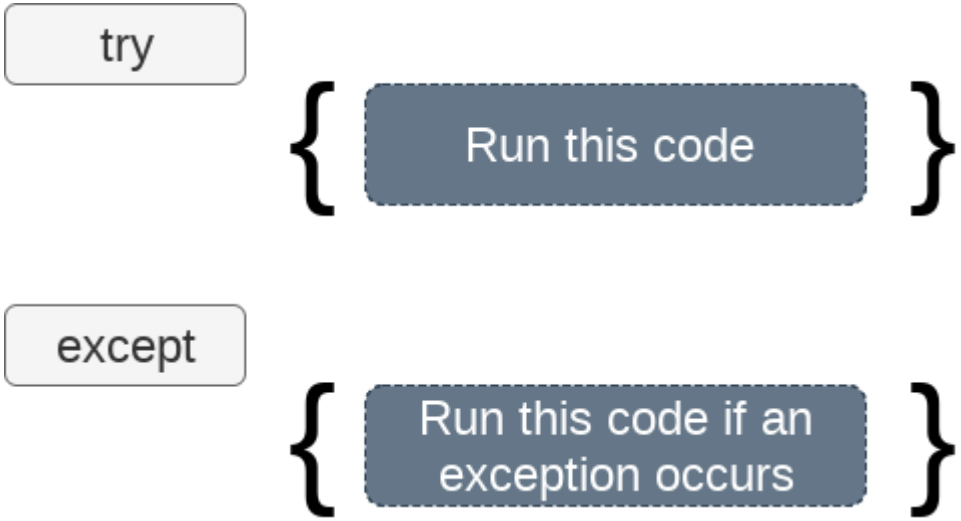
```
Enter a:10
Enter b:0
Traceback (most recent call last):
  File "exception-test.py", line 3, in <module>
    c = a/b;
ZeroDivisionError: division by zero
```

The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions. We can handle these exceptions in the following way.

Exception handling in python

The try-expect statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block.
The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

Consider the following example.

Example 1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

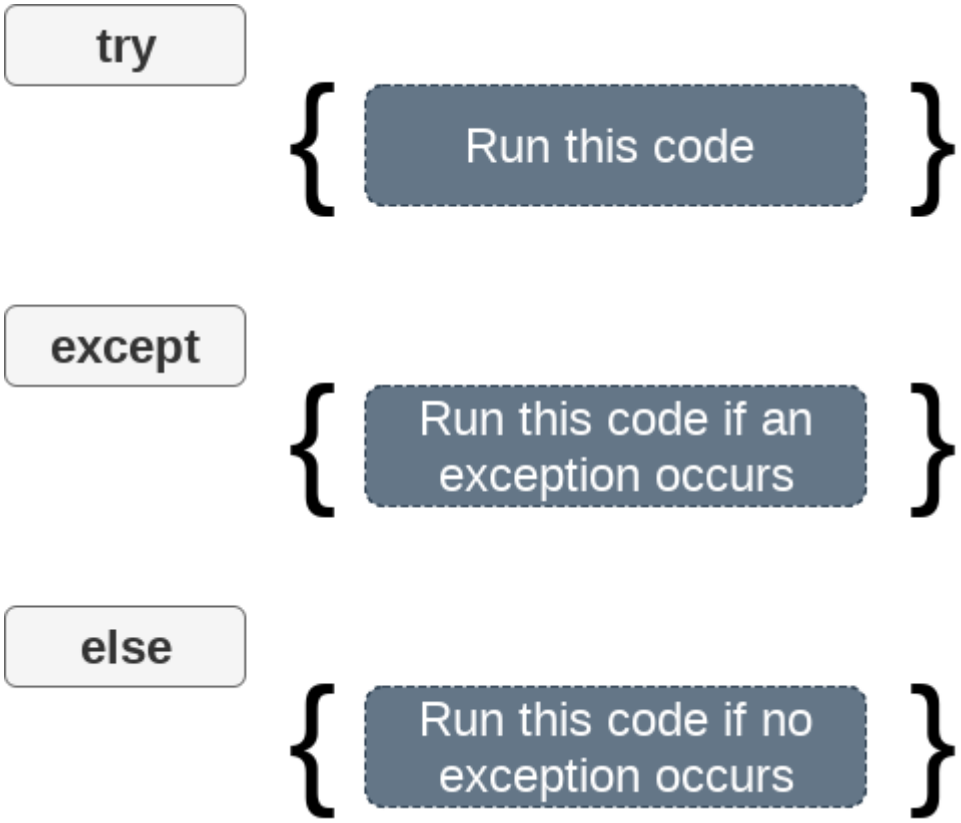
Output:

```
Enter a:10
Enter b:0
Can't divide with zero
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the else statement with the try-except statement is given below.

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```



Consider the following program.

Example 2

```
1. try:
2.     a = int(input("Enter a:"))
3.     b = int(input("Enter b:"))
4.     c = a/b
5.     print("a/b = %d"%c)
```

```
# Using Exception with except statement. If we print(Exception) it will return exception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```

The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement.

Consider the following example.

Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
division by zero
```

Points to remember

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.

4. The statements that don't throw the exception should be placed inside the else block.

Example

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

Output:

```
File not found
```

Declaring Multiple Exceptions

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

Syntax

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
```

```
8.     #block of code
```

Consider the following example.

```
1. try:
2.     a=10/0;
3. except(ArithmeticError, IOError):
4.     print("Arithmetic Exception")
5. else:
6.     print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

The try...finally block

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

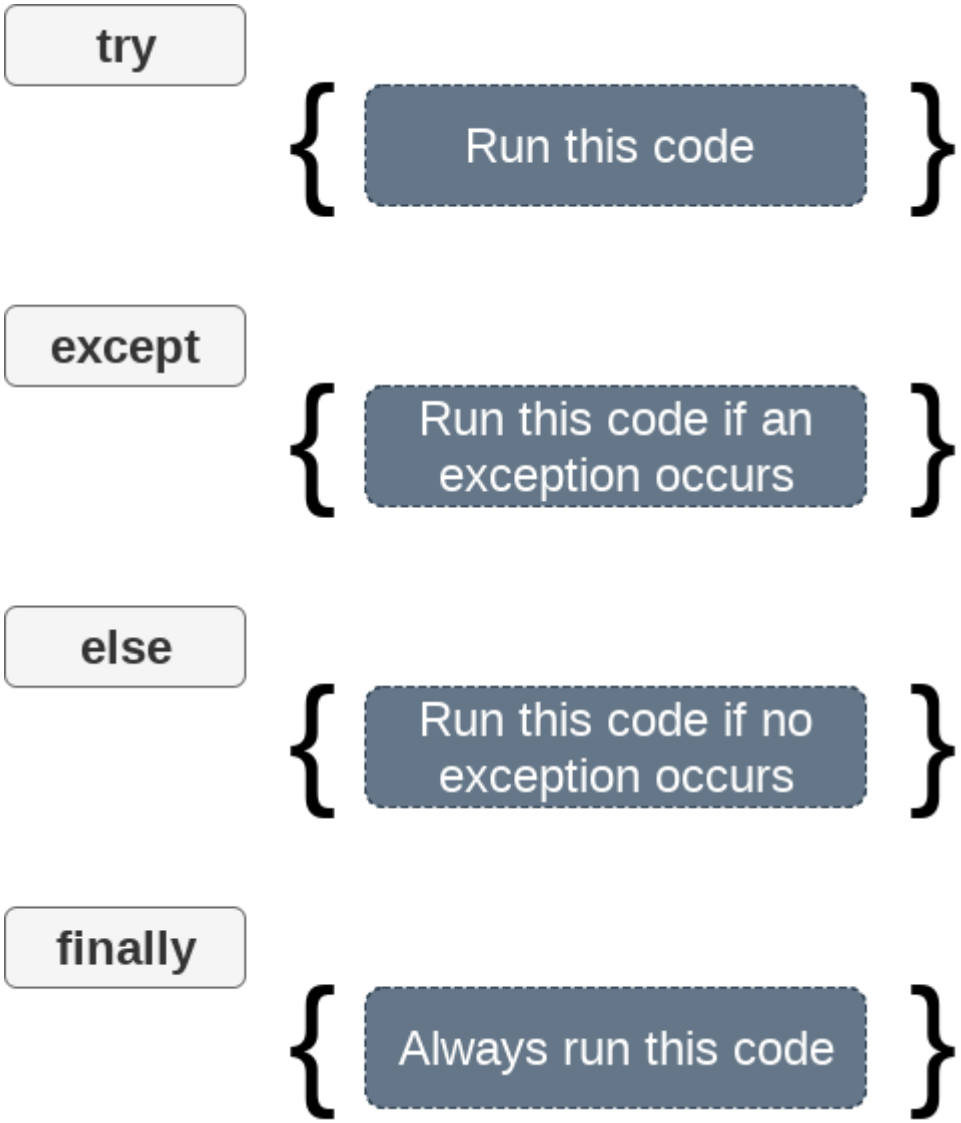
We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax

```
1. try:
2.     # block of code
3.     # this may throw an exception
4. finally:
5.     # block of code
```

6. # this will always be executed



Example

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Output:

```
file closed
Error
```

Raising exceptions

An exception can be raised forcefully by using the **raise** clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

Syntax

```
raise Exception_class,<value>
```

Points to remember

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.
4. We can pass the value to an exception to specify the exception type.

Example

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

```
Enter the age:17
The age is not valid
```

Example 2 Raise the exception with message

```
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
# we can pass the message in the raise statement
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print(e)
```

Output:

```
Enter a positive integer: -5
That is a negative number!
```

Example 3

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b is 0:
        raise ArithmeticError
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

Output:

```
Enter a:10
Enter b:0
The value of b can't be 0
```

Custom Exception

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

Consider the following example.

Example

```
class ErrorInCode(Exception):
```

```
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

Output:

```
Received error: 2000
```

Python Date and time

Python provides the **datetime** module work with real dates and times. In real-world applications, we need to work with the date and time. Python enables us to schedule our Python script to run at a particular timing.

In Python, the date is not a data type, but we can work with the date objects by importing the module named with **datetime, time, and calendar**.

In this section of the tutorial, we will discuss how to work with the date and time objects in Python.

The **datetime** classes are classified in the six main classes.

- **date** - It is a naive ideal date. It consists of the year, month, and day as attributes.
- **time** - It is a perfect time, assuming every day has precisely 24*60*60 seconds. It has hour, minute, second, microsecond, and **tzinfo** as attributes.
- **datetime** - It is a grouping of date and time, along with the attributes year, month, day, hour, minute, second, microsecond, and tzinfo.
- **timedelta -** It represents the difference between two dates, time or datetime instances to microsecond resolution.
- **tzinfo** - It provides time zone information objects.
- **timezone -** It is included in the new version of Python. It is the class that implements the **tzinfo** abstract base class.

Tick

In Python, the time instants are counted since 12 AM, 1st January 1970. The function **time()** of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

Consider the following example

```
1. import time;
2. #prints the number of ticks spent since 12 AM, 1st January 1970
```

3. **print**(time.time())

Output:

```
1585928913.6519969
```

How to get the current time?

The localtime() functions of the time module are used to get the current time tuple. Consider the following example.

Example

```
1. import time;
2.
3. #returns a time tuple
4.
5. print(time.localtime(time.time()))
```

Output:

```
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=3, tm_hour=21, tm_min=21, tm_sec=40, tm_wday=4,
tm_yday=94, tm_isdst=0)
```

Time tuple

The time is treated as the tuple of 9 numbers. Let's look at the members of the time tuple.

Index	Attribute	Values
0	Year	4 digit (for example 2018)

1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 60
6	Day of weak	0 to 6
7	Day of year	1 to 366
8	Daylight savings	-1, 0, 1 , or -1

Getting formatted time

The time can be formatted by using the **asctime()** function of the time module. It returns the formatted time for the time tuple being passed.

Example

```
1. import time
2.   #returns the formatted time
3.
4. print(time.asctime(time.localtime(time.time())))
```

Output:

```
Tue Dec 18 15:31:39 2018
```

Python sleep time

The **sleep()** method of time module is used to stop the execution of the script for a given amount of time. The output will be delayed for the number of seconds provided as the float.

Consider the following example.

Example

```
1. import time
2. for i in range(0,5):
3.     print(i)
4.     #Each element will be printed after 1 second
5.     time.sleep(1)
```

Output:

```
0
1
2
3
4
```

The datetime Module

The **datetime** module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.

To work with dates as date objects, we have to import **the datetime** module into the python source code.

Consider the following example to get the **datetime** object representation for the current time.

Example

```
import datetime
#returns the current datetime object
print(datetime.datetime.now())
```

Output:

```
2020-04-04 13:18:35.252578
```

Creating date objects

We can create the date objects bypassing the desired date in the datetime constructor for which the date objects are to be created.

Consider the following example.

Example

```
import datetime
#returns the datetime object for the specified date
print(datetime.datetime(2020,04,04))
```

Output:

```
2020-04-04 00:00:00
```

We can also specify the time along with the date to create the datetime object. Consider the following example.

Example

```
import datetime

```

```
#returns the datetime object for the specified time

print(datetime.datetime(2020,4,4,1,26,40))
```

Output:

```
2020-04-04 01:26:40
```

In the above code, we have passed in **datetime()** function year, month, day, hour, minute, and millisecond attributes in a sequential manner.

Comparison of two dates

We can compare two dates by using the comparison operators like >, >=, <, and <=.

Consider the following example.

Example

```
from datetime import datetime as dt
#Compares the time. If the time is in between 8AM and 4PM, then it prints working hours otherwise it prints fun hours
if dt(dt.now().year,dt.now().month,dt.now().day,8)<dt.now()<dt(dt.now().year,dt.now().month,dt.now().day,16):
    print("Working hours....")
else:
    print("fun hours")
```

Output:

```
fun hours
```

The calendar module

Python provides a calendar object that contains various methods to work with the calendars.

Consider the following example to print the calendar for the last month of 2018.

Example

```
import calendar;
cal = calendar.month(2020,3)
#printing the calendar of December 2018
print(cal)
```

Output:

```
March 2020
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 31
```

Printing the calendar of whole year

The prcal() method of calendar module is used to print the calendar of the entire year. The year of which the calendar is to be printed must be passed into this method.

Example

```
import calendar
#printing the calendar of the year 2020
s = calendar.prcal(2020)
```

Output:

2020

January

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	31		

February

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	

March

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	31					

April

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			

May

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	31

June

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					

July

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	31		

August

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
31						

September

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				

October

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	31	

November

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						

December

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	31			

Python Regular Expressions

The regular expressions can be defined as the sequence of characters which are used to search for a pattern in a string. The module re provides the support to use regex in the python program. The re module throws an exception if there is some error while using the regular expression.

The **re** module must be imported to use the regex functionalities in python.

```
1. import re
```

Regex Functions

The following regex functions are used in the python.

SN	Function	Description
1	match	This method matches the regex pattern in the string with the optional flag. It returns true if a match is found in the string otherwise it returns false.
2	search	This method returns the match object if there is a match found in the string.
3	findall	It returns a list that contains all the matches of a pattern in the string.
4	split	Returns a list in which the string has been split in each match.
5	sub	Replace one or many matches in the string.

Forming a regular expression

A regular expression can be formed by using the mix of meta-characters, special sequences, and sets.

Meta-Characters

Metacharacter is a character with the specified meaning.

Metacharacter	Description	Example
[]	It represents the set of characters.	"[a-z]"
\	It represents the special sequence.	"\r"
.	It signals that any character is present at some specific place.	"Ja.v."
^	It represents the pattern present at the beginning of the string.	"^Java"
$	It represents the pattern present at the end of the string.	"point"
*	It represents zero or more occurrences of a pattern in the string.	"hello*"
+	It represents one or more occurrences of a pattern in the string.	"hello+"
{}	The specified number of occurrences of a pattern the string.	"java{2}"
|	It represents either this or that character is present.	"java|point"
()	Capture and group	

Special Sequences

Special sequences are the sequences containing \ followed by one of the characters.

Character	Description
\A	It returns a match if the specified characters are present at the beginning of the string.
\b	It returns a match if the specified characters are present at the beginning or the end of the string.
\B	It returns a match if the specified characters are present at the beginning of the string but not at the end.
\d	It returns a match if the string contains digits [0-9].
\D	It returns a match if the string doesn't contain the digits [0-9].
\s	It returns a match if the string contains any white space character.
\S	It returns a match if the string doesn't contain any white space character.
\w	It returns a match if the string contains any word characters.
\W	It returns a match if the string doesn't contain any word.
\Z	Returns a match if the specified characters are at the end of the string.

Sets

A set is a group of characters given inside a pair of square brackets. It represents the special meaning.

SN	Set	Description
1	[arn]	Returns a match if the string contains any of the specified characters in the set.
2	[a-n]	Returns a match if the string contains any of the characters between a to n.
3	[^arn]	Returns a match if the string contains the characters except a, r, and n.
4	[0123]	Returns a match if the string contains any of the specified digits.
5	[0-9]	Returns a match if the string contains any digit between 0 and 9.
6	[0-5][0-9]	Returns a match if the string contains any digit between 00 and 59.
10	[a-zA-Z]	Returns a match if the string contains any alphabet (lower-case or upper-case).

The findall() function

This method returns a list containing a list of all matches of a pattern within the string. It returns the patterns in the order they are found. If there are no matches, then an empty list is returned.

Consider the following example.

Example

```
1. import re
2.
3. str = "How are you. How is everything"
4.
5. matches = re.findall("How", str)
```

```
6.
7. print(matches)
8.
9. print(matches)
```

Output:

```
['How', 'How']
```

The match object

The match object contains the information about the search and the output. If there is no match found, the None object is returned.

Example

```
1. import re
2.
3. str = "How are you. How is everything"
4.
5. matches = re.search("How", str)
6.
7. print(type(matches))
8.
9. print(matches) #matches is the search object
```

Output:

```
<class '_sre.SRE_Match'>
<_sre.SRE_Match object; span=(0, 3), match='How'>
```

The Match object methods

There are the following methods associated with the Match object.

1. **span():** It returns the tuple containing the starting and end position of the match.
2. **string():** It returns a string passed into the function.
3. **group():** The part of the string is returned where the match is found.

Example

```
1. import re
2.
3. str = "How are you. How is everything"
4.
5. matches = re.search("How", str)
6.
7. print(matches.span())
8.
9. print(matches.group())
10.
11. print(matches.string)
```

Output:

```
(0, 3)
How
How are you. How is everything
```

Python OOPs Concepts

Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In Python, we can easily create and use classes and objects.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

Major principles of object-oriented programming system are given below.

- Class
- Object
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

```
1. class ClassName:
2.         <statement-1>
3.         .
```

4. .
5. <statement-N>

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

Example:

```
class car:
    def __init__(self,modelname, year):
        self.modelname = modelname
        self.year = year
    def display(self):
        print(self.modelname,self.year)

c1 = car("Toyota", 2016)
c1.display()
```

Output:

```
Toyota 2016
```

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attribute. The c1 object will allocate memory for these values. We will learn more about class and object in the next tutorial.

Method

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

Inheritance

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides the re-usability of the code.

Polymorphism

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways. For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

Object-oriented vs. Procedure-oriented Programming languages

The difference between object-oriented and procedure-oriented programming is given below:

Index	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
3.	It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.
4.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
5.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Python Class and Objects

We have already discussed in previous tutorial, a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated. Let's understand it by an example.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called instantiation.

In this section of the tutorial, we will discuss creating classes and objects in Python. We will also discuss how a class attribute is accessed by using the object.

Creating classes in Python

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

Syntax

```
1. class ClassName:
2.     #statement_suite
```

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__.** A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class **Employee** which contains two fields as Employee id, and name.

The class also contains a function **display()**, which is used to display the information of the **Employee.**

Example

```
1. class Employee:
```

```
    id = 10
    name = "Devansh"
    def display (self):
        print(self.id,self.name)
```

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

```
<object-name> = <class-name>(<arguments>)
```

The following example creates the instance of the class Employee defined in the above example.

Example

```
class Employee:
    id = 10
    name = "John"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
# Creating a emp instance of Employee class
```

```
emp = Employee()
emp.display()
```

Output:

```
ID: 10
Name: John
```

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

Delete the Object

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

Example

```
class Employee:
    id = 10
    name = "John"

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))
    # Creating a emp instance of Employee class

emp = Employee()

# Deleting the property of object
del emp.id
```

```
13. # Deleting the object itself
14. del emp
15. emp.display()
```

It will through the Attribute error because we have deleted the object **emp**.

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python

In Python, the method the **__init__()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.

We can pass any number of arguments at the time of creating the class object, depending upon the **__init__()** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

Example

```
1. class Employee:
2.     def __init__(self, name, id):
3.         self.id = id
4.         self.name = name
5.
6.     def display(self):
7.         print("ID: %d \nName: %s" % (self.id, self.name))
8.
9.
10. emp1 = Employee("John", 101)
11. emp2 = Employee("David", 102)
12.
13. # accessing display() method to print employee 1 information
14.
15. emp1.display()
16.
17. # accessing display() method to print employee 2 information
18. emp2.display()
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

Example

```
1. class Student:
2.     count = 0
3.     def __init__(self):
4.         Student.count = Student.count + 1
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

Example

```
1. class Student:
2.     # Constructor - non parameterized
3.     def __init__(self):
4.         print("This is non parametrized constructor")
5.     def show(self,name):
6.         print("Hello",name)
```

```
student = Student()
student.show("John")
```

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

Example

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("John")
student.show()
```

Output:

```
This is parametrized constructor
Hello John
```

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects. Consider the following example.

Example

```
class Student:
    roll_num = 101
    name = "Joseph"

    def display(self):
        print(self.roll_num,self.name)

st = Student()
st.display()
```

Output:

```
101 Joseph
```

More than One Constructor in Single class

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

Example

```
class Student:
    def __init__(self):
        print("The First Constructor")
    def __init__(self):
        print("The second contructor")

st = Student()
```

Output:

```
The Second Constructor
```

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. Internally, the object of the class will always call the last constructor if the class has multiple constructors.

Note: The constructor overloading is not allowed in Python.

Python built-in class functions

The built-in functions defined in the class are described in the following table.

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Example

```
1. class Student:
2.     def __init__(self, name, id, age):
```

```
        self.name = name
        self.id = id
        self.age = age

    # creates the object of the class Student
s = Student("John", 101, 22)

# prints the attribute name of the object s
print(getattr(s, 'name'))

# reset the value of attribute age to 23
setattr(s, "age", 23)

# prints the modified value of age
print(getattr(s, 'age'))

# prints true if the student contains the attribute with name id

print(hasattr(s, 'id'))
# deletes the attribute age
delattr(s, 'age')

# this will give an error since the attribute age has been deleted
print(s.age)
```

Output:

```
John
23
True
```

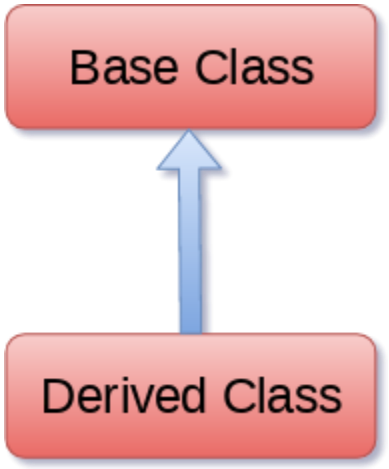
```
AttributeError: 'Student' object has no attribute 'age'
```

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.



Syntax

```
1. class derived-class(base class):
2.     <class-suite>
```

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Syntax

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
    <class - suite>
```

Example 1

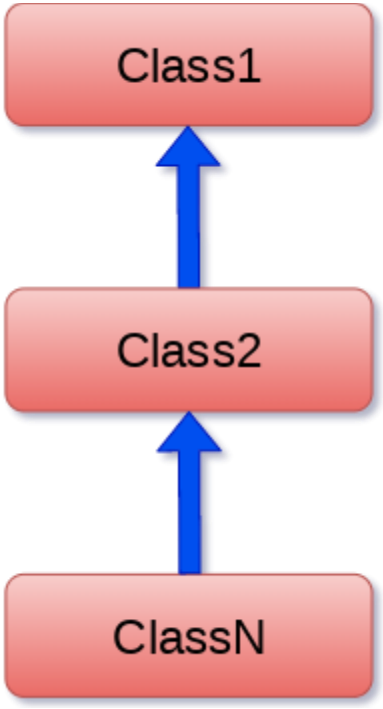
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
```

8. .

Example

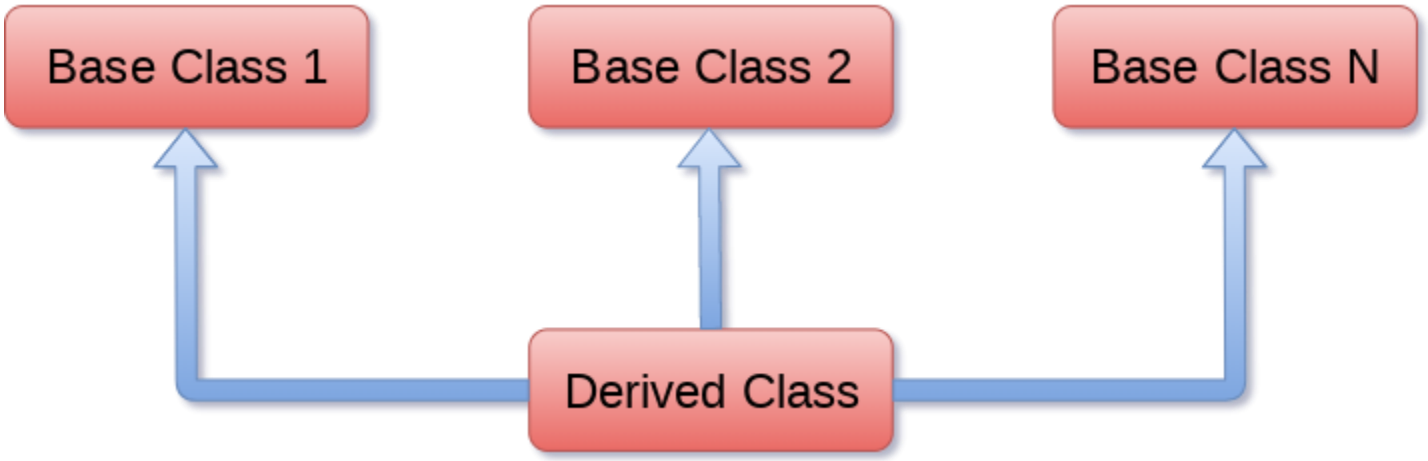
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

Output:

```
True
False
```

The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(isinstance(d,Derived))
```

Output:

```
True
```

Method Overriding

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

Example

```
class Animal:
    def speak(self):
        print("speaking")
```

```
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

Output:

```
Barking
```

Real Life Example of method overriding

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

Data abstraction in python

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Consider the following example.

Example

```
1. class Employee:
2.     __count = 0;
3.     def __init__(self):
4.         Employee.__count = Employee.__count+1
5.     def display(self):
6.         print("The number of employees",Employee.__count)
7. emp = Employee()
8. emp2 = Employee()
9. try:
10.     print(emp.__count)
11. finally:
12.     emp.display()
```

Output:

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```

Python Arrays

An array is defined as a collection of items that are stored at contiguous memory locations. It is a container which can hold a fixed number of items, and these items should be of the same type. An array is popular in most programming languages like C/C++, JavaScript, etc.

Array is an idea of storing multiple items of the same type together and it makes easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in single variable. If you have a list of items that are stored in their corresponding variables like this:

car1 = "Lamborghini"

car2 = "Bugatti"

car3 = "Koenigsegg"

If you want to loop through cars and find a specific one, you can use the array.

The array can be handled in Python by a module named **array**. It is useful when we have to manipulate only specific data values. Following are the terms to understand the concept of an array:

Element - Each item stored in an array is called an element.

Index - The location of an element in an array has a numerical index, which is used to identify the position of the element.

Array Representation

An array can be declared in various ways and different languages. The important points that should be considered are as follows:

- Index starts with 0.
- We can access each element via its index.

- The length of the array defines the capacity to store the elements.

Array operations

Some of the basic operations supported by an array are as follows:

- **Traverse** - It prints all the elements one by one.
- **Insertion** - It adds an element at the given index.
- **Deletion** - It deletes an element at the given index.
- **Search** - It searches an element using the given index or by the value.
- **Update** - It updates an element at the given index.

The Array can be created in Python by importing the array module to the python program.

```
1. from array import *
2. arrayName = array(typecode, [initializers])
```

Accessing array elements

We can access the array elements using the respective indices of those elements.

```
1. import array as arr
2. a = arr.array('i', [2, 4, 6, 8])
3. print("First element:", a[0])
4. print("Second element:", a[1])
5. print("Second last element:", a[-1])
```

Output:

```
First element: 2
Second element: 4
```

```
Second last element: 8
```

Explanation: In the above example, we have imported an array, defined a variable named as "a" that holds the elements of an array and print the elements by accessing elements through indices of an array.

How to change or add elements

Arrays are mutable, and their elements can be changed in a similar way like lists.

```
import array as arr
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])

# changing first element
numbers[0] = 0
print(numbers)    # Output: array('i', [0, 2, 3, 5, 7, 10])

# changing 3rd to 5th element
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)    # Output: array('i', [0, 2, 4, 6, 8, 10])
```

Output:

```
array('i', [0, 2, 3, 5, 7, 10])
array('i' ,[0, 2, 4, 6, 8, 10])
```

Explanation: In the above example, we have imported an array and defined a variable named as "numbers" which holds the value of an array. If we want to change or add the elements in an array, we can do it by defining the particular index of an array where you want to change or add the elements.

Why to use arrays in Python?

A combination of arrays saves a lot of time. The array can reduce the overall size of the code.

How to delete elements from an array?

The elements can be deleted from an array using Python's **del** statement. If we want to delete any value from the array, we can do that by using the indices of a particular element.

```
1. import array as arr
2. number = arr.array('i', [1, 2, 3, 3, 4])
3. del number[2]                    # removing third element
4. print(number)                    # Output: array('i', [1, 2, 3, 4])
```

Output:

```
array('i', [10, 20, 40, 60])
```

Explanation: In the above example, we have imported an array and defined a variable named as "number" which stores the values of an array. Here, by using del statement, we are removing the third element [3] of the given array.

Finding the length of an array

The length of an array is defined as the number of elements present in an array. It returns an integer value that is equal to the total number of the elements present in that array.

Syntax

```
1. len(array_name)
```

Array Concatenation

We can easily concatenate any two arrays using the + symbol.

Example

```
a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
b=arr.array('d',[3.7,8.6])
c=arr.array('d')
c=a+b
print("Array c = ",c)
```

Output:

```
Array c= array('d', [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])
```

Explanation

In the above example, we have defined variables named as "a, b, c" that hold the values of an array.

Example

```
import array as arr
x = arr.array('i', [4, 7, 19, 22])
print("First element:", x[0])
print("Second element:", x[1])
print("Second last element:", x[-1])
```

Output:

```
First element: 4
Second element: 7
Second last element: 22
```

Explanation: In the above example, first, we have imported an array and defined a variable named as "x" which holds the value of an array and then, we have printed the elements using the indices of an array.

Python Decorator

Decorators are one of the most helpful and powerful tools of Python. These are used to modify the behavior of the function. Decorators provide the flexibility to wrap another function to expand the working of wrapped function, without permanently modifying it.

"In Decorators, functions are passed as an argument into another function and then called inside the wrapper function."

It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.

Before understanding the **Decorator**, we need to know some important concepts of Python.

What are the functions in Python?

Python has the most interesting feature that everything is treated as an object even classes or any variable we define in Python is also assumed as an object. Functions are **first-class** objects in the Python because they can reference to, passed to a variable and returned from other functions as well. The example is given below:

```
1. def func1(msg):
2.     print(msg)
3. func1("Hii")
4. func2 = func1
5. func2("Hii")
```

Output:

```
Hii
Hii
```

In the above program, when we run the code it give the same output for both functions. The **func2** referred to function **func1** and act as function. We need to understand the following concept of the function:

- The function can be referenced and passed to a variable and returned from other functions as well.

- The functions can be declared inside another function and passed as an argument to another function.

Inner Function

Python provides the facility to define the function inside another function. These types of functions are called inner functions. Consider the following example:

```
1. def func():
2.     print("We are in first function")
3.     def func1():
4.         print("This is first child function")
5.     def func2():
6.         print(" This is second child function")
7.     func1()
8.     func2()
9. func()
```

Output:

```
We are in first function
This is first child function
This is second child function
```

In the above program, it doesn't matter how the child functions are declared. The execution of the child function makes effect on the output. These child functions are locally bounded with the **func()** so they cannot be called separately.

A function that accepts other function as an argument is also called **higher order function**. Consider the following example:

```
1. def add(x):
2.     return x+1
3. def sub(x):
4.     return x-1
```

```
def operator(func, x):
    temp = func(x)
    return temp
print(operator(sub,10))
print(operator(add,20))
```

Output:

```
9
21
```

In the above program, we have passed the **dec()** function and **inc()** function as argument in **operator()** function.

A function can return another function. Consider the below example:

```
def hello():
    def hi():
        print("Hello")
    return hi
new = hello()
new()
```

Output:

```
Hello
```

In the above program, the **hi()** function is nested inside the **hello()** function. It will return each time we call **hi()**.

Decorating functions with parameters

Let's have an example to understand the parameterized decorator function:

```
def divide(x,y):
    print(x/y)
def outer_div(func):
    def inner(x,y):
        if(x<y):
            x,y = y,x
            return func(x,y)
     return inner
divide1 = outer_div(divide)
divide1(2,4)
```

Output:

```
2.0
```

Syntactic Decorator

In the above program, we have decorated **out_div()** that is little bit bulky. Instead of using above method, Python allows to **use decorator in easy way with @symbol**. Sometimes it is called "pie" syntax.

```
def outer_div(func):
    def inner(x,y):
        if(x<y):
            x,y = y,x
            return func(x,y)
     return inner
# syntax of generator
@outer_div
def divide(x,y):
    print(x/y)
```

Output:

```
2.0
```

Reusing Decorator

We can reuse the decorator as well by recalling that decorator function. Let’s make the decorator to its own module that can be used in many other functions. Creating a file called **mod_decorator.py** with the following code:

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

We can import mod_decorator.py in other file.

```
from decorator import do_twice
@do_twice
def say_hello():
    print("Hello There")
say_hello()
```

Output:

```
Hello There
Hello There
```

Python Decorator with Argument

We want to pass some arguments in function. Let's do it in following code:

```
from decorator import do_twice
@do_twice
def display(name):
    print(f"Hello {name}")
display()
```

Output:

```
TypeError: display() missing 1 required positional argument: 'name'
```

As we can see that, the function didn't accept the argument. Running this code raises an error. We can fix this error by using ***args** and ****kwargs** in the inner wrapper function. Modifying the **decorator.py** as follows:

```
def do_twice(func):
    def wrapper_function(*args,**kwargs):
        func(*args,**kwargs)
        func(*args,**kwargs)
    return wrapper_function
```

Now **wrapper_function()** can accept any number of argument and pass them on the function.

```
from decorator import do_twice
@do_twice
def display(name):
    print(f"Hello {name}")
display("John")
```

Output:

```
Hello John
Hello John
```

Returning Values from Decorated Functions

We can control the return type of the decorated function. The example is given below:

```
1. from decorator import do_twice
2. @do_twice
3. def return_greeting(name):
4.     print("We are created greeting")
5.     return f"Hi {name}"
6. hi_adam = return_greeting("Adam")
```

Output:

```
We are created greeting
We are created greeting
```

Fancy Decorators

Let's understand the fancy decorators by the following topic:

Class Decorators

Python provides two ways to decorate a class. Firstly, we can decorate the method inside a class; there are built-in decorators like **@classmethod**, **@staticmethod** and **@property** in Python. The **@classmethod** and **@staticmethod** define methods inside class that is not connected to any other instance of a class. The @property is generally used to modify the getters and setters of a class attributes. Let’s understand it by the following example:

Example: 1- **@property decorator** - By using it, we can use the class function as an attribute. Consider the following code:

```
1. class Student:
2.     def __init__(self,name,grade):
3.         self.name = name
```

```
        self.grade = grade
    @property
    def display(self):
        return self.name + " got grade " + self.grade

stu = Student("John","B")
print("Name:", stu.name)
print("Grade:", stu.grade)
print(stu.display)
```

Output:

```
Name: John
Grade: B
John got grade B
```

Example:2 - **@staticmethod decorator**- The @staticmethod is used to define a static method in the class. It is called by using the class name as well as instance of the class. Consider the following code:

```
class Person:
    @staticmethod
    def hello():
        print("Hello Peter")
per = Person()
per.hello()
Person.hello()
```

Output:

```
Hello Peter
Hello Peter
```

Singleton Class

A singleton class only has one instance. There are many singletons in Python including True, None, etc.

Nesting Decorators

We can use multiple decorators by using them on top of each other. Let's consider the following example:

```
@function1
@function2
def function(name):
    print(f "{name}")
```

In the above code, we have used the nested decorator by stacking them onto one another.

Decorator with Arguments

It is always useful to pass arguments in a decorator. The decorator can be executed several times according to the given value of the argument. Let us consider the following example:

```
Import functools

def repeat(num):

#Creating and returning a wrapper function
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper(*args,**kwargs):
            for _ in range(num):
                value = func(*args,**kwargs)
            return value
```

```
        return wrapper
    return decorator_repeat

#Here we are passing num as an argument which repeats the print function
@repeat(num=5)
def function1(name):
    print(f"{name}")
```

Output:

```
JavatPoint
JavatPoint
JavatPoint
JavatPoint
JavatPoint
```

In the above example, **@repeat** refers to a function object that can be called in another function. The **@repeat(num = 5)** will return a function which acts as a decorator.

The above code may look complex but it is the most commonly used decorator pattern where we have used one additional **def** that handles the arguments to the decorator.

Python Generators

What is Python Generator?

Python Generators are the functions that return the traversal object and used to create iterators. It traverses the entire items at once. The generator can also be an expression in which syntax is similar to the list comprehension in Python.

There is a lot of complexity in creating iteration in Python; we need to implement **__iter__()** and **__next__()** method to keep track of internal states.

It is a lengthy process to create iterators. That's why the generator plays an essential role in simplifying this process. If there is no value found in iteration, it raises **StopIteration** exception.

How to Create Generator function in Python?

It is quite simple to create a generator in Python. It is similar to the normal function defined by the **def** keyword and uses a **yield** keyword instead of return. Or we can say that if the body of any function contains a **yield** statement, it automatically becomes a generator function. Consider the following example:

```
1. def simple():
2. for i in range(10):
3.     if(i%2==0):
4.         yield i
5.
6. #Successive Function call using for loop
7. for i in simple():
8.     print(i)
```

Output:

```
0
2
```

```
4
6
8
```

yield vs. return

The **yield** statement is responsible for controlling the flow of the generator function. It pauses the function execution by saving all states and yielded to the caller. Later it resumes execution when a successive function is called. We can use the multiple yield statement in the generator function.

The return statement **returns** a value and terminates the whole function and only one return statement can be used in the function.

Using multiple yield Statement

We can use the multiple yield statement in the generator function. Consider the following example.

```
1. def multiple_yield():
2.     str1 = "First String"
3.     yield str1
4.
5.     str2 = "Second string"
6.     yield str2
7.
8.     str3 = "Third String"
9.     yield str3
10. obj = multiple_yield()
11. print(next(obj))
12. print(next(obj))
13. print(next(obj))
```

Output:

```
First String
Second string
Third String
```

Difference between Generator function and Normal function

- Normal function contains only one L**return** statement whereas generator function can contain one or more **yield** statement.
- When the generator functions are called, the normal function is paused immediately and control transferred to the caller.
- Local variable and their states are remembered between successive calls.
- StopIteration exception is raised automatically when the function terminates.

Generator Expression

We can easily create a generator expression without using user-defined function. It is the same as the lambda function which creates an anonymous function; the generator's expressions create an anonymous generator function.

The representation of generator expression is similar to the Python list comprehension. The only difference is that **square bracket is replaced by round parentheses**. The list comprehension calculates the entire list, whereas the generator expression calculates one item at a time.

Consider the following example:

```
list = [1,2,3,4,5,6,7]

# List Comprehension
z = [x**3 for x in list]

# Generator expression
a = (x**3 for x in list)

```

```
print(a)
print(z)
```

Output:

```
<generator object <genexpr> at 0x01BA3CD8>
[1, 8, 27, 64, 125, 216, 343]
```

In the above program, list comprehension has returned the list of cube of elements whereas generator expression has returned the reference of calculated value. Instead of applying a **for loop**, we can also call **next()** on the generator object. Let's consider another example:

```
list = [1,2,3,4,5,6]

z = (x**3 for x in list)

print(next(z))

print(next(z))

print(next(z))

print(next(z))
```

Output:

```
1
8
27
64
```

Note:- When we call the next(), Python calls __next__() on the function in which we have passed it as a parameter.

In the above program, we have used the **next()** function, which returned the next item of the list.

Example: Write a program to print the table of the given number using the generator.

```
def table(n):
    for i in range(1,11):
        yield n*i
          i = i+1

for i in table(15):
    print(i)
```

Output:

```
15
30
45
60
75
90
105
120
135
150
```

In the above example, a generator function is iterating using for loop.

Advantages of Generators

There are various advantages of Generators. Few of them are given below:

1. Easy to implement

Generators are easy to implement as compared to the iterator. In iterator, we have to implement **__iter__()** and **__next__()** function.

2. Memory efficient

Generators are memory efficient for a large number of sequences. The normal function returns a sequence of the list which creates an entire sequence in memory before returning the result, but the generator function calculates the value and pause their execution. It resumes for successive call. An infinite sequence generator is a great example of memory optimization. Let's discuss it in the below example by using **sys.getsizeof()** function.

```
1. import sys
2. # List comprehension
3. nums_squared_list = [i * 2 for i in range(1000)]
4. print(sys.getsizeof("Memory in Bytes:"nums_squared_list))
5. # Generator Expression
6. nums_squared_gc = (i ** 2 for i in range(1000))
7. print(sys.getsizeof("Memory in Bytes:", nums_squared_gc))
```

Output:

```
Memory in Bytes: 4508
Memory in Bytes: 56
```

We can observe from the above output that list comprehension is using 4508 bytes of memory, whereas generator expression is using 56 bytes of memory. It means that generator objects are much efficient than the list compression.

3. Pipelining with Generators

Data Pipeline provides the facility to process large datasets or stream of data without using extra computer memory.

Suppose we have a log file from a famous restaurant. The log file has a column (4th column) that keeps track of the number of burgers sold every hour and we want to sum it to find the total number of burgers sold in 4 years. In that scenario, the generator can generate a pipeline with a series of operations. Below is the code for it:

```
with open('sells.log') as file:
burger_col = (line[3] for line in file)  per_hour = (int(x) for x in burger_col if x != 'N/A')
print("Total burgers sold = ",sum(per_hour))
```

4. Generate Infinite Sequence

The generator can produce infinite items. Infinite sequences cannot be contained within the memory and since generators produce only one item at a time, consider the following example:

```
def infinite_sequence():
    num = 0
    while True:
        yield num
            num += 1

for i in infinite_sequence():
    print(i)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

```
10
11
12
13
14
15
16
17
18
19
20
.........
..........
315
316
317
Traceback (most recent call last):
  File "C:\Users\DEVANSH SHARMA\Desktop\generator.py", line 33, in <module>
    print(i)
KeyboardInterrupt
```

In this tutorial, we have learned about the Python Generators.

Python read csv file

CSV File

A **csv** stands for "comma separated values", which is defined as a simple file format that uses specific structuring to arrange tabular data. It stores tabular data such as spreadsheet or database in plain text and has a common format for data interchange. A **csv** file opens into the excel sheet, and the rows and columns data define the standard format.

Python CSV Module Functions

The CSV module work is used to handle the CSV files to read/write and get data from specified columns. There are different types of CSV functions, which are as follows:

- **csv.field_size_limit -** It returns the current maximum field size allowed by the parser.
- **csv.get_dialect -** It returns the dialect associated with a name.
- **csv.list_dialects -** It returns the names of all registered dialects.
- **csv.reader -** It read the data from a csv file
- **csv.register_dialect -** It associates dialect with a name. The name must be a string or a Unicode object.
- **csv.writer -** It writes the data to a csv file
- **o csv.unregister_dialect -** It deletes the dialect which is associated with the name from the dialect registry. If a name is not a registered dialect name, then an error is being raised.
- **csv.QUOTE_ALL -** It instructs the writer objects to quote all fields. csv.QUOTE_MINIMAL - It instructs the writer objects to quote only those fields which contain special characters such as quotechar, delimiter, etc.
- **csv.QUOTE_NONNUMERIC -** It instructs the writer objects to quote all the non-numeric fields.

- **csv.QUOTE_NONE -** It instructs the writer object never to quote the fields.

Reading CSV files

Python provides various functions to read csv file. We are describing few method of reading function.

- **Using csv.reader() function**

In Python, the **csv.reader()** module is used to read the csv file. It takes each row of the file and makes a list of all the columns.

We have taken a txt file named as python.txt that have default delimiter **comma(,)** with the following data:

```
name,department,birthday month
Parker,Accounting,November
Smith,IT,October
```

Example

```
import csv
with open('python.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
```

Output:

```
Column names are name, department, birthday month
  Parker works in the Accounting department, and was born in November.
  Smith works in the IT department, and was born in October.
Processed 3 lines.
```

In the above code, we have opened 'python.csv' using the **open()** function. We used **csv.reader()** function to read the file, that returns an iterable reader object. The **reader** object have consisted the data and we iterated using **for** loop to print the content of each row

Read a CSV into a Dictionar

We can also use **DictReader()** function to read the csv file directly into a dictionary rather than deal with a list of individual string elements.

Again, our input file, python.txt is as follows:

1. name,department,birthday month
2. Parker,Accounting,November
3. Smith,IT,October

Example

```
1. import csv
2. with open('python.txt', mode='r') as csv_file:
3.     csv_reader = csv.DictReader(csv_file)
4.     line_count = 0
5.     for row in csv_reader:
6.         if line_count == 0:
```

```
            print(f'The Column names are as follows {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department, and was born in {row["birthday month"]}.')

        line_count += 1
    print(f'Processed {line_count} lines.')
```

Output:

```
The Column names are as follows name, department, birthday month
   Parker works in the Accounting department, and was born in November.
   Smith works in the IT department, and was born in October.
Processed 3 lines.
```

Reading csv files with Pandas

The Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user.

Reading the csv file into a pandas **DataFrame** is quick and straight forward. We don't need to write enough lines of code to open, analyze, and read the csv file in pandas and it stores the data in **DataFrame**.

Here, we are taking a slightly more complicated file to read, called hrdata.csv, which contains data of company employees.

```
Name,Hire Date,Salary,Leaves Remaining
John Idle,08/15/14,50000.00,10
Smith Gilliam,04/07/15,65000.00,8
Parker Chapman,02/21/14,45000.00,10
```

5. Jones Palin,10/14/13,70000.00,3
6. Terry Gilliam,07/22/14,48000.00,7
7. Michael Palin,06/28/13,66000.00,8

Example

```
1. import pandas
2. df = pandas.read_csv('hrdata.csv')
3. print(df)
```

In the above code, the three lines are enough to read the file, and only one of them is doing the actual work, i.e., pandas.read_csv()

Output:

```
          Name                Hire Date     Salary      Leaves Remaining
0     John Idle               03/15/14      50000.0       10
1     Smith Gilliam           06/01/15      65000.0       8
2     Parker Chapman          05/12/14      45000.0       10
3     Jones Palin             11/01/13      70000.0       3
4     Terry Gilliam           08/12/14      48000.0       7

5     Michael Palin           05/23/13      66000.0       8
```

Python Write CSV File

CSV File

A CSV stands for "comma-separated values", which is defined as a simple file format that uses specific structuring to arrange tabular data. It stores tabular data such as spreadsheet or database in plain text and has a standard format for data interchange. The CSV file opens into the excel sheet, and the rows and columns data define the standard format.

Python CSV Module Functions

The CSV module work is to handle the CSV files to read/write and get data from specified columns. There are different types of CSV functions, which are as follows:

- **csv.field_size_limit -** It returns the current maximum field size allowed by the parser.
- **csv.get_dialect -** Returns the dialect associated with a name.
- **csv.list_dialects -** Returns the names of all registered dialects.
- **csv.reader -** Read the data from a CSV file
- **csv.register_dialect -** It associates dialect with a name, and name must be a string or a Unicode object.
- **csv.writer -** Write the data to a CSV file
- **csv.unregister_dialect -** It deletes the dialect, which is associated with the name from the dialect registry. If a name is not a registered dialect name, then an error is being raised.
- **csv.QUOTE_ALL -** It instructs the writer objects to quote all fields.
- **csv.QUOTE_MINIMAL -** It instructs the writer objects to quote only those fields which contain special characters such as quotechar, delimiter, etc.
- **csv.QUOTE_NONNUMERIC -** It instructs the writer objects to quote all the non-numeric fields.
- **csv.QUOTE_NONE -** It instructs the writer object never to quote the fields.

Writing CSV Files

We can also write any new and existing CSV files in Python by using the csv.writer() module. It is similar to the csv.reader() module and also has two methods, i.e., **writer** function or the **Dict Writer** class.

It presents two functions, i.e., **writerow()** and **writerows()**. The **writerow()** function only write one row, and the **writerows()** function write more than one row.
Dialects

It is defined as a construct that allows you to create, store, and re-use various formatting parameters. It supports several attributes; the most frequently used are:

- **Dialect.delimiter:** This attribute is used as the separating character between the fields. The default value is a comma (,).
- **Dialect.quotechar:** This attribute is used to quote fields that contain special characters.
- **Dialect.lineterminator:** It is used to create new lines, and the default value is '\r\n'.

Let's write the following data to a CSV File.

```
data = [{'Rank': 'B', 'first_name': 'Parker', 'last_name': 'Brian'},
{'Rank': 'A', 'first_name': 'Smith', 'last_name': 'Rodriguez'},
{'Rank': 'C', 'first_name': 'Tom', 'last_name': 'smith'},
{'Rank': 'B', 'first_name': 'Jane', 'last_name': 'Oscar'},
{'Rank': 'A', 'first_name': 'Alex', 'last_name': 'Tim'}]
```

Example -

```
import csv

with open('Python.csv', 'w') as csvfile:
    fieldnames = ['first_name', 'last_name', 'Rank']
```

```
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'Rank': 'B', 'first_name': 'Parker', 'last_name': 'Brian'})
    writer.writerow({'Rank': 'A', 'first_name': 'Smith',
                     'last_name': 'Rodriguez'})
    writer.writerow({'Rank': 'B', 'first_name': 'Jane', 'last_name': 'Oscar'})
    writer.writerow({'Rank': 'B', 'first_name': 'Jane', 'last_name': 'Loive'})

print("Writing complete")
```

Output:

```
Writing complete
```

It returns the file named as 'Python.csv' that contains the following data:

```
first_name,last_name,Rank
Parker,Brian,B
Smith,Rodriguez,A
Jane,Oscar,B
Jane,Loive,B
```

Write a CSV into a Dictionary

We can also use the class **DictWriter** to write the CSV file directly into a dictionary.

A file named as python.csv contains the following data:

```
Parker, Accounting, November
```

```
Smith, IT, October
```

Example -

```
1. import csv
2. with open('python.csv', mode='w') as csv_file:
3.     fieldnames = ['emp_name', 'dept', 'birth_month']
4.     writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
5.     writer.writeheader()
6.     writer.writerow({'emp_name': 'Parker', 'dept': 'Accounting', 'birth_month': 'November'})
7.     writer.writerow({'emp_name': 'Smith', 'dept': 'IT', 'birth_month': 'October'})
```

Output:

```
emp_name,dept,birth_month
Parker,Accounting,November
Smith,IT,October
```

Writing CSV Files Using Pandas

Pandas is defined as an open source library which is built on the top of Numpy library. It provides fast analysis, data cleaning and preparation of the data for the user.

It is as easy as reading the CSV file using pandas. You need to create the DataFrame, which is a two-dimensional, heterogeneous tabular data structure and consists of three main components- data, columns, and rows. Here, we take a slightly more complicated file to read, called hrdata.csv, which contains data of company employees.

```
1. Name,Hire Date,Salary,Leaves Remaining
2. John Idle,08/15/14,50000.00,10
3. Smith Gilliam,04/07/15,65000.00,8
4. Parker Chapman,02/21/14,45000.00,10
```

5. Jones Palin,10/14/13,70000.00,3
6. Terry Gilliam,07/22/14,48000.00,7
7. Michael Palin,06/28/13,66000.00,8

Example -

```
1. import pandas
2. df = pandas.read_csv('hrdata.csv',
3.         index_col='Employee',
4.         parse_dates=['Hired'],
5.         header=0,
6.         names=['Employee', 'Hired', 'Salary', 'Sick Days'])
7. df.to_csv('hrdata_modified.csv')
```

Output:

```
Employee, Hired, Salary, Sick Days
John Idle, 2014-03-15, 50000.0,10
Smith Gilliam, 2015-06-01, 65000.0,8
Parker Chapman, 2014-05-12, 45000.0,10
Jones Palin, 2013-11-01, 70000.0,3
Terry Gilliam, 2014-08-12 , 48000.0,7
Michael Palin, 2013-05-23, 66000.0,8
```

Python read excel file

Excel is a spreadsheet application which is developed by Microsoft. It is an easily accessible tool to organize, analyze, and store the data in tables. It is widely used in many different applications all over the world. From Analysts to CEOs, various professionals use Excel for both quick stats and serious data crunching.

Excel Documents

An Excel spreadsheet document is called a workbook which is saved in a file with **.xlsx** extension. The first row of the spreadsheet is mainly reserved for the header, while the first column identifies the sampling unit. Each workbook can contain multiple sheets that are also called a worksheets. A box at a particular column and row is called a cell, and each cell can include a number or text value. The grid of cells with data forms a sheet.

The active sheet is defined as a sheet in which the user is currently viewing or last viewed before closing Excel.

Reading from an Excel file

First, you need to write a command to install the **xlrd** module.

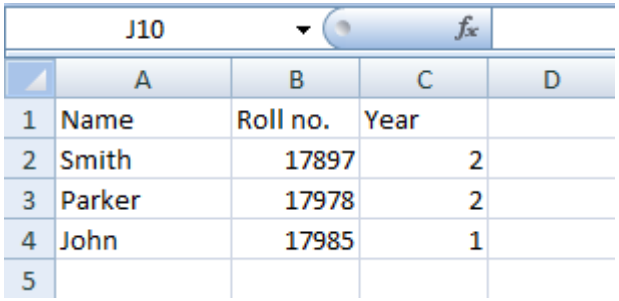
1. pip install xlrd

Creating a Workbook

A workbook contains all the data in the excel file. You can create a new workbook from scratch, or you can easily create a workbook from the excel file that already exists.

Input File

We have taken the snapshot of the workbook.



	A	B	C	D
1	Name	Roll no.	Year	
2	Smith	17897	2	
3	Parker	17978	2	
4	John	17985	1	
5				

Code

```
# Import the xlrd module
import xlrd

# Define the location of the file
loc = ("path of file")

# To open the Workbook
wb = xlrd.open_workbook(loc)
sheet = wb.sheet_by_index(0)

# For row 0 and column 0
sheet.cell_value(0, 0)
```

Explanation: In the above example, firstly, we have imported the xlrd module and defined the location of the file. Then we have opened the workbook from the excel file that already exists.

Reading from the Pandas

Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user and supports both xls and xlsx extensions from the URL.

It is a python package which provides a beneficial data structure called a data frame.

Example

```
Example -
import pandas as pd

# Read the file
data = pd.read_csv(".csv", low_memory=False)

# Output the number of rows
print("Total rows: {0}".format(len(data)))

# See which headers are available
print(list(data))
```

Reading from the openpyxl

First, we need to install an openpyxl module using pip from the command line.

```
pip install openpyxl
```

After that, we need to import the module.

We can also read data from the existing spreadsheet using openpyxl. It also allows the user to perform calculations and add content that was not part of the original dataset.

Example

```
import openpyxl
my_wb = openpyxl.Workbook()
```

3. my_sheet = my_wb.active
4. my_sheet_title = my_sheet.title
5. **print**("My sheet title: " + my_sheet_title)

Output:

```
My sheet title: Sheet
```

Python Write Excel File

The Python write excel file is used to perform the multiple operations on a spreadsheet using the **xlwt** module. It is an ideal way to write data and format information to files with .xls extension.

If you want to write data to any file and don't want to go through the trouble of doing everything by yourself, then you can use a for loop to automate the whole process a little bit.

Write Excel File Using xlsxwriter Module

We can also write the excel file using the **xlsxwriter** module. It is defined as a Python module for writing the files in the XLSX file format. It can also be used to write text, numbers, and formulas to multiple worksheets. Also, it supports features such as charts, formatting, images, page setup, auto filters, conditional formatting, and many others.

We need to use the following command to install xlsxwriter module:

1. pip install xlsxwriter

> *Note- Throughout XlsxWriter, rows, and columns are zero-indexed. The first cell in a worksheet is listed as, A1 is (0,0), B1 is (0,1), A2 is (1,0), B2 is (1,1).....,and so on.*

Write Excel File Using openpyxl Module

It is defined as a package which is generally recommended if you want to read and write .xlsx, xlsm, xltx, and xltm files. You can check it by running **type(wb)**.

The load_workbook() function takes an argument and returns a workbook object, which represents the file. Make sure that you are in the same directory where your spreadsheet is located. Otherwise, you will get an error while importing.

You can easily use a for loop with the help of the range() function to help you to print out the values of the rows that have values in column 2. If those particular cells are empty, you will get None.

Writing data to Excel files with xlwt

You can use the xlwt package, apart from the XlsxWriter package to create the spreadsheets that contain your data. It is an alternative package for writing data, formatting information, etc. and ideal for writing the data and format information to files with .xls extension. It can perform multiple operations on the spreadsheet.

It supports features such as formatting, images, charts, page setup, auto filters, conditional formatting, and many others.

Pandas have excellent methods for reading all kinds of data from excel files. We can also import the results back to pandas.

Writing Files with pyexcel

You can easily export your arrays back to a spreadsheet by using the save_as() function and pass the array and name of the destination file to the dest_file_name argument.

It allows us to specify the delimiter and add dest_delimiter argument. You can pass the symbol that you want to use as a delimiter in-between " ".

Code

```
# import xlsxwriter module
import xlsxwriter

book = xlsxwriter.Book('Example2.xlsx')
sheet = book.add_sheet()

# Rows and columns are zero indexed.
row = 0
column = 0

content = ["Parker", "Smith", "John"]
```

```
12.
13. # iterating through the content list
14. for item in content :
15.
16.     # write operation perform
17.     sheet.write(row, column, item)
18.
19.     # incrementing the value of row by one with each iterations.
20.     row += 1
21.
22. book.close()
```

Output: