PYTHON PROGRAMMING

BY
PRUTHVI RAJ A
ASST.PROFESSOR
DEPT OF COMPUTER
SCIENCE

INTRODUCTION TO PYTHON UNIT -1

Introduction :

- Features and Applications of Python;
- Python Versions;
- Installation of Python;
- Python Command Line mode and Python IDEs;
- Simple Python Program.

Python Basics:

- Identifiers;
- Keywords;
- Statements and Expressions;
- Variables;
- Operators;
- Precedence and Association;
- Data Types;
- Indentation;
- Comments;
- Built-in Functions- Console Input and Console Output,
- Type Conversions;
- Python Libraries;
- Importing Libraries with Examples.
- Python Control Flow:
- **Types of Control Flow:**
  - Control Flow Statements- if, else, elif,
  - while loop,
  - break, continue statements,
  - for loop Statement;
  - range () and exit () functions

PYTHON PROGRAMMING

Very beginning learn about

What?

Where?

Why?

What is Python?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
- OR
- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics developed by Guido van Rossum.
- It was originally released in 1991. Designed to be easy as well as fun, the name "Python" is a nod to the British comedy group Monty Python.
- Python has a reputation as a beginner-friendly language, replacing Java as the most widely used introductory language because it handles much of the complexity for the user, allowing beginners to focus on fully grasping programming concepts rather than minute details.

Where It is used for?

- web development (server-side),
- software development,
- mathematics,
- system scripting.
- Processing big data
- Fast prototyping
- Developing production-ready software

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Things to know

- The most recent major version of Python is Python 3, which we shall be using .
- However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- Python will be written in a text editor.
- It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

- Example
- print("Hello, World!")
- **Python Install**
- Many PCs and Macs will have python already installed.
- To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python –version
```

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: https://www.python.org/

Verify installation

- Once IDLE is launched, go to →file menu
- Click on → new file to create new python file.

How to use IDLE

- Print(“morning”)
- Save the file with .py extension and run the file(run menu→ click on run module or press F5)
- If the output is morning then installation is successful and IDLE is ready to use.

Python Quickstart

- Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.
- The way to run a python file is like this on the command line:
- C:\Users\panka>AppData\Local\Programs\Python\Python311\my.py
- Where “my.py" is the name of your python file.
- Let's write our first Python file, called my.py, which can be done in any text editor.

- **The Python Command Line**
- To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.
- Type the following on the Windows, Mac or Linux command line:
- **C:\Users\Your Name>python**
- Or, if the "python" command did not work, you can try "py":
- **C:\Users\Your Name>py**
- From there you can write any python, including our hello world example from earlier in the tutorial:
- **>>> print("Hello, World!")**
- Which will write "Hello, World!" in the command line:
- **Hello, World!**
- Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:
-
- **>>>exit()**

- **Features and Benefits of Python**
  - Compatible with a variety of platforms including Windows, Mac, Linux, Raspberry Pi, and others
  - Uses a simple syntax comparable to the English language that lets developers use fewer lines than other programming languages
  - Operates on an interpreter system that allows code to be executed immediately, fast-tracking prototyping
  - Can be handled in a procedural, object-orientated, or functional way
- **Python Identifiers**
- Identifiers are the name given to variables, classes, methods, functions etc. For example

```
language = 'Python'
```

- Here, language is a variable (an identifier) which holds the value 'Python'.
- We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,

```
continue = 'Python'
```

- The above code is wrong because we have used continue as a variable name. (CONTINUE IS KEYWORD)

Rules for Naming an Identifier:

- Identifiers cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter or _. The first letter of an identifier cannot be a digit.
- It's a convention to start an identi _.
- Whitespaces are not allowed.
- We cannot use special symbols lik

Some Valid and Invalid Identifiers i

GFGC

Valid Identifiers	Invalid Identifiers
_score	@core
return_value	return
highest_score	highest score
name1	1name
convert_to_strin g	convert to_string

Things to Remember

- Python is a case-sensitive language. This means, Variable and variable are not the same.
- Always give the identifiers a name that makes sense.
- While c = 10 is a valid name, writing count = 10 would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.
- Multiple words can be separated using an underscore, like this_is_a_long_variable.

Govt. First Grade College, Malur

Department of Computer Science

PYTHON PROGRAMMING

Unit 1

Prepared By Pruthvi Raj

What is Python?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
- OR
- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics developed by Guido van Rossum.
- It was originally released in 1991. Designed to be easy as well as fun, the name "Python" is a nod to the British comedy group Monty Python.
- Python has a reputation as a beginner-friendly language, replacing Java as the most widely used introductory language because it handles much of the complexity for the user, allowing beginners to focus on fully grasping programming concepts rather than minute details.

Where It is used for?

- web development (server-side),
- software development,
- mathematics,
- system scripting.
- Processing big data
- Fast prototyping
- Developing production-ready software

Prepared By Pruthvi Raj

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python History and Versions

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.

Python 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) 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.

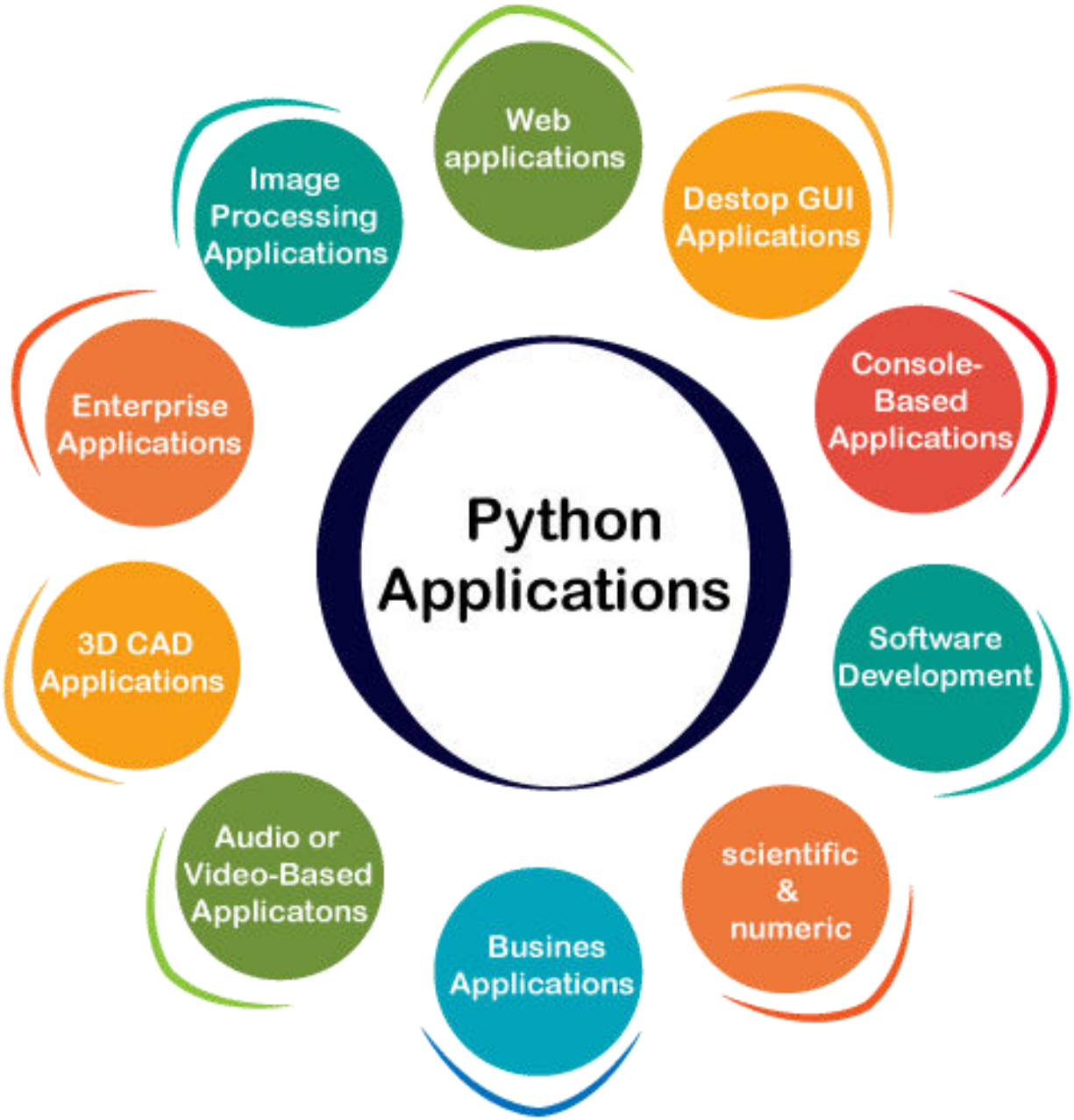
8) GUI Programming Support

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

9) 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.

Python Applications



Prepared By Pruthvi Raj

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.

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.

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.

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.

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.

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.

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 AnyCAD.

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

Python Install

- Many PCs and Macs will have python already installed.
- To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python –version
```

If you find that you do not have Python installed on your computer, then you can download it for free from the following website: https://www.python.org/

- **Verify installation**
  - Once IDLE is launched, go to →file menu
  - Click on → new file to create new python file.

First Python Program

- Python provides us the two ways to run a program:

1. Using Interactive interpreter prompt
2. Using a script file

Example Program

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

Python Basics

- **PYTHON CHARACTER SET**

  Character set is a set of valid characters that a language can recognize. A character represents any letter, digit or any other symbol. Python supports Unicode encoding standard. That means, Python has the following character set :

Letters	A-Z, a-z
Digits	0-9
Special symbols	space +-/*\() [] {} // = != == <, > . '"""",;:%!& # <= >= @_(underscore)
Whitespaces	Blank space, tabs (→), carriage return (), newline, formfeed.
Other characters characters as	Python can process all ASCII and Unicode part of data or literals

Identifiers

Identifiers are fundamental building blocks of a program and are used as the general terminology for the names given to different parts of the program viz. variables, objects, classes, functions, lists, dictionaries etc. Identifier forming rules of Python are being specified below:

- An identifier is an arbitrarily long sequence of letters and digits.
- The first character must be a letter; the underscore (_) counts as a letter.
- Upper and lower-case letters are different. All characters are significant.
- The digits 0 through 9 can be part of the identifier except for the first character.
- Identifiers are unlimited in length. Case is significant i.e., Python is case sensitive as it treats upper and lower-case characters differently.
- An identifier cannot contain any special character except for underscore (_).
- An identifier must not be a keyword of python.

Valid Identifiers	**Invalid Identifiers**
_score	@core
return_value	return
highest_score	highest score
name1	1name
convert_to_string	convert to_string
MY_BOOK	MY book

KEYWORDS IN PYTHON

- Keywords are the reserved words in Python. We cannot use a keyword as a variable name, function name or any other identifier.
- Here's a list of all keywords in Python Programming

Keywords in Python programming language

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Statements in Python

A **statement is an instruction that a Python interpreter can execute**. So, in simple words, we can say anything written in Python is a statement.

For example, a = 10 is an assignment statement.

where a is a variable name and 10 is its value.

There are mainly four types of statements in Python

- Print statements,
- Assignment statements,
- Conditional statements,
- Looping statements,
- Return statements.

Multi-Line Statements

Python statement ends with the token NEWLINE character.

But we can extend the statement over multiple lines using line continuation character (\). This is known as an **explicit continuation**.

- Example

```
>>addition = 10 + 20 + \
            30 + 40 + \
            50 + 60 + 70
>>print(addition)
 # Output: 280
```

- **Implicit continuation:**

We can use parentheses () to write a multi-line statement.

We can add a line continuation statement inside it.

Whatever we add inside a parentheses () will treat as a single statement even it is placed on multiple lines.

Example:

```
addition = (10 + 20 + 30 + 40 +
50 + 60 + 70)
 print(addition)
 # Output: 280
```

EXPRESSION STATEMENTS

- Expression statements are used to compute and write a value. An expression statement evaluates the expression list and calculates the value.
- An expression is a combination of values, variables, and operators.
- A single value all by itself is considered an expression. Following are all legal expressions (assuming that the variable x has been assigned a value):
- `x + 20`
- So here x + 20 is the expression statement which computes the final value if we assume variable x has been assigned a value (10). So final value of the expression will become 30.
- **Note : "But in a script, an expression all by itself doesn't do anything! So we mostly assign an expression to a variable, which becomes a statement for an interpreter to execute".**

PYTHON VARIABLES

- Variables are identifiers of a physical memory location, which is used to hold values temporarily during program execution.
- Python interpreter allocates memory based on the values(letter or a number) data type of variable, different data types like integers, decimals, characters, etc. can be stored in these variables.

Python Variable Declaration

- In Python, like many other programming languages, there is no need to define variables in advance.
- As soon as a value is assigned to a variable, it is automatically declared. **This is why Python is called a dynamically typed language.**

The syntax for creating variables in Python is given below:

```
<variable-name> = <value>
```

Assigning value to Variables

- Python interpreter can determine what type of data is stored, so before assigning a value, variables do not need to be declared.
- Usually, in all programming languages, equal sign = is used to assign values to a variable. It assigns the values of the right side operand to the left side operand.
- The left side operand of = operator is the name of a variable, and the right side operand is value.
- **Example:**

```
name = "Packing box" # A string
height = 10 # An integer assignment
width = 20.5 # A floating point
print (name)
print (height)
print (width)
```

OUTPUT

Packing box
10
20.5

Common Rules for Naming Variables in Python

Python has some variable related rules that must be followed:

- Variable names are case-sensitive.
- Variable names must begin with a letter or underscore.
- A variable name can only contain alphanumeric characters and underscore, such as (a-z, A-Z, 0-9, and _).
- A variable name can not contents space.
- Reserved words cannot be used as a variable name.

Example

This will create two variables:

```
a = 4
A = "Sally"
#A will not overwrite a
```

Data Types

Python offers following built-in core data types :

1. **Numbers** - immutable(cannot be changed/modified)
2. **String -** immutable(cannot be changed/modified)
3. **List** - mutable(changed/modified)
4. **Tuple -** immutable(cannot be changed/modified**)**
5. **Dictionary** - mutable(changed/modified)

1. Numbers

Numeric values are stored in numbers. The whole number, float, and complex qualities have a place with a Python Numbers data type. Python offers the type() function to determine a variable's data type. The instance () capability is utilized to check whether an item has a place with a specific class.

Example

```
a = 5
print("The type of a", type(a))
b = 40.5
print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
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 kinds of numerical data.

- **Int:** Whole number worth can be any length, like numbers 10, 2, 29, - 20, - 150, and so on. An integer can be any length you want in Python. Its worth has a place with int.
- **Float:** Float stores drifting point numbers like 1.9, 9.902, 15.2, etc. It can be accurate to within 15 decimal places.
- **Complex:** An intricate number is in the form A+Bi where i is the imaginary number, equal to the sqare root of -1 i.e., $\sqrt{-1}$, that is 1^2 The complex numbers like 2.14j, 2.0 + 2.3j, etc.

2. Sequence Type or String type

- The sequence of characters in the quotation marks can be used to describe the string. A string can be defined in Python using single, double, or triple quotes.
- String dealing with Python is a direct undertaking since Python gives worked-in capabilities and administrators to perform tasks in the string.
- When dealing with strings, the operation "hello"+" python" returns "hello python," and the operator + is used to combine two strings.
- Because the operation "Python" *2 returns "Python," the operator * is referred to as a repetition operator.
- Example

```
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
```

3. List

- Lists in Python are like arrays in C, but lists can contain data of different types. The things put away in the rundown are isolated with a comma (,) and encased inside square sections [].
- To gain access to the list's data, we can use slice [:] operators. Like how they worked with strings, the list is handled by the concatenation operator (+) and the repetition operator (*).
- 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)
```

Out put

```
[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]
```

4. TUPLE

- In many ways, a tuple is like a list. Tuples, like lists, also contain a collection of items from various data types. A parenthetical space () separates the tuple's components from one another.
- Because we cannot alter the size or value of the items in a tuple, it is a read-only data structure.
- Example

```
up  = ("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"
```

Out put

```
<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
```

5. Dictionary

- A dictionary is a key-value pair set arranged in any order. It stores a specific value for each key, like an associative array or a hash table. Value is any Python object, while the key can hold any primitive data type.
- The comma (,) and the curly braces are used to separate the items in the dictionary.
- 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())
```

Out Put

```
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'])
```


6. Sets

- The data type's unordered collection is Python Set. It is iterable, mutable(can change after creation), and has remarkable components. The elements of a set have no set order; It might return the element's altered sequence. Either a sequence of elements is passed through the curly braces and separated by a comma to create the set or the built-in function set() is used to create the set. It can contain different kinds of values.

Prepared By Pruthvi Raj

EXAMPLE

```
# Creating Empty set
set1 = set()

set2 = {'James', 2, 3,'Python'}

#Printing Set value
print(set2)

# Adding element to the set

set2.add(10)
print(set2)

#Removing element from the set
set2.remove(2)
print(set2)
```

Out Put

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


Python – Operators

- Python operators are the constructs which can manipulate the value of operands. These are symbols used for the purpose of logical, arithmetic and various other operations.
- Consider the expression 4 + 5 = 9. Here, 4 and 5 are called **operands** and + is called **operator**. In this tutorial, we will study different types of Python operators.
- **Types of Python Operators**
- Python language supports the following types of operators.
- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

- Python arithmetic operators are used to perform mathematical operations on numerical values. These operations are Addition, Subtraction, Multiplication, Division, Modulus, Exponents and Floor Division.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 10, b = 10 => a+b = 20
- (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 = 5 => a - b = 15
/ (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 = 4 => a * b = 80
% (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)**	As it calculates the first operand's power to the second operand, it is an exponent operator.
// (Floor division)	It provides the quotient's floor value, which is obtained by dividing the two operands.

Prepared By Pruthvi Raj

Python Comparison Operators

- Python comparison operators compare the values on either sides of them and decide the relation among them. They are also called relational operators. These operators are equal, not equal, greater than, less than, greater than or equal to and less than or equal to.

Operator	Name	Example
==	Equal	4 == 5 is not true.
!=	Not Equal	4 != 5 is true.
>	Greater Than	4 > 5 is not true.
<	Less Than	4 < 5 is true.
>=	Greater than or Equal to	4 >= 5 is not true.
<=	Less than or Equal to	4 <= 5 is true.

Python Assignment Operators

- Python assignment operators are used to assign values to variables. These operators include simple assignment operator, addition assign, subtraction assign, multiplication assign, division and assign operators etc.

Operator	Name	Example
=	Assignment Operator	a = 10
+=	Addition Assignment	a += 5 (Same as a = a + 5)
-=	Subtraction Assignment	a -= 5 (Same as a = a - 5)
*=	Multiplication Assignment	a *= 5 (Same as a = a * 5)
/=	Division Assignment	a /= 5 (Same as a = a / 5)
%=	Remainder Assignment	a %= 5 (Same as a = a % 5)
**=	Exponent Assignment	a **= 2 (Same as a = a ** 2)
//=	Floor Division Assignment	a //= 3 (Same as a = a // 3)

Python Bitwise Operators

- Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two vari

Operator	Name	Example
&	Binary AND	Sets each bit to 1 if both bits are 1
|	Binary OR	Sets each bit to 1 if one of two bits is 1
^	Binary XOR	Sets each bit to 1 if only one of two bits is 1
~	Binary Ones Complement	Inverts all the bits
<<	Binary Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Binary Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Python Logical Operators

- The assessment of expressions to make decisions typically makes use of the logical operators. The following logical operators are supported by Python.

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Python Membership Operators

- Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

Operator	Example	Description
in	5 in {1, 2, 3, 4, 5}	It returns true if it finds the corresponding value in a specified sequence and false otherwise.
not in	5 not in {1, 2, 3, 4}	It returns true if it does not find the corresponding value in a given sequence and true otherwise.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −**Types of Identity Operators in Python**

Identity Operator in Python has two types :

1. "is" operator
2. "is not" operator

is Operator

is operator is a positive equivalence checking operator. It is used to determine if two variables refer to an identical memory location or

not.

is operator returns True if both operands have the same unique id (points to the same memory location).

Example : The Use of "is" Identity Operator in Python

is not Operator

is not operator is a negative equivalence checking operator. It is used to determine if two variables refer to a distinct memory location or not.

is not operator returns True if both operands are stored at different memory locations.

Example : The Use of "is not" Identity Operator in python

Write a python program to demonstrate “is not operator”

Find the difference between == and “is operator”

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

Types of Comments in Python

- Single-Line Comments
- Multi-Line Comments

Single-Line Comments

- A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation.
- **example**
- # This code is to show an example of a single-line comment
- **print**('This statement does not have a hashtag before it')

Multi-Line Comments

Python does not provide the facility for multi-line comments. However, there are indeed many ways to create multi-line comments.

With Multiple Hashtags (#)

example

```
# it is a
# comment
# extending to multiple lines
```

Using String Literals

Because Python overlooks string expressions that aren't allocated to a variable, we can utilize them as comments.

example

‘it is a comment extending to multiple lines’

add a multiline string (triple quotes) in your code, and place your comment inside it:

```
"""
    This is a comment
    written in
    more than just one line
    """
    print("Hello, World!")
```


Console input in Python

- The **interactive** shell in *Python* is treated as a **console**. We can take the user entered data the **console** using **input()** function.

```
# using input() to take user input
num = input('Enter a number: ')
print('You Entered:', num)
 print('Data type of num:', type(num))
```

output

Enter a number: 34

You Entered: 34

Data type of num: <class 'str'>

To convert user input into a number we can use int() or fl

```
num = int(input('Enter a number: '))
```

Console output in Python

- In Python, we can simply use the print() function to print output. For

```
print('Python is powerful') # Output: Python is powerful
```

- Here, the print() function displays the string enclosed inside the single quotation.

Syntax of print()

- In the above code, the print() function is taking a single parameter.
- Here, the print() function accepts 5 parameters.

```
print(object= separator= end= file= flush=)
```

- Here,
- **object** - value(s) to be printed
- **sep** (optional) - allows us to separate multiple objects inside print().
- **end** (optional) - allows us to add add specific values like new line "\n", tab "\t"
- **file** (optional) - where the values are printed. It's default value is sys.stdout (screen)

EXAMPLE

Example 1: Python Print Statement

```
print('Good Morning!')
print('It is rainy today')
```

Output

```
Good Morning! It is rainy today
```

In the above example, the print() statement only includes the object to be printed. Here, the value for end is not used. Hence, it takes the default value '\n'.

So we get the output in two different lines.

Example 2: Python print() with end Parameter

```
# print with end whitespace print('Good Morning!', end= ' ') print('It is rainy today')
```

Output

```
Good Morning! It is rainy today
```

- Notice that we have included the end= ' ' after the end of the first print() statement.
- Hence, we get the output in a single line separated by space.

Type Conversion in Python

- Type conversion is the transformation of a Py type of data into another type of data.
- **In Python, there are two kinds of type conversion:**
- Explicit Type Conversion-The programmer must perform this task manually.
- Implicit Type Conversion-By the Python program automatically.

- **Python Explicit Type Conversion**
  - In Explicit Type Conversion, users convert the data type of an object to required data type.
  - We use the built-in functions like int(), float(), str(), etc to perform explicit type conversion.
  - This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example explicit conversion

```
num_string = '12'
num_integer = 23
print("Data type of num_string before Type
    Casting:",type(num_string))
# explicit type conversion
num_string = int(num_string)
print("Data type of num_string after Type
    Casting:",type(num_string))
num_sum = num_integer + num_string
print("Sum:",num_sum)
                                          m_sum))
```

Output

```
Data type of num_string before Type Casting: <class 'str'>
Data type of num_string after Type Casting: <class 'int'>
Sum: 35
Data type of num_sum: <class 'int'>
```

Python Implicit Type Conversion

In certain situations, Python automatically converts one data type to another. This is known as implicit type conversion.

Example 1: Converting integer to float

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

```
integer_number = 123
float_number = 1.23
new_number = integer_number + float_number # display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))
```

Out put

```
Value: 124.23
Data Type: <class 'float'>
```

Python Libraries

What are Python Libraries?

- Python Libraries are collections of python modules that provide several utilities
- Python Libraries are written in many programming languages
- Python libraries are a collection of utility methods, classes and modules that can be used in a python application directly.
- These libraries provide pre-written methods and classes for use within python applications.
- Their use may range from anything from Input/Output to data manipulation and visualization etc.
- Python libraries may be written in a number of programming languages including C, Python, as well as Java in case of Jpython, etc.
- Python libraries provide code bundles that can be used repeatedly within a python application.

Some Important Python Libraries

In the following section, we shall look into a few of the important python libraries that are frequently in use and provide essential utilities.

Matplotlib

- The matplotlib library is used in data visualization in python and is an important tool in Data Science.
- Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- It can be used by installing the matplotlib library and importing the neccessary methods using the import keyword.
- It consis), scatter() bar() stem() st

Pandas

- The Pandas or Python Data Analysis Library is another important tool in Data Sciences and provides utilities that help in Data Analysis.
- pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
- Pandas is built on top of NumPy. It also provides a huge number of functions and can be accessed by installing the pandas library.

```
import pandas as pd
import numpy as np
```

```
s = pd.Series ([10.8,10.4,10.3,7.4,0.25],
               index = ['VW','Toyota','Renault','Kia','Tesla'])
```

```
s
```

```
VW         10.80
Toyota     10.40
Renault    10.30
Kia         7.40
Tesla       0.25
dtype: float64
```

NumPy

- NumPy is used for scientific calculations and it provides utilities that help in executing large mathematical

calculations and it supports big matrices and multidimensional data.

- NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier

transforms, and more.

- NumPy supports a wide range of hardware and computing platforms, and plays well with distributed,GPU, and sparse array

libraries.

- It can

```
import numpy as np
a = np.array([0, 1, 2, 3])        # Create a rank 1 array
print(a)                          #print array a
print(type(a))                    #type of array a
print(a.ndim)                     #dimension of array a
print(a.shape)                    #shape(row,column) of array a
print(len(a))                     #length of array a
```

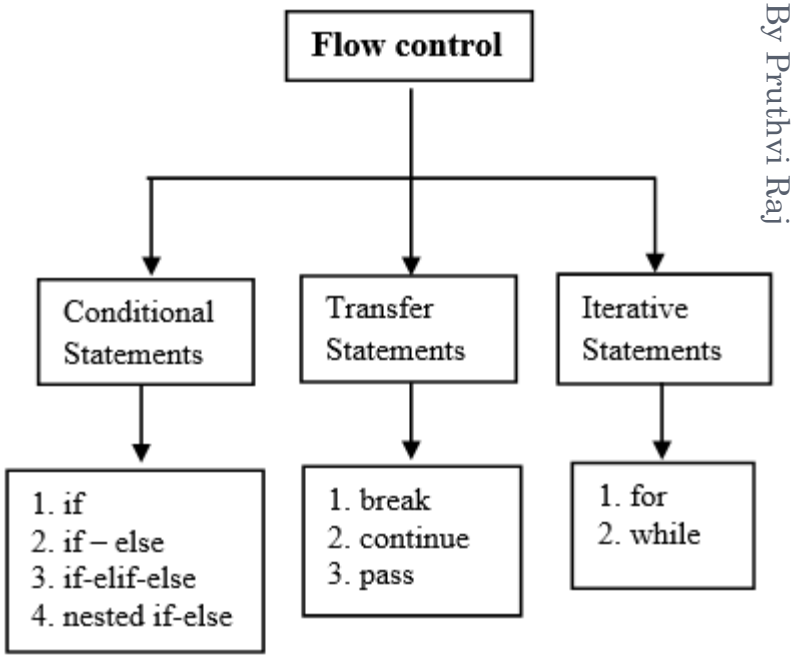
```
[0 1 2 3]
<class 'numpy.ndarray'>
1
(4,)
4
```

CONTROL FLOW STATEMENTS

Loops are employed in Python to iterate over a section of code continually. Control statements are designed to serve the purpose of modifying a loop's execution from its default behavior. Based on a condition, control statements are applied to alter how the loop executes.

Types of Flow Control in Python

1. Conditional statements
2. Iterative statements.
3. Transfer statements



1. Conditional statements

In Python, condition statements act depending on whether a given condition is true or false. You can execute different blocks of codes depending on the outcome of a condition. Condition statements always evaluate to either True or False.

There are three types of conditional statements.

1. if statement
2. if-else
3. if-elif-else
4. nested if-else

If statement in Python

- In control statements, The if statement is the simplest form. It takes a condition and evaluates to either True or False.
- If the condition is True, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and The controller moves to the next line.
- **Syntax of the if statement**

```
if condition:
    statement 1
    statement 2
    statement n
```

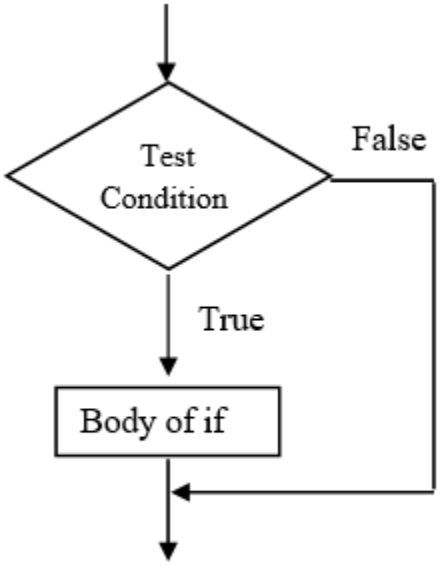


Fig. Flowchart of if statement

Let’s see the example of the if statement. In this example, we will calculate the square of a number if it greater than 5

Example

```
number = 6
if number > 5:
# Calculate square
   print(number * number)
print('Next lines of code')
```

Output

```
36
Next lines of code
```


If – else statement

- The if-else statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.
- **Syntax of the if-else sta**

```
if condition:
statement 1 else:
statement 2
```

- if the condition is True, then statement 1 will be executed If the condition is False, statement 2 will be executed. See the following flowchart for more detail.

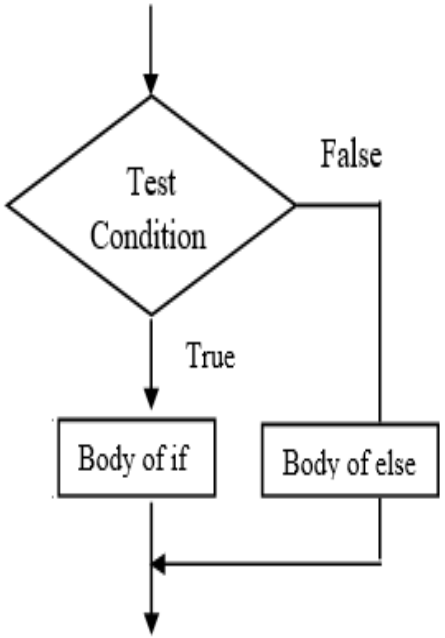


Fig. Flowchart of if-else

- Example
- password = input('Enter password ‘)
- if password == “ncms":
    print("Correct password")
  else:
   print("Incorrect Password")

Output 1:
Enter password ncms
Correct password
Output 2:
Enter password
PYnative Incorrect
Password

Prepared By Pruthvi Raj

if-elif-else in Python

- In Python, the if-elif-else condition statement has an elif blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.
- With the help of if-elif-else we can make a tricky decision. The elif statement checks multiple con e and if ulfills, th de.
- Sy -else sta

```
if condition-1:
    statement 1
elif condition-2:
    statement 2
elif condition-3:
    statement 3
    ...
else:
    statement
```

```
print("Please enter the values from 0 to 4")
x=int(input("Enter a number: "))
if x==0:
print("You entered:", x)
elif x==1:
print("You entered:", x)
elif x==2:
print("You entered:", x)
elif x==3:
print("You entered:", x)
elif x==4:
print("You entered:", x)
else:
print("Beyond the range than specified")
```

Nested if-else statement

- In Python, the nested if-else statement is an if statement inside another if-else statement. It is allowed in Python to put any number of if statements in another if statement.
- Indentation is the only way to differentiate the level of nesting. The nested if-else is useful when we want to make a series of decisions.
- S… …se:

```
if conditon_outer:
    if condition_inner:
        statement of inner if
    else:
        statement of inner
else:
    statement ot outer if
else:
    Outer else
statement outside if block
```

Example: Find a greater number between two numbers

```
num1 = int(input('Enter first number '))
num2 = int(input('Enter second number '))
if num1 >= num2:
    if num1 == num2:
      print(num1, 'and', num2, 'are equal')
    else:
        print(num1, 'is greater than', num2)
    else:
        print(num1, 'is smaller than', num2)
```

2. Iterative statements.

- In computer programming, loops are used to repeat a block of code.
- For example, if we want to show a message **100** times, then we can use a loop. It's just a simple example; you can achieve much more with loops.
- **There are 2 types of loops in Python:**
- for loop
- while loop

Python for Loop

- In Python, a for loop is used to iterate over sequences such as lists, tuples, string, etc. For example,

```
languages = ['Swift', 'Python', 'Go', 'JavaScript']
# run a loop for each item of the list
for language in languages:
print(language)
```

Output

Swift
Python
Go
JavaScript

- In the above example, we have created a list called languages.
- Initially, the value of language is set to the first element of the array, i.e. Swift, so the print statement inside the loop is executed.
- language is updated with the next element of the list, and the print statement is executed again. This way, the loop runs until the last element of the list is accessed.

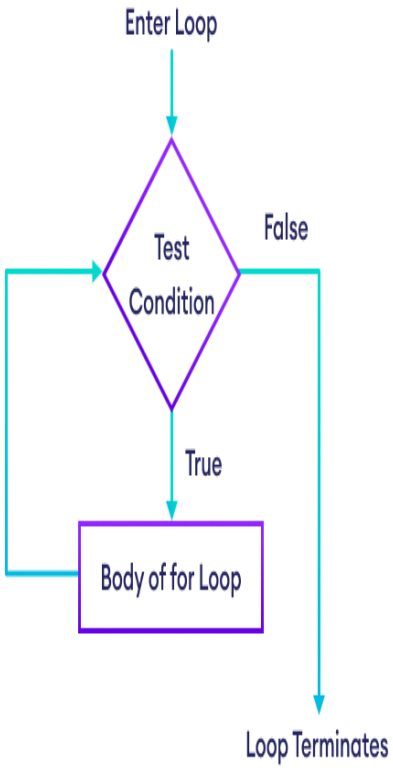
for Loop Syntax

- The syntax of a for loop is:

```
for val in sequence:
# statement(s)
```

- Here, **val** accesses each item of **sequence** on each
- iteration.
- The loop continues until we reach the last item in the sequence.

Flowchart of Python for Loop



- **Example: Loop Through a String**
- for x in 'Python':
- print(x)
- **Output**
- P
- y
- t
- h
- o
- N
- **Using a for Loop Without Accessing Items**
- It is not mandatory to use items of a sequence within a for loop. For example
- languages = ['Swift', 'Python', 'Go']
- for language in languages:
- print('Hello')
- print('Hi')
- **Output**
- Hello
- Hi
- Hello
- Hi
- Hello
- Hi
- Here, the loop runs three times because our list has three items. In each iteration, the loop body prints 'Hello' and 'Hi'. The items of the list are not used within the loop.

Python while Loop

- In programming, loops are used to repeat a block of code. For example, if we want to show a message 100 times, then we can use a loop. It's just a simple example, we can achieve much more with loops.

Python while loop is used to run a block code until a certain condition is met.

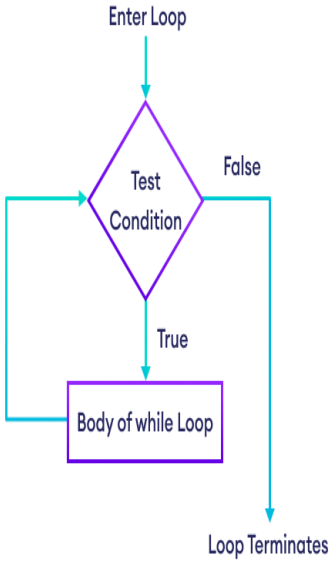
The syntax of while loop is:

```
while condition: # body of while loop
```

- Here,

1. A while loop evaluates the condition
2. If the condition evaluates to True, the code inside the while loop is executed.
3. condition is evaluated again.
4. This process continues until the condition is False.
5. When condition evaluates to False, the loop stops.

Flowchart of Python while Loop



- **Example: Python program to display numbers from 1 to 5 using while Loop**
- **# initialize the variable**
- **i = 1**
- **n = 5**
- **# while loop from i = 1 to 5**
- **while i <= n:**
- **   print(i)**
- **   i = i + 1**
- **Output**
- **1**
- **2**
- **3**
- **4**
- **5**

- **Infinite while Loop in Python**
- If the condition of a loop is always True, the loop runs for infinite times (until the memory is full). For example,

  age = 32
- **# the test condition is always True**
- **while** age > 18:
- **   print(**'You can vote')

- In the above example, the condition always evaluates to True. Hence, the loop body will run for infinite times.

- **Python range()**
- The range() function returns a sequence of numbers between the give range.

```
# create a sequence of numbers from 0 to 3
 numbers = range(4)
# iterating through the sequence of numbers
for i in numbers: print(i)
 Output:
 0
 1
 2
 3
```

- **Note:** range() returns an immutable sequence of numbers that can be easily converted to lists, tuples, sets etc.

- **Syntax of range()**
- The range() function can take a maximum of three arguments:

range(start, stop, step)

- The start and step parameters in range() are optional.
- Now, let's see how range() works with different number of arguments.

Example 1: range() with Stop Argument

If we pass a single argument to range(), it means we are passing the stop argument.

In this case, range() returns a sequence of numbers starting from 0 up to the number (but not including the number).

```
# numbers from 0 to 3 (4 is not included)
numbers = range(4)
print(list(numbers))
# [0, 1, 2, 3]

# if 0 or negative number is passed, we get an empty sequence
numbers = range(-4)
print(list(numbers))
  # []
```

Example 2: range() with Start and Stop Arguments

- If we pass two arguments to range(), it means we are passing start and stop arguments.
- In this case, range() returns a sequence of numbers starting from start (inclusive) up to stop (exclusive).

```
# numbers from 2 to 4 (5 is not included)
numbers = range(2, 5)
print(list( numbers))
# [2, 3, 4]
# numbers from -2 to 3 (4 is not included)
numbers = range(-2, 4)
print(list(numbers))
 # [-2, -1, 0, 1, 2, 3]
# returns an empty sequence of numbers
numbers = range(4, 2)
print(list(numbers))
# []
```

Operators in and not in

- The **in** operator tests if a given value is contained in a sequence or not and returns True or False.
- The **not in** operator tests if a given value is contained in a sequence or not and returns True or False.
- Example

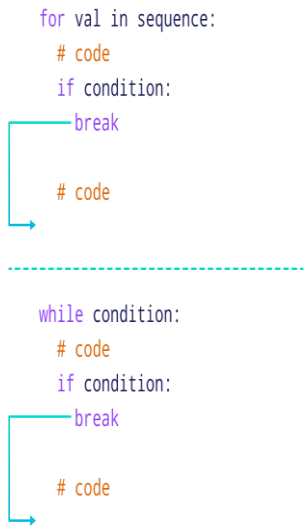
1. 3 in [1,2,3,4] it will return True.
2. 5 in [1,2,3,4,] it will return False
3. 3 not in [1,2,3,4] it will return False.
4. 5 not in [1,2,3,4,] it will return True

Python break Statement

The break statement is used to terminate the loop immediately when it is encountered.

The syntax of the break statement is:

Working of Python break Statement



Python break Statement with for Loop

We can use the break statement with the for loop to terminate the loop when a certain condition is met. For example,

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output

```
0
1
2
```

In the above example, we have used the for loop to print the value of i. Notice the use of the break statement,

```
if i == 3:
break
```

Here, when i is equal to 3, the break statement terminates the loop. Hence, the output doesn't include values after 2.

Note: The break statement is almost always used with decision-making statements.

Python break Statement with while Loop

We can also terminate the while loop using the break statement. For example,

```
# program to find first 5 multiples of 6
i = 1
while i <= 10:
    print('6 * ',(i), '=',6 * i)
    if i >= 5:
        break
        i = i + 1
```

Output

```
6 *  1 = 6
6 *  2 = 12
6 *  3 = 18
6 *  4 = 24
6 *  5 = 30
```

In the above example, we have used the while loop to find the first 5 multiples of 6. Here notice the line,

```
if i >= 5:
    break
```

This means when i is greater than or equal to 5, the while loop is terminated.

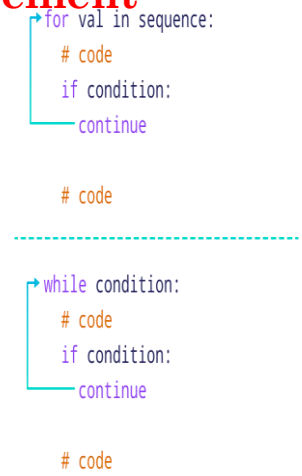
Python continue Statement

The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

The syntax of the continue statement is:

```
continue
```

Working of Python continue Statement



Python continue Statement with for Loop

We can use the continue statement with the for loop to skip the current iteration of the loop. Then the control of the program jumps to the next iteration. For example,

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

Output

```
0
1
2
4
```

In the above example, we have used the for loop to print the value of i. Notice the use of the continue statement,

```
if i == 3:
    continue
```

Here, when i is equal to 3, the continue statement is executed. Hence, the value 3 is not printed to the output.

Python continue Statement with while Loop

In Python, we can also skip the current iteration of the while loop using the continue statement. For example,

```
# program to print odd numbers from 1 to 10
num = 0
while num < 10:
    num += 1
        if (num % 2) == 0:
        continue
    print(num)
```

Output

1
3
5
7
9

In the above example, we have used the while loop to print the odd numbers between 1 to 10. Notice the line,

```
if (num % 2) == 0:
    continue
```

Here, when the number is even, the continue statement skips the current iteration and starts the next iteration.

Python Exit()

- The built-in Python procedures exit(), quit(), sys.exit(), and os. exit() are most frequently used to end a program.
- **Syntax of exit() in Python**

```
exit() Function
```

- We can use the in-built exit() function to quit and come out of the execution loop of the program in Python.

sys.exit() Function

- sys.exit() is a built-in function in the Python sys module that allows us to end the execution of the program.
- **Syntax:**

```
sys.exit(argument)
```

Example of exit() in Python

```
number = 7
    if number < 8:
    # exits the program
    print(exit)
    exit()
else:
    print("number is not less than 8")
```

output

```
Use exit() or Ctrl-D (end-of-file) to exit
```

After writing the above code (python exit() function), once you run the code it will just give us the exit message. Here, if the value of the "number" is less than 8 then the program is forced to exit, and it will print the exit message.

THANK YOU

Prepared By Pruthvi Raj

Unit 2

Python

Prepared by Pruthvi Raj A Dept. of CS GFGC MALUR

Python Exception Handling

ERROR

Errors are problems that occur in the program due to an illegal operation performed by the user or by the fault of a programmer, which halts the normal flow of the program. Errors are also termed bugs or faults. There are mainly two types of errors in python programming. Let us learn about both types of python errors:

1. **Syntax errors**
2. **Logical Errors or Exceptions**

1. Syntax Errors

A syntax error is one of the most basic types of error in programming. Whenever we do not write the proper syntax of the python programming language (or any other language) then the python interpreter throws an error known as a syntax error.

Example:

```
number = 100
if number > 50
print("Number is greater than 50!")
```

Out put:

```
File "test.py", line 3
if number > 50
                ^
SyntaxError: invalid syntax
```

2. Logical Errors

- Logical Errors are those errors that cannot be caught during compilation time. As we cannot check these errors during compile time, we name them Exceptions. Since we cannot check the logical errors during compilation time, it is difficult to find them.
- There is one more name of logical error. Run time errors are those errors that cannot be caught during compilation time. So, run-time can cause some serious issues so we should handle them effectively.

- some of the most common logical types of errors in python programming.

➢ ZeroDivisionError Exception
➢ Indentation is not Correct
➢ Built-in Exceptions

ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.

Exception in Python

An exception is an unexpected event that occurs during program execution. For example,

divide_by_zero = 7 / 0

The above code causes an exception as it is not possible to divide a number by 0.

Definition: An Exception is an error that happens during the execution of a program. Whenever there is an error, Python generates an exception that could be handled. It basically prevents the program from getting crashed.

Common inbuilt Exceptions in Python

• ImportError	Raised when an import statement fails
• NameError	Raised when an identifier is not found in the local or non-local or global scope.
• SyntaxError	Raised when there is an error in python syntax.
• TypeError	Raised when a specific operation of a function is triggered for an invalid data type
• RuntimeError	Raised when an error does not fall into any other category
• ArithmeticError	Errors that occur during numeric calculation are inherited by it.
• OverflowError	When a calculation exceeds the max limit for a specific numeric data type
• ZeroDivisionError	Raised when division or modulo by zero takes place.
• EOFError	Raised when the end of file is reached, usually occurs when there is no input from input() function

Exception Handling

- Exception handling is the process of responding to the occurrence of exceptions-anomalous or exceptional conditions requiring special processing during the execution of a program.
- In Python, we catch exceptions and handle them using try, except, else and finally code blocks.

Try and Except Statement

Syntax:

```
try :
    #statements in try block
except :
    #executed when error in try block
```

- **The try:** block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.
- If the exception does occur, the program flow is transferred to the **except:** block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message.
- example

```
try:
        a=5
        b='0’
        print(a/b)
except:
        print('Some error occurred.’)
        print("Out of try except
blocks.")
```

Python try with else clause

- In some situations, we might want to run a certain block of code if the code block inside try runs without any errors.
- For these cases, you can use the optional else keyword with the try statement.
- Let's look at an example: program to print the reciprocal of even numbers

```
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Out put

1. Enter a number: 1
   Not an even number!
2. Enter a number: 4
   0.25
3. Enter a number: 0 Traceback (most recent call last): File "<string>", line 7, in <module> reciprocal = 1/num ZeroDivisionError: division by zero

Python try...finally

- In Python, the finally block is always executed no matter whether there is an exception or not.
- The finally block is optional. And, for each try block, there can be only one finally block.
- Let's see an example,

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

Out put

```
Error: Denominator cannot be 0.
This is finally block.
```

Functions in Python

A function in Python is a collection of connected statements that performs a single task. Functions help in the division of our program into smaller, **modular** portions. Functions help our program become more ordered and controllable as it grows in size. It also eliminates **repetition** and makes the code reusable.

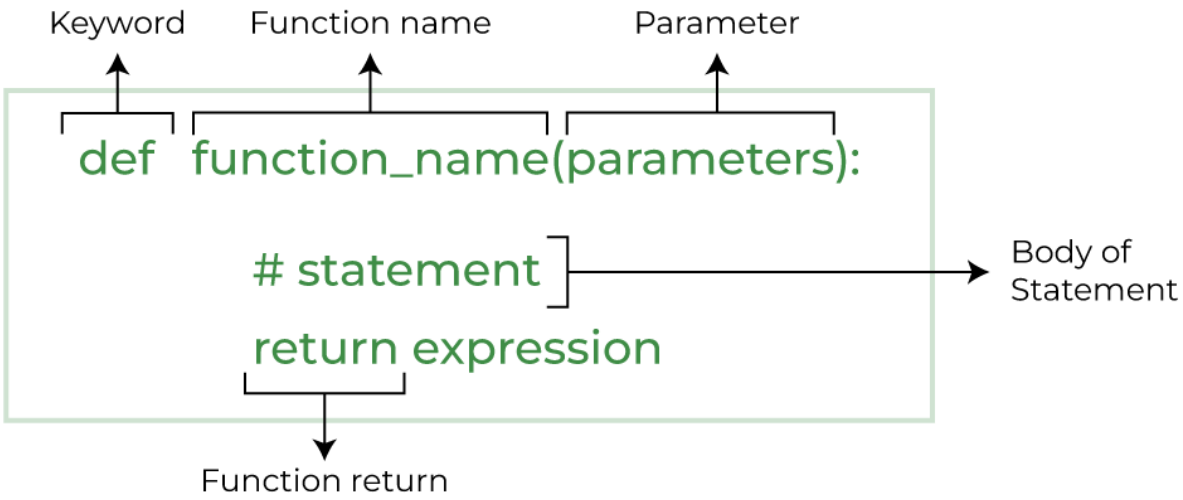
What are Functions in Python?

Functions are modular blocks of code designed to perform specific tasks. They enhance code efficiency and clarity by reducing code repetition and enabling code reuse.

For example of functions

- A function can be related to the process of making tea. If you were to make tea for the first time, your parents would have defined the steps to make it. The next time they want to drink tea, they'll call out to you directly by saying 'Make Tea'.
- Similarly, in the case of functions, you first define the sequence of steps that you would like to carry out to achieve a certain task. Later, you can call out the Function by its name, and these steps will be performed.

syntax to declare a function is:



Example :

```
def square( num ):
    """
    This function computes the square of the number.
    """
    return num**2
object_ = square(6)
print( "The square of the given number is: ", object_ )
```

The components of a python function definition are:

1. Keyword def:

In python, the keyword def is used to define a function. It marks the beginning of the function definition.

2. Name of the Function:

function_name is a unique identifier that is considered the name of the Function.

3. Parameters:

These are the values passed to the Function. 0 or more parameters can be included in the parentheses.

4. **Colon(:):** The:

represent the start of the indented function block.

5. Statements:

These include the sequence of tasks that the Function is to perform. It can also use the pass keyword in case of no statements. These statements are usually indented at the same level (usually 2 or 4 spaces).

6. Return Statement:

The optional return statement symbolizes the return of values from the Function to the calling code.

Rules for Naming Functions in Python

- They must begin with a letter or an underscore.
- They should be lowercase.
- They can consist of numbers but shouldn't start with one.
- They can be of any length.
- They shouldn't be a keyword in python, i.e. be reserved for any purpose. To improve readability and to maintain conventions, separate words by underscores. Ensure that you give function names that define the purpose of the Function.

Types of Functions in Python

1. User-defined:

User-defined functions - We can create our own functions based on our requirements.

Example:

```
def square( num ):
    """
    This function computes the square of the number.
    """
    return num**2
object_ = square(6)
print( "The square of the given number is: ", object_ )
```

Output:

The square of the given number is: 36

2. **Built-in:**

These are already defined within Python, and thus a user need not define them again. They can be directly used in a program or application.

Example:

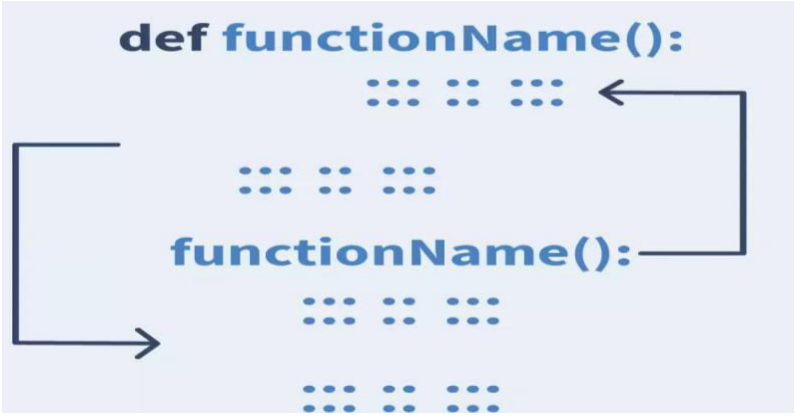
abs() is used to find the absolute value of a number.

```
x = abs(-7.25)
print(x)
```

Output:

7.25

How Does a Python Function Work?



- A python function consists of 2 parts:
- Function definition and function call.
- To create a function in Python, you have to first define the Function, giving it a name, parameters, statements, etc., and then call it separately in the program using its names and parameters, if any.

How to Call a Function in Python?

A function in Python is called by using its name followed by parentheses. In case parameters are present, these are included in the parentheses.

Example: To call the Function named hello, simply type the following:

hello()

```
1. # Example Python Code for calling a function
2. # Defining a function
3. def a_function( string ):
4.     "This prints the value of length of string"
5.     return len(string)
6.  # Calling the function we defined

7. print( "Length of the string Functions is: ", a_function( "Functions" ) )

8. print( "Length of the string Python is: ", a_function( "Python" ) )
```

Output:

Length of the string Functions is: 9

Length of the string Python is: 6

Passed by Reference or Passed by Value

Call by Value in Python

- When **Immutable objects** such as whole numbers, strings, etc are passed as arguments to the function call, the it can be considered as Call by Value.
- This is because when the values are modified within the function, then the **changes do not get reflected** outside the function.

```
def myFunc(a):
print("Value received in 'a'
=", a)
a+=2
print("Value of 'a' changes
to :",a)

num=13
print("Initial number: ",
num)
myFunc(num)
print("Value of number= ",
num)
```

Output:

Initial number: 13

Value received in 'a' = 13

Value of 'a' changes to : 15

Value of number= 13

Explanation: As we can see here, the value of the number changed from 13 to 15, inside the function, but the value remained 13 itself outside the function. Hence, the changes did not reflect in the value assigned outside the function.

➢ Pass by reference

- When **Mutable objects** such as list, dict, set, etc., are **passed as arguments** to the function call, it can be called Call by reference in Python. When the values are modified within the function, the **change also gets reflected** outside the function.
- **Example 1 showing Call by reference in Python**

```
def myFunc(myList):
print("List received: ",myList)
myList.append(3)
myList.extend([7,1])
print("List after adding some elements:", myList)
myList.remove(7)
print("List within called function:", myList)
return List1=[1]
print("List before function call :",List1)
myFunc(List1)
print("List after function call: ",List1)
```

Output:

List before function call : [1]

List received: [1]

List after adding some elements: [1, 3, 7, 1]

List within called function: [1, 3, 1]

List after function call: [1, 3, 1]

Explanation: Here we can see that the list value within the function changed from [1] to [1, 3, 1] and the changes got reflected outside the function.

Function Arguments

- Functions with arguments/parameters are called parameterized functions. To call these functions, the number and order of arguments should be as given in the function definition.

- A function can have 1 or more parameters. A function in python can accept up to 255 parameters.

- The arguments/parameters used in the python function definition are called formal arguments or formal parameters while the ones in the function call are called local arguments or local parameters. The name is based on their location in the program.

- Function parameters can have their type defined along with them, as given in the example:

Example: Function with Single Parameter

```
def hello(name):
    print ("Hello", name)
hello(“madam")
```

Output

Hello madam

Example: Function with Multiple Parameters

```
def hello(name1, name2, name3):
    print("Hello", name1, "\nHi", name2, "\nHola", name3)
hello("Siri", "Kanchana", "Madhuri")
```

Output:

Hello Siri
Hi Kanchana
Hola Madhuri

There are four major types of arguments:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

1. Required Arguments

In such functions, the exact number of arguments are required to be passed in the order in which it is defined in the Function.

Example:

```
def hello(name):
    print ("Hello", name)
hello(“BCA")
```

Output

Hello BCA

2. Keyword Arguments

The term “keyword” is pretty self-explanatory. It can be broken down into two parts— a key and a word (i.e., a value) associated with that key.

To understand keyword arguments in Python, let us first look at the example given below.

For example:

```
def divide_two(a, b):
    res = a / b
    return res
 res = divide_two(12, 3)
print(res)
```

Output: 4.0

3. Default Arguments

- Default function arguments in python, as the name suggests, are some default argument values that we provide to the function at the time of function declaration.
- Thus, when calling the function, if the programmer doesn't provide a value for the parameter for which we specified the default argument, the function assumes the default value as the argument in that case.
- Thus, calling the function will not result in any error even if we don't provide any value as an argument for the parameter with a default argument.
- We declare a default argument by using the equal-to (=) operator at the time of function declaration.

- **Example:**

```
def add_numbers( a = 7,  b = 8):
    sum = a + b
    print('Sum:', sum)
# function call with two arguments
add_numbers(2, 3)
#  function call with one argument
add_numbers(a = 2)
# function call with no arguments
add_numbers()
```

Output

Sum: 5

Sum: 10

Sum: 15

4. Variable-length Arguments or Arbitrary arguments

In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

1. *args in Python (Non-Keyword Arguments)
2. **kwargs in Python (Keyword Arguments)

Exmple:

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print(arg)
myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
Hello
Welcome
to
GeeksforGeeks
```

- **2nd example :**

```
# Python program to illustrate
# *kwargs for variable number of keyword arguments

def myFun(**kwargs):
    for key, value in kwargs.items():
            print("%s == %s" % (key, value))

# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

Output:

```
first == Geeks
mid == for
last == Geeks
```

Recursion in Python

What is Recursion in Python?

- What would happen if we placed two mirrors parallel to each other? You will see multiple images stretched to infinity caused by the reflections between the surfaces.
- Imagine you want to know the name of a person in the queue that you are standing in. You ask people to find out about it.
- They keep asking the next person until they find an answer. Once an answer is found, they send it back until it reaches you.
- The above two examples depict a process in which a particular action is repeated until a condition is met. These processes have a strong resemblance to what we call recursion.
- A process in which a function calls itself is called recursion. This process helps ease the method of solving problems by replacing iterative code with recursive statements.

Use of Recursion in Python

- Quite often, people wonder why we should use recursion when loops exist. Everything can be coded without recursion.
- The difference between iteration and recursion is there is no sequential end to recursive code. It tests on a base condition and can go on as long as that is satisfied.
- **Recursion is used in Python when problems can be broken into simpler parts for easier computation and more readable code.**

Recursive function in Python has two parts:

(a) Base Case -This helps us to terminate the recursive function. It is a simple case that can be answered directly and doesn't use recursion. If satisfied, it returns the final computable answer. If this is omitted, the function will run till infinity.

Python interpreter limits the number of recursive calls for a function to 1000 by giving a recursion error.

b) **General (Recursive) Case** - This case uses recursion and is called unless the base condition is satisfied.

Syntax:

```
def rec_func_name():
  if(condition)                    # base case
     simple statement without recursion
  else                             # general case
     statement calling rec_func_name()
```

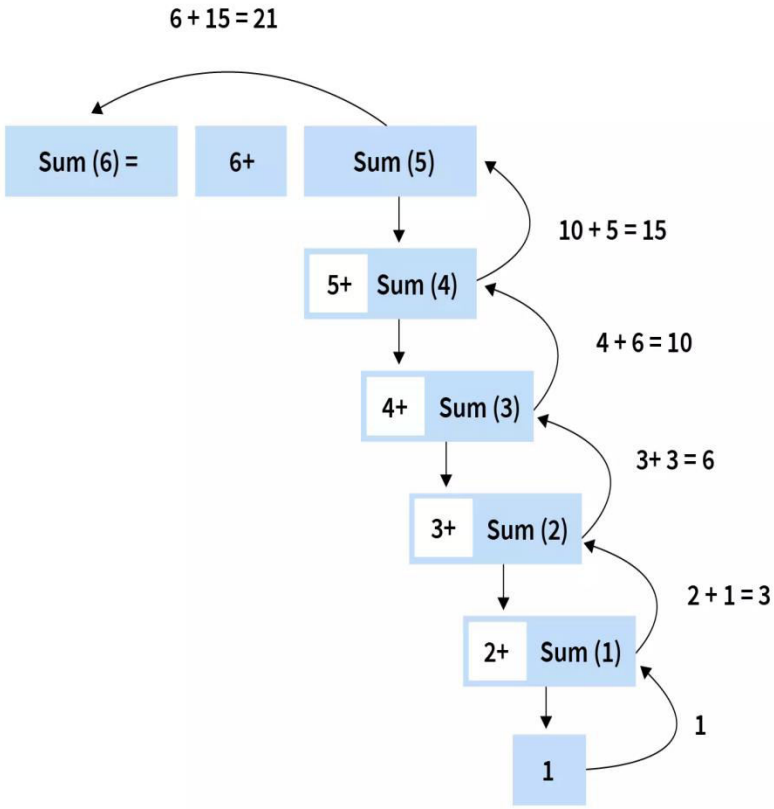
Example of Recursive Function

Suppose you want to find the sum of n natur
numbers in python. We would write the foll
code to get the answer.

```
def sum(n):
   if n <= 1:   # base case
      return n
   else:       # general or recursive case
      ans = sum(n - 1)
   return n + ans
print(sum(6))
```

Output:

21



Types of Recursion in Python

There are mainly 2 types of recursive functions:

1. Direct Recursion
2. Indirect Recursion

1) Direct Recursion

- When a function calls itself within the same function repeatedly, it is called the direct recursion.

Structure of the direct recursion

```
fun()
{
// write some code
fun();
// some code
}
```

- In the above structure of the direct recursion, the outer fun() function recursively calls the inner fun() function, and this type of recursion is called the direct recursion.

- **Factorial is an example of direct recursive as shown below:**

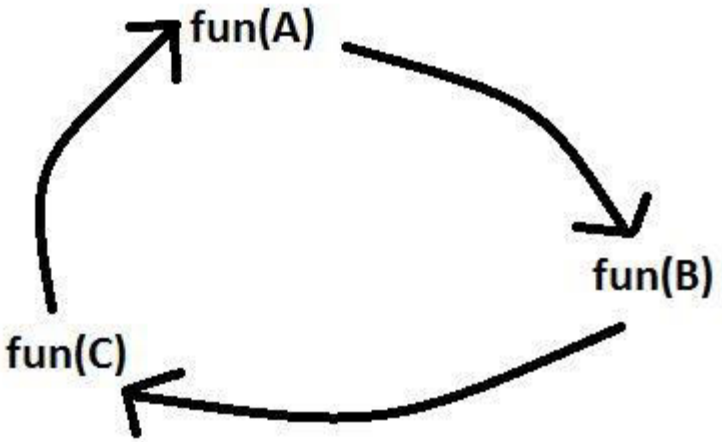
```
def factorial(n):
 if n < 0 or n == 1:
 # The function terminates here
 return 1
 else:
 value = n*factorial(n-1)
 return value

# Returns the factorial of 5
factorial(5)
```

Output:

120

2. Indirect Recursion

- When a function is mutually called by another function in a circular manner, the function is called an indirect recursion function.



- From the above diagram fun(A) is calling for fun(B), fun(B) is calling for fun(C) and fun(C) is calling for fun(A) and thus it makes a cycle.

 # Python program to show Indirect Recursion

```
def funA(n):
    if (n > 0):
        print("", n, end='')

        # Fun(A) is calling fun(B)
        funB(n - 1)

def funB( n):
    if (n > 1):
        print("", n, end='')

        # Fun(B) is calling fun(A)
        funA(n // 2)

# Driver code
funA(20)
```

Scope and Lifetime of Variables in Functions

- The scope defines the accessibility of the python object. To access the particular variable in the code, the scope must be defined as it cannot be accessed from anywhere in the program.
- A variable is only available from inside the region it is created. This is called **scope**.
- Based on the scope, we can classify Python variables into three types:

1. **Local Variables**
2. **Global Variables**
3. **Nonlocal Variables**

1. Local Variables

•When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function.

•These types of variables are called local variables. For example,

```
def greet():
    # local variable
    message = 'Hello'
    print('Local', message)
greet()
# try to access message variable
# outside greet() function
print(message)
```

Output

Local Hello

NameError: name 'message' is not defined

2. **Python Global Variables**

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created in Python.

```
# declare global variable
message = 'Hello'
def greet():
    # declare local variable
    print('Local', message)
greet()
print('Global', message)
```

Output

Local Hello

Global Hello

3. **Python Nonlocal Variables**

The ‘nonlocal’ keyword is used to work in scenarios where we have nested functions. Using the nonlocal keyword, we instruct the variable that it should not belong to the inner function. Let’s understand this with an example:

```
def func1():
  a = 20
  def func2():
    nonlocal a
    print('func2 ', a)
  func2()
  print('func1 ', a)
func1()
```

Output:

func2 20

func1 20

Strings

What is a string in python?

•In computer programming, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

•We use single quotes or double quotes to represent a string in Python.

Creating and storing String in Python

A string in Python can be easily created using single, double, or even triple quotes. The characters that form the string are enclosed within any of these quotes.

Note: Triple quotes are generally used when we are working with multiline strings in Python and docstring in Python.

Let’s take a look at an example where we will create strings using the three types of quotes –

Code:

```
single_string='Hello'
double_string="Hello World"
triple_string="""Welcome to Scaler"""
print(single_string)
print(double_string)
print(triple_string)
```

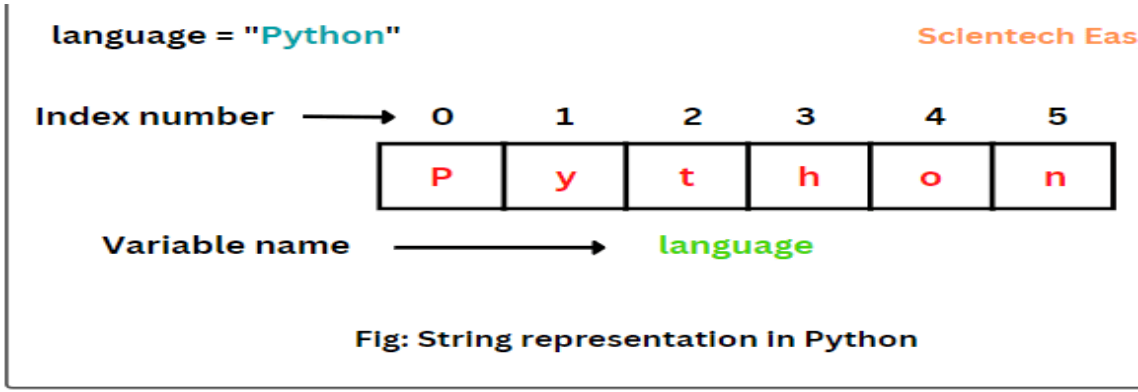


Fig: String representation in Python

Output:

Hello

Hello World

Welcome to Scaler

Access String Characters in Python

We can access the characters in a string in three ways.

1. Indexing: One way is to treat strings as a list and use index values. For example,

```
greet = 'hello'
# access1st index element
print(greet[1]) # "e“
```

2. Negative Indexing: Similar to a list, Python allows negative indexing for its strings. For example,

```
greet = 'hello'
# access 4th last element
print(greet[-4]) # "e"
```

3. Slicing: Access a range of characters in a string by using the slicing operator colon **:**. For example,

Note: If we try to access an index out of the range or use numbers other than an integer, we will get errors.

The syntax looks like this –

```
'str[start_index: end_index]'
```

While performing slicing operations, one thing to remember is that the start index value is always included while that of the last index is always excluded.

Slicing can also be negative or positive, just like indexing in Python!

Let's consider the following examples to understand this better –

Code:

```
message="Welcome to Scaler"
print(message[2:5]) # positive slicing

print(message[-10:-2]) #negative slicing
print(message[:5]) #slicing from the start
print(message[2:]) #slicing to the end
```

Output:

```
lco
 to scal
welco
lcome to Scaler
```

How to delete a string?

We have established that strings are immutable. In simple words, it means that once a string is assigned, we cannot make any changes to it. We cannot remove or delete any character of the string.

What we can do is delete the Python string entirely. We can do so by using the del keyword.

The following code snippet shows how we can do that –

```
message="Welcome to Scaler"
print(message)
del message
print(message)
```

```
Welcome to Scaler Traceback
(most recent call last) :
File "main.py", line 4, in <module>
print (message)
NameError: name 'message' is not defined
```

Python str()

The str() method returns the string representation of a given object.

Example

```
# string representation of Adam
print(str('Adam'))
# Output:  Adam
```

str() Syntax

The syntax of str() is:

str(object, encoding='utf-8', errors='strict')

str() Parameters

The str() method takes three parameters:

object - whose string representation is to be returned

encoding - that the given byte object needs to be decoded to (can be UTF-8, ASCII, etc)

errors - a response when decoding fails (can be strict, ignore, replace, etc)

Note: There are six types of errors: strict, ignore, replace, xmlcharrefreplace, namereplace, backslashreplace. The default error is strict.

The str() method returns:

A printable string representation of a given object

string representation of a given byte object in the provided encoding

Example 1: Python() String

```
# string representation of Luke
name = str('Luke')
print(name)
# string representation of an integer 40
age = str(40)
print(age)
# string representation of a numeric string 7ft
height = str('7ft')
print(height)
```

Output

```
Luke
40
7ft
```

In the above example, we have used the str() method with different types of arguments like string, integer, and numeric string.

Example 2: str() with Byte Objects

We can use the str() method with byte objects which are defined by the bytes() method.

In this case, we need to specify the encoding that we want to convert the byte objects to and the type of error checking that the str() method can perform.

```
# declare a byte object
b = bytes('pythön', encoding='utf-8')
# convert a utf-8 byte object to ascii with errors ignored
print(str(b, encoding='ascii', errors='ignore'))
# convert a utf-8 byte object to ascii with strict error
print(str(b, encoding='ascii', errors='strict'))
```

Output

pythn

UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 4: ordinal not in range(128)

In the first example, we have created a byte object b with the string 'pythön' and the encoding utf-8.

We have passed the b object to the str() method and specified the encoding to ascii.

Here, we have set the errors parameter to ignore so, the str() method ignores the character 'ö'. Since the method can't decode this character to ascii, we get the output pythn.

Similarly, in the second example, we've set the error to strict. In this case, the str() method will take the character 'ö' into account and produce UnicodeDecodeError as the output.

String Operators in Python

Arithmetic operators do not work on strings. But, we do have certain operators for performing special operations on strings in Python.

Let's check out some of them –

Operator	Description
+	It is used to concatenate two strings. It appends the second string to the first string.
*	It concatenates multiple copies of the same string. It is basically a repetition operator.
[]	It is used for indexing. The value put between this operator returns the character at the given index.
[:]	It is used for slicing. It returns the sub-strings after slicing as per the given range.
in	It returns true if the character is present in the given string.
not in	It returns true if the character is not present in the given string.

Example Program of String Operators in Python

Code:

```
message1="Hello World!"
message2="Welcome to Scaler"
print(message1+message2) # + operator
print(message1*3) # * operator
print(message1[6]) # [] operator
print(message2[0:7]) # [:] operator
str1="Hello"
if str1 in message1:# in operator
    print("It is a present!")
if str1 not in message2:# not in operator
    print("It is not present!")
```

Output

```
Hello World!Welcome to Scaler
Hello World!Hello World!Hello
World!
w
Welcome
It is a present!
It is not present!
```

Python String Operations

1. Compare Two Strings

We use the == operator to compare two strings. If two strings are equal, the operator returns True. Otherwise, it returns False. For example,

```
str1 = "Hello, world!"
str2 = "I love Python."
str3 = "Hello, world!"
# compare str1 and str2
print(str1 == str2)
# compare str1 and str3
print(str1 == str3)
```

Output

False

True

In the above example,

str1 and str2 are not equal. Hence, the result is False.

str1 and str3 are equal. Hence, the result is True.

2. Join Two or More Strings(Concatenation)

In Python, we can join (concatenate) two or more strings using the + operator.

```
greet = "Hello, "
name = "Jack"
# using + operator
result = greet + name
print(result)
```

Output: Hello, Jack

In the above example, we have used the + operator to join two strings: greet and name.

3. Python String Length

In Python, we use the len() method to find the length of a string.

Syntax:

Len(<Sting>)

For example,

```
wishes = ‘Morning'
# count length of wishes string
print(len(wishes))
```

Output: 7

4.String Membership Test

We can test if a substring exists within a string or not, using the keyword in.

```
print('a' in 'program')
```

output

True

```
print('at' not in 'battle')
```

output

False

5.Python Reverse String

There isn't any built-in function to reverse a given String in Python but the easiest way is to do that is to use a slice that starts at the end of the string, and goes backward.

```
x = "good" [::-1]
print(x)
```

The output will be: doog

Methods of Python String

- Besides those mentioned above, there are various string methods present in Python. Here are some of those methods:

Methods	Description
upper()	converts the string to uppercase
lower()	converts the string to lowercase
partition()	returns a tuple
replace()	replaces substring inside
find()	returns the index of first occurrence of substring
rstrip()	removes trailing characters
split()	splits string from left
startswith()	checks if string starts with the specified string
isnumeric()	checks numeric characters
index()	returns index of substring

String Slicing in Python

- Python slicing is about obtaining a sub-string from the given string by slicing it respectively from start to end.

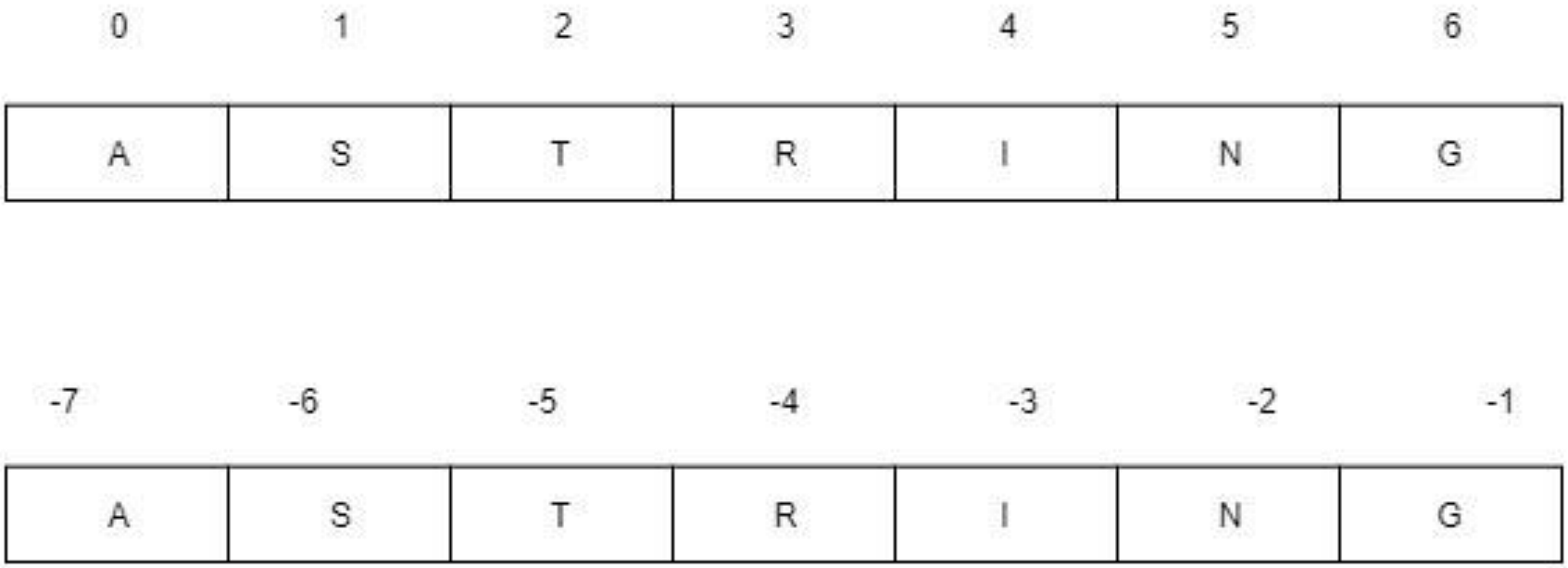
How String slicing in Python works

- For understanding slicing we will use different methods, here we will cover 2 methods of string slicing, one using the in-build slice() method and another using the [:] array slice. String slicing in Python is about obtaining a sub-string from the given string by slicing it respectively from start to end.

- Python slicing can be done in two ways:

1. Using a slice() method
2. Using the array slicing [::] method

Index tracker for positive and negative index: String indexing and slicing in python. Here, the Negative comes into consideration when tracking the string in reverse.



Method 1: Using the slice() method

The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step).

Syntax:

slice(stop)

slice(start, stop, step)

Parameters:

start: Starting index where the slicing of object starts.

stop: Ending index where the slicing of object stops.

step: It is an optional argument that determines the increment between each index for slicing.

Return Type: Returns a sliced object containing elements in the given range only.

example

```
# Python program to demonstrate
# string slicing

# String slicing
String = 'ASTRING'

# Using slice constructor
s1 = slice(3)
s2 = slice(1, 5, 2)
s3 = slice(-1, -12, -2)

print("String slicing")
print(String[s1])
print(String[s2])
print(String[s3])
```

Output

```
String slicing
AST
SR
GITA
```

Method 2: Using the List/array slicing [::] method

In Python, indexing syntax can be used as a substitute for the slice object. This is an easy and convenient way to slice a string using list slicing and Array slicing both syntax-wise and execution-wise. A start, end, and step have the same mechanism as the slice() constructor.

Below we will see string slicing in Python with examples.

Syntax

```
arr[start:stop]        # items start through stop-1
arr[start:]          # items start through the rest of the array
arr[:stop]          # items from the beginning through stop-1
arr[:]              # a copy of the whole array
arr[start:stop:step]    # start through not past stop, by step
```

Example :

In this example, we will see slicing in python list the index start from 0 indexes and ending with a 2 index(stops at 3-1=2).

```
# Python program to demonstrate
# string slicing

# String slicing
String = 'GEEKSFORGEEKS'

# Using indexing sequence
print(String[:3])
```

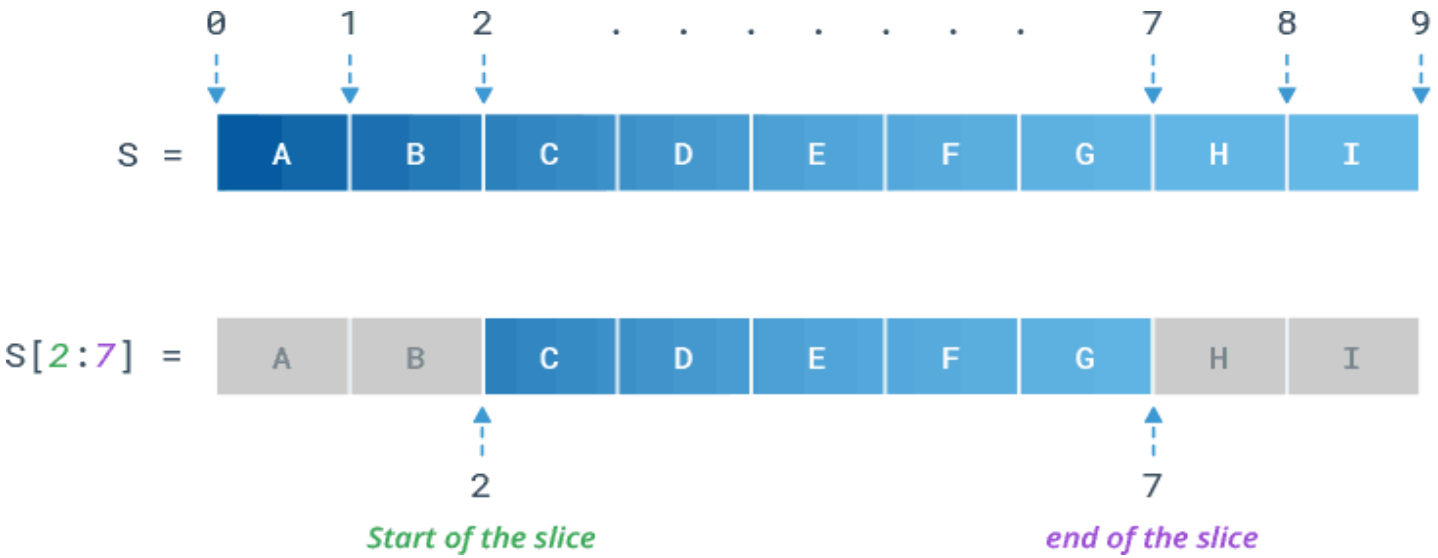
Output:

GEE

Basic Example

Here is a basic example of string slicing.

```
S = 'ABCDEFGHI'
print(S[2:7])
Output # CDEFG
```



Note that the item at index 7 'H' is not included.

Slice with Negative Indices

You can also specify negative indices while slicing a string.

```
S = 'ABCDEFGHI'
print(S[-7:-2])
```

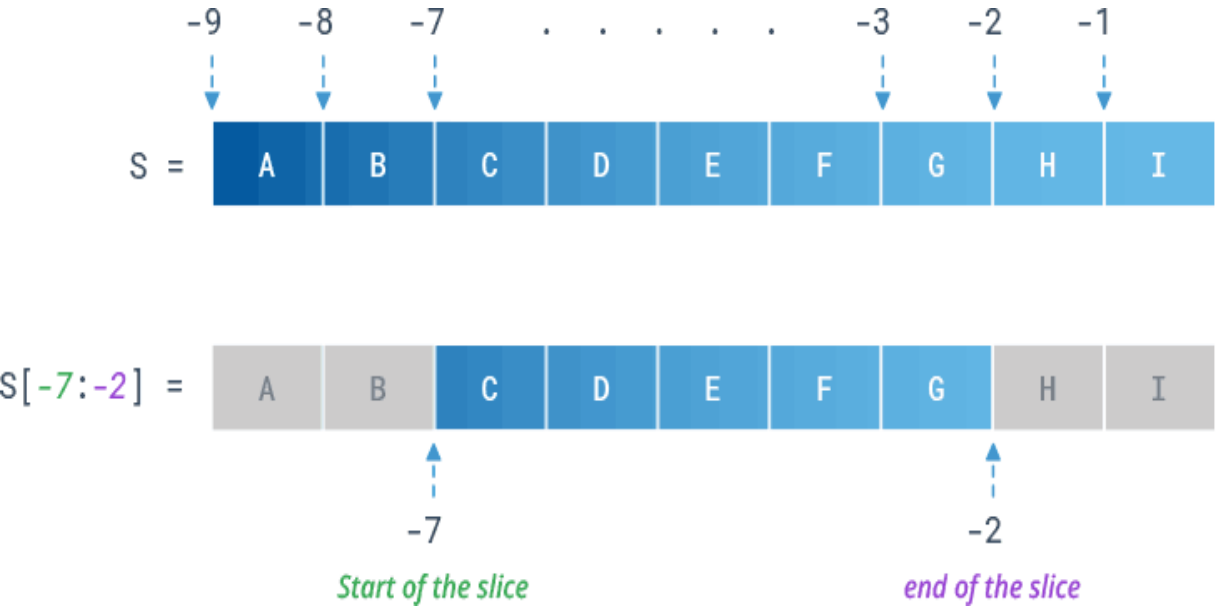
Output # CDEFG

Slice with Positive & Negative Indices

You can specify both positive and negative indices at the same time.

```
S = 'ABCDEFGHI'
print(S[2:-5])
```

output# CD

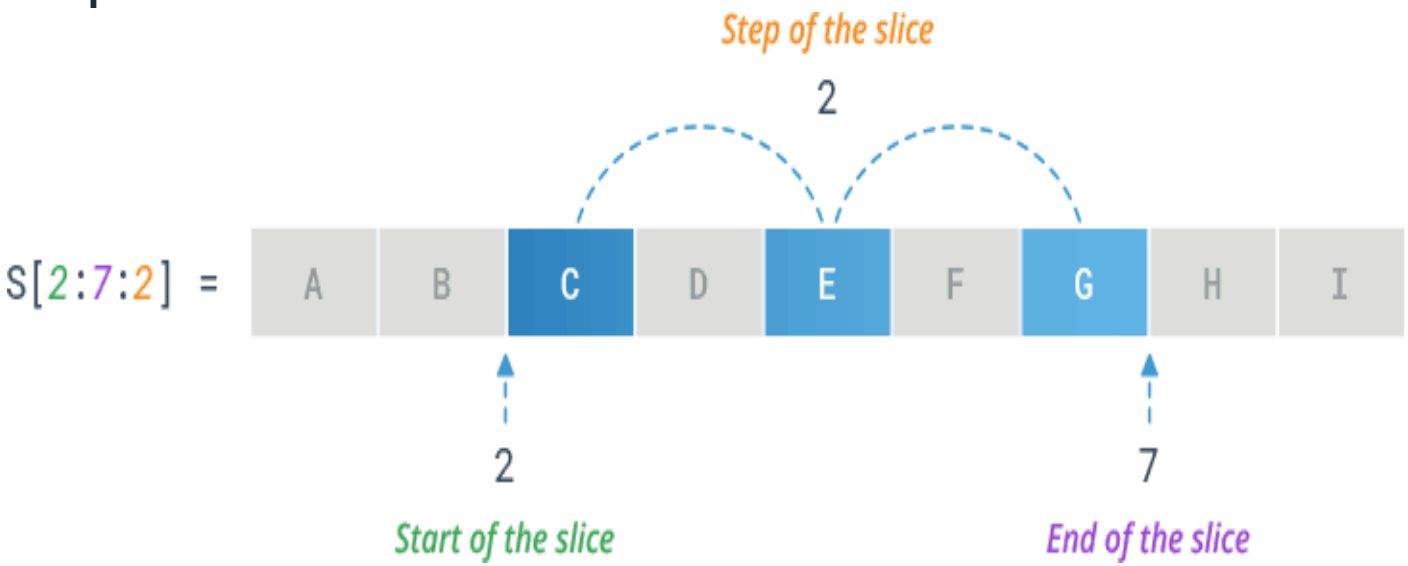


Specify Step of the Slicing

You can specify the step of the slicing using step parameter. The step parameter is optional and by default 1.

```
# Return every 2nd item between
position 2 to 7
S = 'ABCDEFGHI'
print(S[2:7:2])
```

Output # CEG



Negative Step Size

You can even specify a negative step size.

```
# Returns every 2nd item
between position 6 to 1 in
reverse order
S = 'ABCDEFGHI'
print(S[6:1:-2])
```

Output# GEC

Slice at Beginning & End

Omitting the start index starts the slice from the index 0. Meaning, S[:stop] is equivalent to S[0:stop]

```
# Slice first three characters from the string
S = 'ABCDEFGHI’
print(S[:3])

Output# ABC
```

Whereas, omitting the stop index extends the slice to the end of the string. Meaning, S[start:] is equivalent to S[start:len(S)]

```
# Slice last three characters from the string
S = 'ABCDEFGHI’
print(S[6:])

Output# GHI
```

Python String join() Method

- Python join() is an inbuilt string function in Python used to join elements of the sequence separated by a string separator. This function joins elements of a sequence and makes it a string. (Or)
- The join in Python takes all the elements of an iterable and joins them into a single string. It will return the joined string. You have to specify a string separator that will be used to separate the concatenated string.
- **Syntax**

```
string.join(iterable)
```

Parameter	Condition	Description
`iterable`	Required	Any iterable (like list, tuple, dictionary etc.) whose items are strings

Return Value

The method returns the string obtained by concatenating the items of an iterable.

Basic Examples

```
# Join all items in a list with
comma
L = ['red', 'green', 'blue’]
x = ','.join(L)
print(x)
```

Output
red,green,blue

```
# Join list items with newline
 L = ['First Line', 'Second Line’]
x = '\n'.join(L)
print(x)
```

output
First Line
Second Line

Join a List of Integers

If there are any non-string values in iterable, a TypeError will be raised.

```
L = [1, 2, 3, 4, 5, 6]
x = ','.join(L)
print(x)
```

Output

```
# Triggers TypeError: sequence
item 0:
 expected string, int found
```

To avoid such exception, you need to convert each item in a list to string. The list comprehension makes this especially convenient.

```
L = [1, 2, 3, 4, 5, 6]
x = ','.join(str(val) for val in L)
print(x)
# Prints 1,2,3,4,5,6
```

join() on Dictionary

When you use a dictionary as an iterable, all dictionary keys are joined by default.

```
L = {'name':'Bob', 'city':'seattle’}
x = ','.join(L) print(x)
# Prints city,name
```

To join all values, call values() method on dictionary and pass it as an iterable.

```
L = {'name': 'Bob', 'city':'seattl
x = ','.join(L.values())
print(x)
# Prints Seattle, Bob
```

To join all keys and values, use join() method with list comprehension.

```
L = {'name':'Bob',
'city':'seattle’}
x = ','.join('='.join((key,val))
 for (key,val) in L.items())
 print(x)
 # Prints city=seattle,name=Bob
```

Traversing

Traversing a string means accessing all the elements of the string one after the other by using the subscript. A string can be traversed using for loop or while loop.

Example

Using while loop

In Python, a while loop may have an optional else block.

Here, the else part is executed after the condition of the loop evaluates to False.

Example

```
counter = 0
while counter < 3:
  print('Inside loop’)
  counter = counter + 1
else:
  print('Inside else')
```

Output

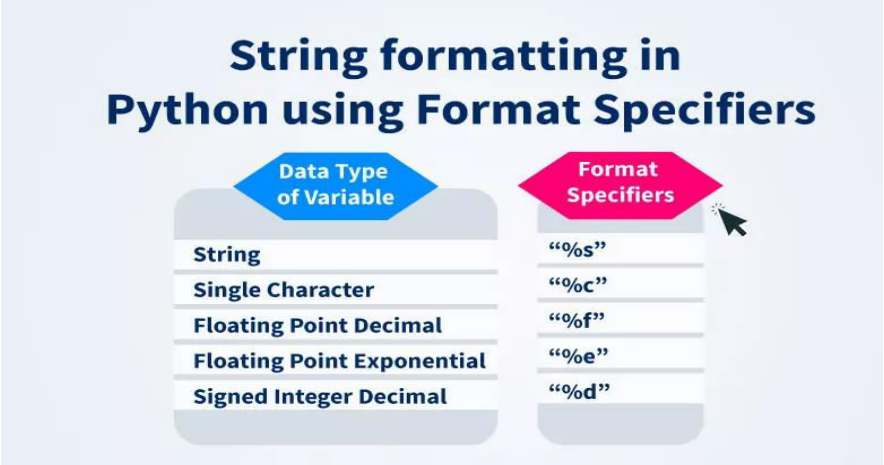
Inside loop
Inside loop
Inside loop
Inside else

Format Specifiers

Ghhh

Introduction

String formatting is a way to insert a variable or another string in a predefined string. We can use various methods of string formatting in Python to manipulate strings. We can use format specifiers like other programming languages to perform string formatting. We can also use the format() method and f-strings to manipulate strings in Python.



String Formatting in Python Using Format Specifiers

Like other programming languages, we can also use format specifiers to format strings in Python. Format specifiers are used with the % operator to perform string formatting.

The % operator, when invoked on a string, takes a variable or tuple of variables as input and places the variables in the specified format specifiers. Following is a table of the most commonly used format specifiers for different types of input variables.

NOTE: To format a string in Python using format specifiers, We put the format specifier in the predefined string at the position where the variable is to be inserted.

Code:

```
myCode = 1117
myName = "Scaler"
myVar = 1234
myStr = "Variable is {2} and Code of {0} is {1}".format(myName, myCode, myVar)
print(myStr)
```

Output –
Variable is 1234 and Code of Scaler is 1117

We will specify the desired number of format specifiers to insert two or more variables into the string.

Then we will pass a tuple of variables as input to the % operator.

The % operator inserts the variables at the specified positions in a serial manner as follows.

Code:

```
myCode = 1117
myName = "Scaler"
myStr = "Code of %s is %d" % (myName, myCode)
print(myStr)
```

Output –
Code of Scaler is 1117

Note:

Specifying variables with the correct data type for each format specifier is important. Otherwise, the TyperError exception is raised as follows.

Raw and Unicode Strings in Python

Python strings become raw strings when they are prefixed with r or R, such as r'...' and R'...'. Raw strings treat backslashes (/) as literal characters.

Raw strings are useful for strings with a lot of backslashes, like regular expressions or directory paths.

What is a Raw String

Python raw strings are created by prefixing string literals with 'r' or 'R'. Raw Python strings treat backslashes / as literals. Backslashes are useful when we don't want them treated as escape characters in a string.

Example:

```
str = "This is a \n normal string example"
print(str)
raw_str = r"This is a \n raw string example"
print(raw_str)
```

Output:

```
This is a
normal string example
This is a \n raw string example
```

Unicode Strings

Unicode is a standard encoding system that is used to represent characters from almost all languages. Every Unicode character is encoded using a unique integer code point between 0 and 0x10FFFF. A Unicode

string is a sequence of zero or more code points.

Example

hindi_str=u “नमस्ते”

Print(hindi_str)

Output

नमस्ते

Escape Sequences in Python

The escape sequence is used to escape some of the characters present inside a string.

Suppose we need to include both double quote and single quote inside a string,

example = "He said, "What's there?""

print(example) # throws error

Since strings are represented by single or double quotes, the compiler will treat "He said, " as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character \ in Python.

example

```
# escape double quotes
example = "He said, \"What's there?\""
# escape single quotes
example = 'He said, "What\'s there?"'
print(example)
# Output: He said, "What's there?“
```

Assignment Q1

Explain different types of escape sequence with example

Python String Formatting (f-Strings)

Python f-Strings make it really easy to print values and variables. For example,

```
name = ‘Jack'
country = 'UK'
print(f'{name} is from {country}’)
```

Output

Jack is from UK

Here, f'{name} is from {country}' is an f-string.

This new formatting syntax is powerful and easy to use.

Unit 3

Lists

- A list can be defined as a collection of values or items of different types.
- python lists are **mutable** type which implies that we may modify its element after it has been formed. the items in the list are separated with the comma (,) and enclosed with the square brackets [].

Characteristics of a Python List

The various characteristics of a list are:

1. **Ordered:** Lists maintain the order in which the data is inserted.
2. **Mutable: In** list element(s) are changeable. It means that we can modify the items stored within the list.
3. **Heterogenous:** Lists can store elements of various data types.
4. **Dynamic:** List can expand or shrink automatically to accommodate the items accordingly.
5. **Duplicate Elements:** Lists allow us to store duplicate data.

Creating Lists in Python

A list is created by placing the items/ elements separated by a comma (,) between the square brackets ([ ]).

create Lists in Python in different ways.

```
# Creation of a List in Python

# Creating an empty List
empty_List = []

# Creating a List of Integers
integer_List = [26, 12, 97, 8]

# Creating a List of floating point numbers
float_List = [5.8, 12.0, 9.7, 10.8]
# Creating a List of Strings
string_List = ["Interviewbit", "Preparation", "India"]

# Creating a List containing items of different data types
List = [1, "Interviewbit", 9.5, 'D']

# Creating a List containing duplicate items
duplicate_List = [1, 2, 1, 1, 3,3, 5, 8, 8]
```

Accessing Values/Elements in List in Python

- Elements stored in the lists are associated with a unique integer number known as an index.
- The first element is indexed as 0, and the second is 1, and so on. So, a list containing six elements will have an index from 0 to 5.
- For accessing the elements in the List, the index is mentioned within the index operator ([ ]) preceded by the list's name.
- Another way we can access elements/values from the list in python is using a negative index.
- The negative index starts at -1 for the last element, -2 for the last second element, and so on.

Example

```
# Creating a List with
# the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of
    Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6,
    'Geeks']
print("\nList with the use of
    Mixed Values: ")
print(List)
```

Output

- List with the use of Numbers: [1, 2, 4, 4, 3, 3, 3, 6, 5]
- List with the use of Mixed Values: [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

List Operations in Python

1. Joining list

Joining two lists is very easy just like you perform addition, literally ;-). The concatenation operator +, when used with two lists, joins two lists. Consider the example given below :

The + operator concatenates two lists and creates a new list

>>>lst1=[1,3,5]

>>>lst2=[6,1,0]

>>> lst1+ lst2

[1, 3, 5, 6, 7, 8]

- As you can see that the resultant list has firstly elements of first list lst1 and followed by elements of second list lst2. You can also join two or more lists to form a new list, e.g.,
- >>> lst1 = [10, 12, 14]
- >>> lst2 = [20, 22, 24]
- >>> lst3 = [30, 32, 34]
- >>> lst = lst1 + lst2 + lst3
- >>> lst
- [10, 12, 14, 20, 22, 24, 30, 32, 34]

2. **Repeating or Replicating Lists**

Like strings, you can use * operator to replicate a list specified number of times,

e.g.,

```
>>>list lst1 = [1, 3, 5])

>>> lst1 * 3

[1, 3, 5, 1, 3, 5, 1, 3, 5]
```

3. Slicing the Lists

List slices, like string slices are the sub part of a list extracted out. You can use indexes of list elements to create list slices as per following format :

Syntax:

seq = L[start:stop]

Example:

```
>>> lst =[10, 12, 14, 20, 22, 24, 30, 32, 34]

>>> seq = lst [3: -3]

>>> seq

[20, 22, 24]
```

Built-in Functions on Lists;

- Len()
- Index()
- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Clear()
- Count()
- Reverse()
- sort()
- Sorted()
- Min(), max(), sum()

Implementation of Stacks and Queues using Lists

Stack is a abstract data type that can be defined as a linear collection of data items that supports last in and first out (LIFO) semantics for insertion and deletion of data elements.

Implementation of Stack using list in Python

Stack in python can be implemented in various ways using existing forms of data structures. Python stack can be implemented using the following ways:

- **Append()(push)**
- **Pop()(delete**)

Stack in Python can be implemented simply by using a list data structure. Stack function push() can be mimicked by using the **append()** function which adds the elements to the top of the stack.

We can use the pop() method of the list data type to write the pop method that pops out the topmost element from the stack.

note: The list stores the new element in the next block to the last one. If the list grows out of a block of memory then Python allocates some memory. That's why the list becomes slow.

In below mentioned Python program, the stack has been implemented in Python and the basic operational function assumes that the end of the list will hold the top element of the stack.

```
# Python code to demonstrate
    Implementing
# stack using list
stack = ["Amar", "Akbar", "Anthony"]
stack.append("Ram")
stack.append("Iqbal")
print(stack)

# Removes the last item
print(stack.pop())

print(stack)

# Removes the last item
print(stack.pop())

print(stack)
```

Output:

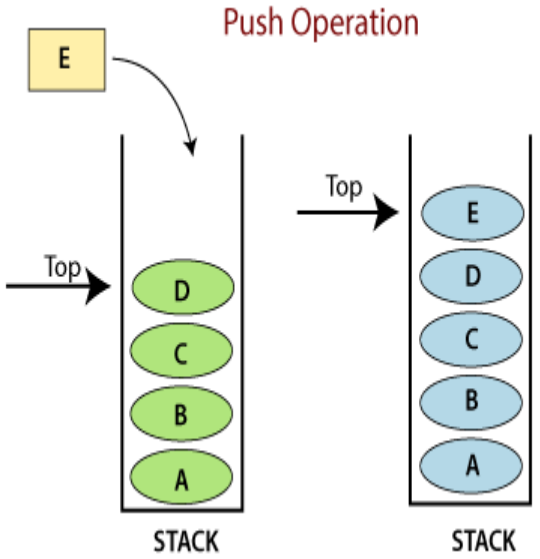
```
['Amar', 'Akbar', 'Anthony', 'Ram', 'Iqbal']

['Amar', 'Akbar', 'Anthony', 'Ram']

['Amar', 'Akbar', 'Anthony']
```

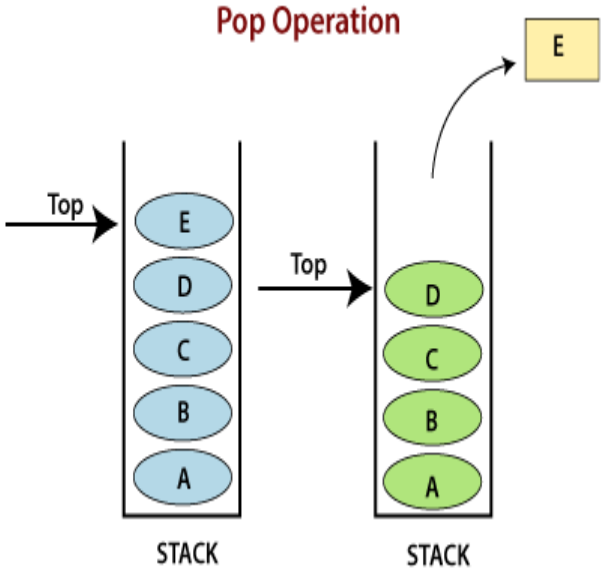
Operations:

- **Adding -** It adds the items in the stack and increases the stack size. The addition takes place at the top of the stack.
- **Deletion -** It consists of two conditions, first, if no element is present in the stack, then underflow occurs in the stack, and second, if a stack contains some elements, then the topmost element gets removed. It reduces the stack size.
- **Traversing -** It involves visiting each element of the stack.

Characteristics:

- Insertion order of the stack is preserved.
- Useful for parsing the operations.
- Duplicacy is allowed.



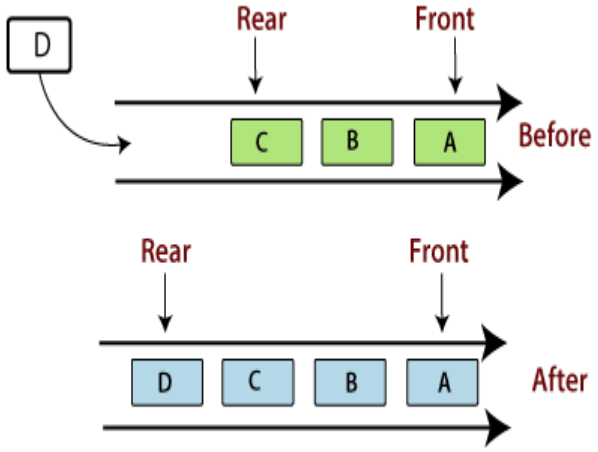


Queue

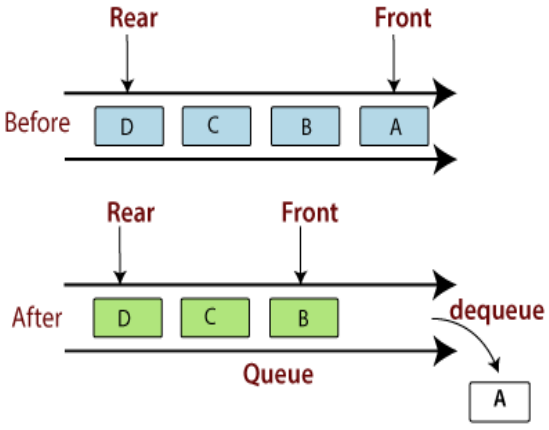
A Queue follows the First-in-First-Out (FIFO) principle. It is opened from both the ends hence we can easily add elements to the back and can remove elements from the front.

To implement a queue, we need two simple operations:

- **enqueue -** It adds an element to the end of the queue.
- **dequeue -** It removes the element from the beginning of the queue.



Queue Enqueue



Queue Dequeue

Operations on Queue

- **Addition -** It adds the element in a queue and takes place at the rear end, i.e., at the back of the queue.
- **Deletion -** It consists of two conditions - If no element is present in the queue, Underflow occurs in the queue, or if a stack contains some elements then element present at the front gets deleted.
- **Traversing - I**t involves to visit each element of the queue.

Characteristics

- Insertion order of the queue is preserved.
- Duplicacy is allowed.
- Useful for parsing CPU task operations.

```
# Python code to demonstrate
    Implementing
# Queue using list
queue = ["Amar", "Akbar",
    "Anthony"]
queue.append("Ram")
queue.append("Iqbal")
print(queue)

# Removes the first item
print(queue.pop(0))

print(queue)

# Removes the first item
print(queue.pop(0))

print(queue)
```

Output :

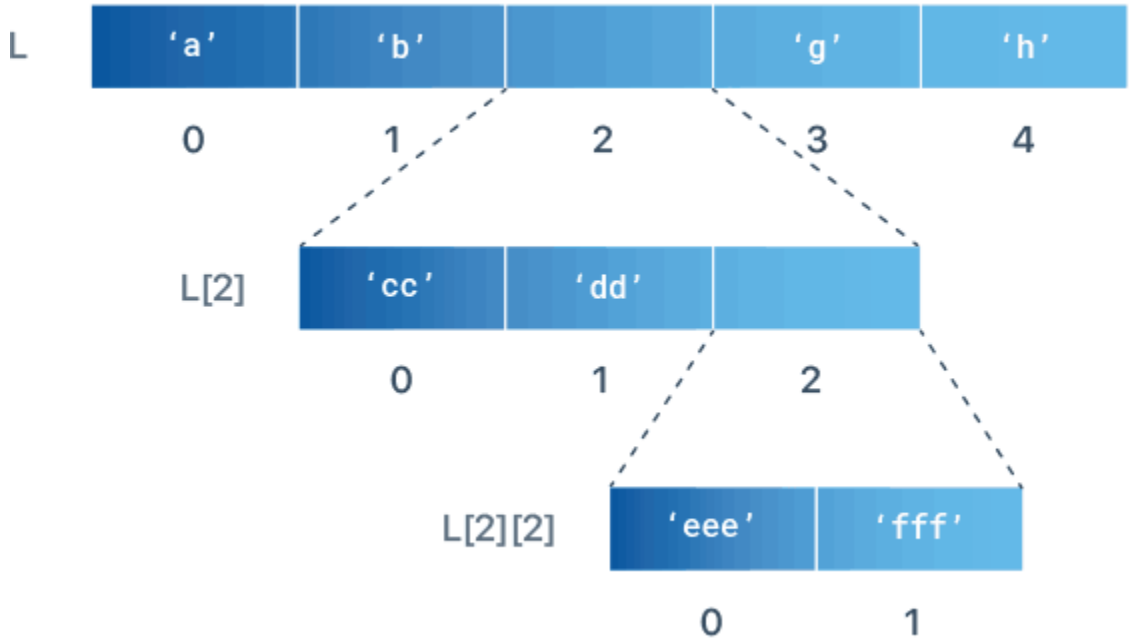
```
['Amar', 'Akbar', 'Anthony', 'Ram', 'Iqbal']

['Akbar', 'Anthony', 'Ram', 'Iqbal']

['Anthony', 'Ram', 'Iqbal']
```

Nested List

a list storing several other lists as its items. It is known as a **nested list**. The nested list is created by placing the various lists separated by comma (,) within the square brackets ([ ]).

Example :

L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']




```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[2])
# Prints ['cc', 'dd', ['eee', 'fff']]
print(L[2][2])
 # Prints ['eee', 'fff']
print(L[2][2][0])
# Prints eee
```

Dictionaries

- Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

Or

- Dictionaries are mutable, unordered collections with elements in the form of key : value pairs that associate keys to values.
- Dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value.
- Example:

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(Dict)
```

Creating a Dictionary

- In Python, a dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by a 'comma'.
- The dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**.
- Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.
- **Note –** Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.
- **example:**

```
Dict = {"Name": "Gayle", "Age": 25}
```

- In the above example dictionary **Dict**, The keys **Name** and **Age** are the strings which comes under the category of an immutable object.

Accessing the dictionary values

To access data contained in lists and tuples, indexing has been studied. The keys of the dictionary can be used to obtain the values because they are unique from one another. The following method can be used to access dictionary values.

Example :

```
Employee = {"Name": "Dev", "Age": 20,
"salary":45000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s"
%Employee["Name"])
print("Age : %d"
%Employee["Age"])
print("Salary : %d"
%Employee["salary"])
print("Company : %s"
%Employee["Company"])
```

out put

```
<class 'dict'>
printing Employee data ....
Name : Dev
Age : 20
Salary : 45000
Company : WIPRO
```

Characteristics of Dictionary

1. Unordered Set

A dictionary is a unordered set of key: value pairs. Its values can contain references to any type of object.

2. Not a Sequence

Unlike a string, list and tuple, a dictionary is not a sequence because it is unordered set of elements.

3. Indexed by Keys, Not Numbers

Dictionaries are indexed by keys. According to Python, a key can be "any non-mutable type.

4. Keys must be Unique

Each of the keys within a dictionary must be: 6 unique. Since keys are used to identify values in a dictionary, there cannot be duplicate keys in a dictionary.

5. Dictionaries are Mutable

Like lists, dictionaries are also mutable. We can change the value of a certain key "in place" using the assignment statement.

6. Internally Stored as Mappings

Internally, the key: value pairs of a dictionary are associated with one another with some internal function (called hash-function). This way of linking is called mapping.

Assignment questions

1. Operations on Dictionaries
2. Adding Deleting and Updating Dictionaries with example.

Built-in Functions on Dictionaries

- **all()**
- any()
- len()
- cmp()
- sorted()
- **type(variable)**

1. all() function

- The Python all() function returns true if all the elements of a given iterable (List, Dictionary, Tuple, set, etc.) are True otherwise it returns False.
- It also returns True if the iterable object is empty.

Syntax: all(iterable)

Were ,

Iterable: It is an iterable object such as a dictionary,tuple,list,set,etc.

Returns: boolean

Example:

```
# All elements of dictionary are true
d = {1: "Hello", 2: "Hi"}
print(all(d))
# All elements of dictionary are false
d = {0: "Hello", False: "Hi"}
print(all(d))
# Some elements of dictionary
# are true while others are false
d = {0: "Salut", 1: "Hello", 2: "Hi"}
print(all(d))
# Empty dictionary
d = {}
print(all(d))
# all() with condition - to check if all
    letters of word 'time' are there
l = {"t":1, "i":1, "m":2, "e":0}
print(all(ele > 0 for ele in l.values()))
```

Output

```
True
False
False
True
False
```

2. any() function

- Python any() function returns True if any of the elements of a given iterable(List, Dictionary, Tuple, set, etc) are True else it returns False.

Syntax: *any(iterable)*

- ***Iterable:*** *It is an iterable object such as a dictionary, tuple, list, set, etc.*
- ***Returns:*** *Returns True if any of the items is True.*

Example:

```
# All keys of dictionary are true
d = {1: "Hello", 2: "Hi"}
print(any(d))

# All keys of dictionary are false
d = {0: "Hello", False: "Hi"}
print(any(d))

# Some keys of dictionary
# are true while others are false
d = {0: "Salut", 1: "Hello", 2: "Hi"}
print(any(d))

# Empty dictionary
d = {}
print(any(d))
```

Output:

```
True
False
True
False
```

3. len() function

- Python dictionary method **len()** gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
- **Syntax**

len(dict)

dict – This is the dictionary, whose length needs to be calculated.

Example

```
dict = {'Name': 'Zara', 'Age': 7};
print ("Length :%d " %len (dict))
```

output

Length : 2

4. cmp() function

- Python dictionary method **cmp()** compares two dictionaries based on key and values.
- **Syntax**

cmp(dict1, dict2)

Parameters

- **dict1** – This is the first dictionary to be compared with dict2.
- **dict2** – This is the second dictionary to be compared with dict1.

Example

```
dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = {'Name': 'Mahnaz', 'Age': 27};
dict3 = {'Name': 'Abid', 'Age': 27};
dict4 = {'Name': 'Zara', 'Age': 7};
print "Return Value : %d" % cmp
    (dict1, dict2)
print "Return Value : %d" % cmp
    (dict2, dict3)
print "Return Value : %d" % cmp
    (dict1, dict4)
```

Output

```
Return Value : -1
Return Value : 1
Return Value : 0
```

4. Sorted() Function

- There are two elements in a Python dictionary-keys and values. You can sort the dictionary by keys, values, or both. In this article, we will discuss the methods of sorting dictionaries by key or value using Python.

- **Example**

```
# Creates a sorted dictionary
    (sorted by key)
from collections import
    OrderedDict


dict = {'ravi': '10', 'rajnish': '9',
            'sanjeev': '15', 'yash': '2',
    'suraj': '32'}
dict1 =
    OrderedDict(sorted(dict.items
    ()))
print(dict1)
```

Output

```
OrderedDict([('rajnish', '9'),
    ('ravi', '10'), ('sanjeev', '15'),
    ('suraj', '32'), ('yash', '2')])
```

5. **type(variable)**

- Python dictionary method **type()** returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.
- **Syntax**

type(dict)

Parameters

- **dict** – This is the dictionary.

Example:

```
dict = {'Name': 'Zara', 'Age': 7};
print "Variable Type : %s" % type
    (dict)
```

Output

Variable Type : <type 'dict'>

Methods on Dictionaries

1. dict.clear()
2 dict.copy()
3 dict.fromkeys()
4 dict.get(key, default=None)
5 dict.has_key(key)
6 dict.items()
7 dict.keys()
8 dict.setdefault(key, default=None)
9 dict.update(dict2)
10 dict.values()

1. **dict.clear()**

Python dictionary method clear() removes all items from the dictionary.

Syntax

Following is the syntax for clear() method −

dict.clear()

Example:

```
dict = {'Name': 'Zara', 'Age': 7};
print "Start Len : %d" %  len(dict)
dict.clear()
print "End Len : %d" %  len(dict)
```

Out put

Start Len : 2

End Len : 0

2 dict.copy()

Python dictionary method copy() returns a shallow copy of the dictionary.

Syntax

Following is the syntax for copy() method −

dict.copy()

Example:

```
dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = dict1.copy()
print "New Dictionary : %s" %
    str(dict2)
```

Output:

New Dictionary : {'Age': 7, 'Name': 'Zara'}

3 dict.fromkeys()

Python dictionary method fromkeys() creates a new dictionary with keys from seq and values set to value.

Syntax

dict.fromkeys(seq[, value])

Parameters

seq − This is the list of values which would be used for dictionary keys preparation.

value − This is optional, if provided then value would be set to this value

Example:

```
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print "New Dictionary : %s" %
    str(dict)

dict = dict.fromkeys(seq, 10)
print "New Dictionary : %s" %
    str(dict)
```

Output

```
New Dictionary : {'age': None,
    'name': None, 'sex': None}
New Dictionary : {'age': 10,
    'name': 10, 'sex': 10}
```

4 dict.get(key, default=None)

Python dictionary method get() returns a value for the given key. If key is not available then returns default value None.

Syntax

dict.get(key, default = None)

Parameters

key – This is the Key to be searched in the dictionary.

default – This is the Value to be returned in case key does not exist.

Example:

```
dict = {'Name': 'Zabra', 'Age': 7}
print "Value : %s" % dict.get('Age')
print "Value : %s" % dict.get('Education', "Never")
```

Output:

Value : 7

Value : Never

5 dict.has_key(key)

Python dictionary method has_key() returns true if a given key is available in the dictionary, otherwise it returns a false.

Syntax

dict.has_key(key)

Parameters

key − This is the Key to be searched in the dictionary.

Example:

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" % dict.has_key('Age')
print "Value : %s" % dict.has_key('Sex')
```

Output:

Value : True

Value : False

6 dict.items()

Python dictionary method items() returns a list of dict's (key, value) tuple pairs

Syntax

```
dict.items()
```

Example:

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" % dict.items()
```

Output:

```
Value : [('Age', 7), ('Name', 'Zara')]
```

7 dict.keys()

Python dictionary method keys() returns a list of all the available keys in the dictionary.

Syntax

```
dict.keys()
```

Example:

```
dict = {'Name': 'Zara',
    'Age': 7}
print "Value : %s" %
    dict.keys()
```

Output:

```
Value : ['Age', 'Name']
```

8 dict.setdefault(key, default=None)

Python dictionary method setdefault() is similar to get(), but will set dict[key]=default if key is not already in dict.

Syntax

dict.setdefault(key, default=None)

Parameters

key − This is the key to be searched.

default − This is the Value to be returned in case key is not found.

Example:

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %
    dict.setdefault('Age', None)
print "Value : %s" %
    dict.setdefault('Sex', None)
```

Output:

```
Value : 7
Value : None
```

9 dict.update(dict2)

Python dictionary method update() adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

Syntax

dict.update(dict2)

Parameters

dict2 − This is the dictionary to be added into dict.

Example:

```
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print "Value : %s" %  dict
```

Output:

```
Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}
```

10 dict.values()

Python dictionary method **values()** returns a list of all the values available in a given dictionary.

Syntax

dict.values()

Example:

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" % dict.values()
```

Output:

```
Value : [7, 'Zara']
```

Populating Dictionaries

populate a **dictionary** using for loop and a **dictionary** comprehension and a zip() function.

Method 1:Using Zip And For Loop

- The zip() method takes one value from each list passed and returns it to the for loop during each iteration.
- The returned values are created as a key and value, and finally, it is assigned to the dictionary object.

Example:

```
keys = ["One", "Two", "Three", "Four"]
values = [1,2,3,4]
yourdict = {k:v for k,v in zip(keys, values)}
Yourdict
```

Output:

```
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4}
```

Method 2:Using Dictionary Constructor & Zip

- This section teaches you how to use the dictionary constructor with the zip() method instead of the dictionary comprehension.
- Pass the zip method output to the dictionary constructor, as demonstrated in the following code.

Example:

```
keys = ["One", "Two",
    "Three", "Four"]
values = [1,2,3,4]
yourdict = dict(zip(keys,
    values))
yourdict
```

Output:

```
{'One': 1, 'Two': 2, 'Three':
    3, 'Four': 4}
```

Traversing or Iterating Dictionary

- Iterating through a dictionary means, visiting each key-value pair in order.
- It means accessing a Python dictionary and traversing each key-value present in the dictionary.
- Iterating a dictionary is a very important task if you want to properly use a dictionary.

There are multiple ways to iterate through a dictionary, we are discussing some generally used methods for dictionary iteration in Python which are the following:

- Iterate Python dictionary using build.keys()
- Iterate through all values using .values()
- Looping through Python Dictionary using for loop
- Iterate key-value pair using items()
- Access key Using map() and dict.get
- Access key in Python Using zip()
- Access key Using Unpacking of Dict

PYTHON PROGRAMMING

UNIT - 4

BY

MR. PRUTHVI RAJ A

DEPT OF COMPUTER SCIENCE

File Handling in Python

•File handling is the process of saving data in a Python program in the form of outputs or inputs.

• The python file can be store in the form of binary file or a text file.

•There are six different types of modes are available in Python programming language.

1. r – read an existing file by opening it.
2. w – initiate a write operation by opening an existing file. If the file already has some data in it, it will be overwritten; if it doesn’t exist, it will be created.
3. a – Open a current file to do an add operation. Existing data won’t be replaced by it.
4. r+ – Read and write data to and from the file. It will replace any prior data in the file.
5. w+ – To write and read data, use w+. It will replace current data.
6. a+ – Read and add data to and from the file. Existing data won’t be replaced by it.

Types of Files

Python allow us to create and manage three types of file

1. TEXT FILE
2. BINARY FILE
3. CSV (Comma separated values) files

TEXT FILE

What is Text File?

• A text file is usually considered as sequence of lines.

• Line is a sequence of characters (ASCII or UNICODE), stored on permanent storage media.

• The default character coding in python is ASCII each line is terminated by a special character, known as End of Line (EOL).

• At the lowest level, text file will be collection of bytes.

• Text files are stored in human readable form and they can also be created using any text editor.

BINARY FILE

What is Binary File?

- A binary file contains arbitrary binary data i.e. numbers stored in the file, can be used for numerical operation(s).
- So when we work on binary file, we have to interpret the raw bit pattern(s) read from the file into correct type of data in our program.
- In the case of binary file it is extremely important that we interpret the correct data type while reading the file.
- Python provides special module(s) for encoding and decoding of data for binary file.

CSV files

What is CSV File?

- A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values.
- Each line of the file is a data record.
- Each record consists of one or more fields, separated by commas.
- The use of the comma as a field separator is the source of the name for this file format.
- CSV file is used to transfer data from one application to another.
- CSV file stores data, both numbers and text in a plain text.

Operations on Files–

- Create
- Open
- Read
- Write
- Close Files

Opening and Closing a File in Python

1. open() Function:

This function takes two arguments. First is the filename along with its complete path, and the other is access mode. This function returns a file object.

Syntax:

```
open(filename, mode)
```

Important points:

The file and python script should be in the same directory. Else, you need to provide the full path of the file.

By default, the access mode is read mode if you don't specify any mode.

Access mode tells the type of operations possible in the opened file. There are various modes in which we can open a file. Let's see them:

No.	Modes	Description
1.	**r**	Opens a file in read-only mode. The pointer of the file is at the beginning of the file. This is also the default mode.
2.	**rb**	Same as **r** mode, except this opens the file in binary mode.
3.	**r+**	Opens the file for reading and writing. The pointer is at the beginning of the file.
4.	**rb+**	Same as **r+** mode, except this, opens the file in binary mode.
5.	**w**	Opens the file for writing. Overwrites the existing file and if the file is not present, then creates a new one.
6.	**wb**	Same as **w** mode, except this opens the file in binary format.
7.	**w+**	Opens the file for both reading and writing, rest is the same as **w** mode.
8.	**wb+**	Same as **w+** except this opens the file in binary format.
9.	**a**	Opens the file for appending. If the file is present, then the pointer is at the end of the file, else it creates a new file for writing.
10.	**ab**	Same as **a** mode, except this opens the file in binary format.
11.	**a+**	Opens the file for appending and reading. The file pointer is at the end of the file if the file exists, else it creates a new file for reading and writing.
12.	**ab+**	Same as **a+** mode, except this, opens the file in binary format.

Example of Opening and Closing Files in Python

Code:

When the file is in the same folder where the python script is present. Also access mode is 'r' which is read mode.

```
file = open('test.txt',mode='r')
```

When the file is not in the same folder where the python script is present. In this case, the whole path of the file should be written.

```
file = open('D:/data/test.txt',mode='r')
```

It is general practice to close an opened file as a closed file reduces the risk of being unwarrantedly updated or read. We can close files in python using the close function.

Let's discuss it.

2. close() function:

This function doesn't take any argument, and you can directly call the close() function using the file object.

It can be called multiple times, but if any operation is performed on the closed file, a "ValueError" exception is raised.

Syntax:

```
file.**close()**
```

You can use the **'with'** statement with open also, as it provides better exception handling and simplifies it by providing some cleanup tasks.

Also, it will automatically close the file, and you don't have to do it manually.

Example using with statement

Code:

```
with open("test.txt", mode='r') as f:
    # perform file operations
```

The method shown in the above section is not entirely safe. If some exception occurs while opening the file, then the code will exit without closing the file. A safer way is to use a try-finally block while opening files.

Code:

```
try:
    file = open('test.txt',mode='r')
    # Perform file handling operations
finally:
    file.close()
```

Now, this guarantees that the file will close even if an exception occurs while opening the file.

So, you can use the ‘with’ statement method instead. Any of the two methods is good.

3. Python – Create New File

To create a new file with Python, use open() method with “x” as second parameter and the filename as first parameter.

```
myfile = open("complete_filepath", "x")
```

The open() method with options shown in the above code snippet creates an empty file

Example 1: Create a New File using open()

In the following example, we will create a new file named sample.txt.

Python Program

```
#open file
f = open("sample.txt", "x")
#close file
f.close
```

A new file will be created in the present working directory. You can also provide a complete path to the file if you would like your file to be created at an absolute path in the computer.

Example 2: Create a New File with the same name as that of existing file

In the following example, we will try creating a new file sample.txt. But this file is already present in the same location.

Python Program

```
f = open("sample.txt", "x")
f.close
```

You will get FileExistsError with a similar stack trace as below.

Output

```
Traceback (most recent call last):
  File "example.py", line 1, in <module>
    f = open("sample.txt", "x")
FileExistsError: [Errno 17] File exists: 'sample.txt'
```

4. Read Text File in Python

To read text file in Python, follow these steps.

- Call open() builtin function with filepath and mode passed as arguments. open() function returns a file object.
- Call read() method on the file object. read() returns a string.
- The returned string is the complete text from the text file.

Examples

1. Read a text file “sample.txt”

In the following Python program, we will open sample.txt file in read mode. We will read all the contents of the text file and print the text to the console.

Python Program

```
fileObject = open("sample.txt", "r")
data = fileObject.read()
print(data)
```

Output

2. Read only few characters from the text file

If you need to read only specific number of characters, say N number of characters, present at the starting of the file, pass N (number) as argument to read() function.

In the following Python program, we will read first 20 characters in the file.

Python Program

```
f = open("sample.txt", "r")
data = f.read(20)
print(data)
```

Output

```
Welcome to pythonexa
```

read(20) function returned the first 20 characters from the text file.

3. Read the file in text mode

read, write and execute modes are based on the permissions. There are text and binary based on the nature of content.

In the following example, we will open the file in text mode explicitly by providing “t” along with the read “r” mode.

Python Program

```
f = open("sample.txt", "rt")

data = f.read()

print(data)
```

Output

Welcome to pythonexamples.org

5. Closing a File

The close() function is used to close a file.

Syntax of the Python close function:

```
File_object.close()
```

Example:

```
j=open(“simple.txt”,”r”)

k=j.read()

print(k)

j.close()
```

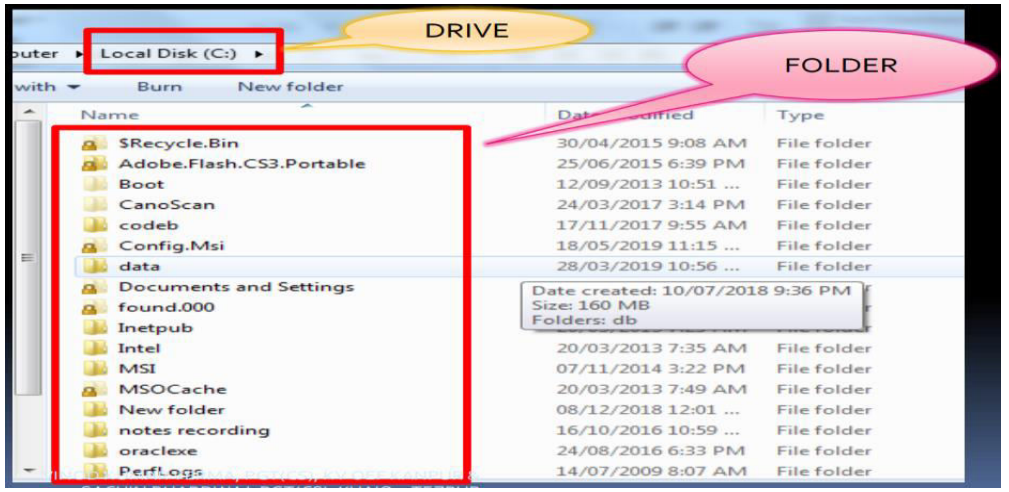
Output:

Hello Nagarjuna

File Names and Paths

To understand path we must be familiar with the terms: DRIVE, FOLDER/DIRECTORY, FILES.

Our hard disk is logically divided into many parts called Drives like C drive, D drive etc.

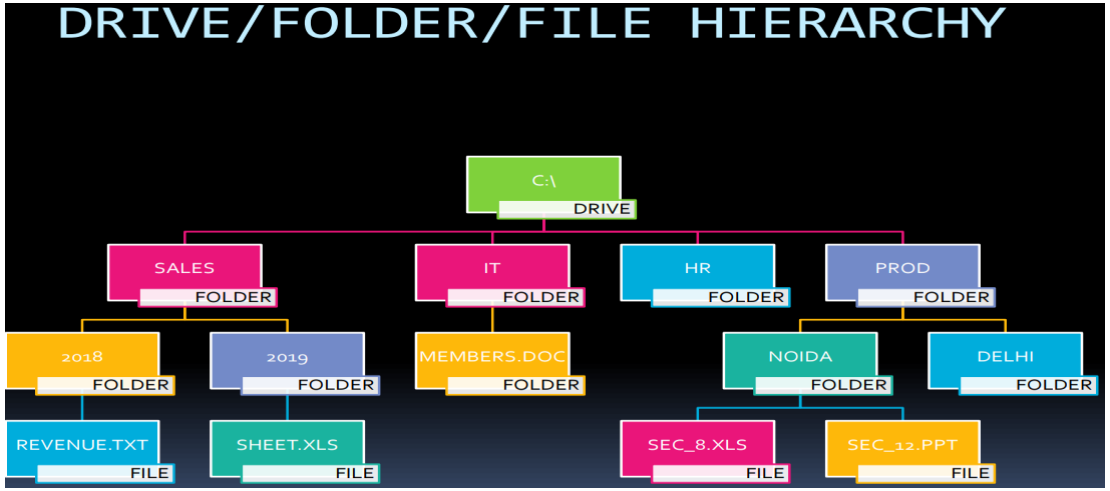


The drive is the main container in which we put everything to store.

The naming format is DRIVE_LETTER:

For Ex C: , D:

- Drive is also known as ROOT DIRECTORY.
- Drive contains Folders and files
- Folder contains sub-folders or files.
- Files are the actual data container.



ABSOLUTE PATH:

Absolute path is the full address of any file or folder

From the drive i.e. from root folder . It is like

Drive_name:\Folder\Folder…..\filename

For eg absolute path of file **Python_unit4.ppt will be**

C:\Users\panka\OneDrive\Documents\NCMS\NEP_notes\Python_unit4.ppt

RELATIVE PATH:

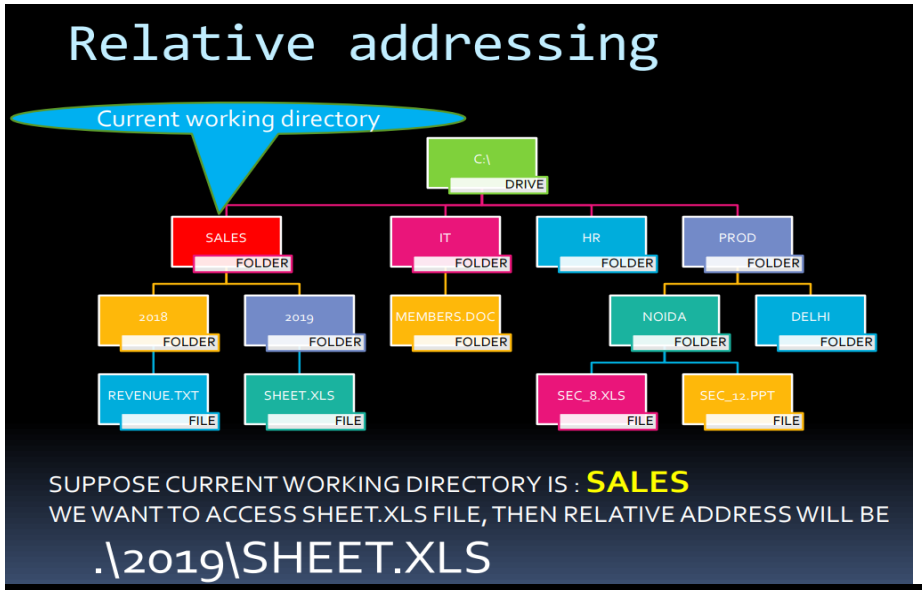
Relative path is the location of file/folder from the current folder.

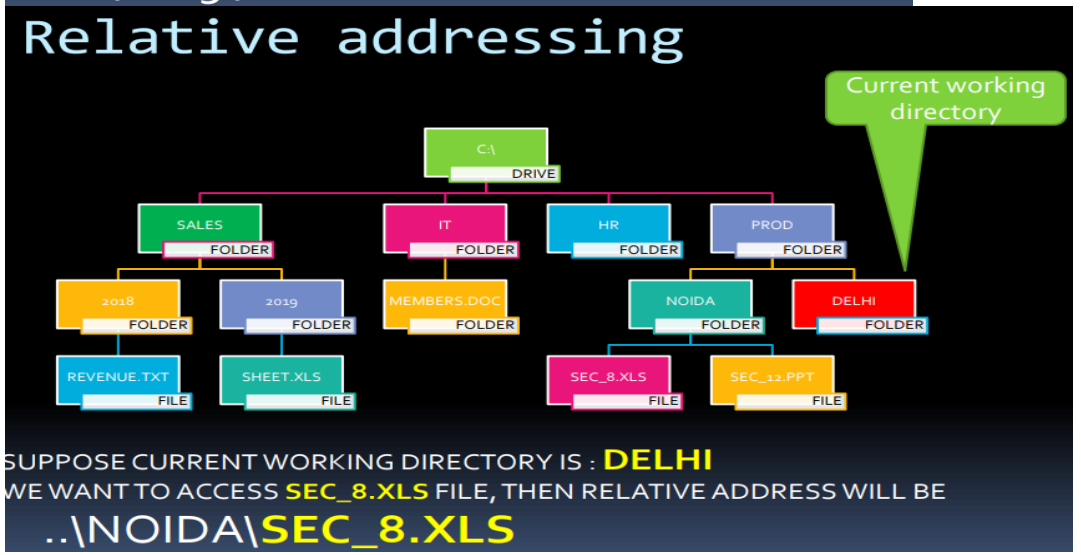
To use relative path special symbols are:

Single dot(.):it refers to current folder

Double dot(..): it refers to parent folder.

Backslash(\):first backslash before(.) and double dot(..)refers to the ROOT folder.





Format operator

- The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings.
- **Example:**

The easiest way to do that is with str:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the format operator uses the syntax % symbol followed by one or more format codes and optionally a format specifier.

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

Python Object-Oriented Programming (OOP)

What Is Object-Oriented Programming in Python?

- Object-Oriented Programming (OOP) is a programming prototype in which we can think about complex problems as objects.
- A prototype is a theory that supplies the base for solving problems.
- So when we're talking about OOP, we're referring to a set of concepts and patterns we use to solve problems with objects.
- An object in Python is a single collection of data (attributes) and behavior (methods). You can think of objects as real things around you. For example, consider calculators:

- As you may notice, the data (attributes) are always nouns, while the behaviors (method) are always verbs.
- This compartmentalization is the central concept of Object-Oriented Programming. You build objects that store data and contain specific kinds of functionality.

Why Do We Use Object-Oriented Programming in Python?

OOP allows you to create secure and reliable software. Many Python frameworks and libraries use this prototype to build their codebase. Some examples are pandas, NumPy, etc.

The main advantages of using OOP in Python.

➢ All Modern Programming Languages Use OOP

This paradigm is language-independent. If you learn OOP in Python, you'll be able to use it in the following:

Java, PHP, JavaScript

➢ OOP Allows You to Code Faster

Coding faster doesn't mean writing fewer lines of code. It means you can implement more features in less time without compromising the stability of a project.

Object-Oriented programming allows you to reuse code by implementing abstraction. This principle makes your code more concise and legible.

➢ OOP Helps You Avoid Spaghetti Code

OOP gives us the possibility of compressing all the logic in objects, therefore avoiding long pieces of nested if's.

➢ OOP Improves Your Analysis of Any Situation

Once you get some experience with OOP, you'll be able to think of problems as small and specific objects.

Class and Objects in Python

- ✓ Suppose you wish to store the number of books you have, you can simply do that by using a variable. Or, say you want to calculate the sum of 5 numbers and store it in a variable, well, that can be done too!
- ✓ Primitive data structures like numbers, strings, and lists are designed to store simple values in a variable. Suppose, your name, or square of a number, or count of some marbles (say).
- ✓ But what if you need to store the details of all the Employees in your company?
- ✓ **For example**, you may try to store every employee in a list, you may later be confused about which index of the list represents what details of the employee(e.g. which is the name field, or the empID etc.)

Example:

employee1 = ['John ', 104120, "Developer", "Dept. 2A"]

employee2 = ['Mark', 211240, "Database Designer", "Dept. 11B"]

employee3 = [‘Smith ', 131124, "Manager", "Dept. 2A"]

Even if you try to store them in a dictionary, after an extent, the whole codebase will be too complex to handle. So, in these scenarios, we use Classes in python.

A **class** is used to create user-defined data structures in Python.

Classes define functions, which are termed methods, that describe the behaviors and actions that an object created from a class can perform.

Classes make the code more manageable by avoiding complex codebases. It does so, by creating a blueprint.

It defines what properties or functions, any object which is derived from the class should have.

IMPORTANT:

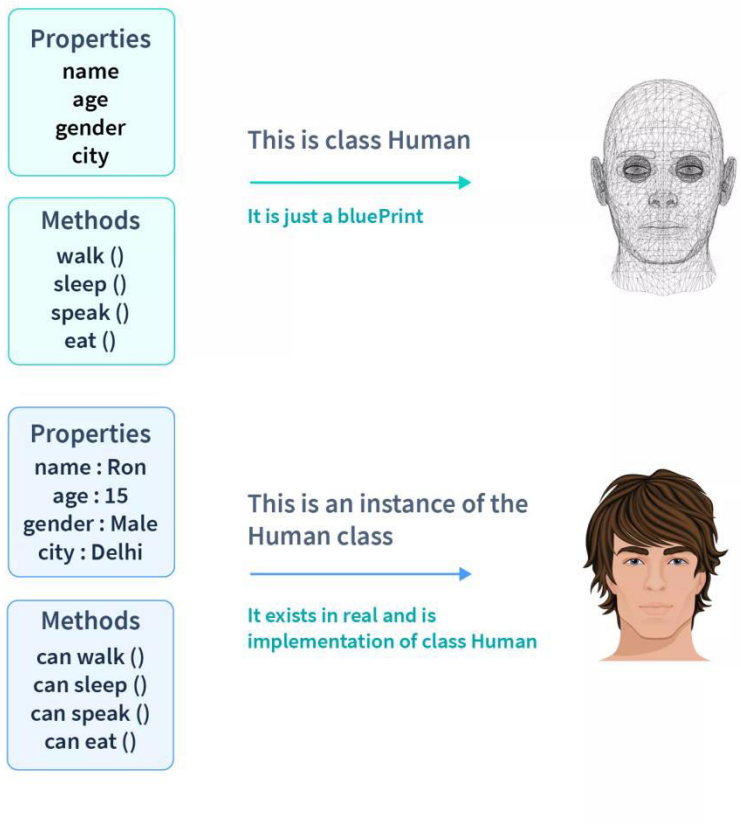
A class just defines the structure of how anything should look. It does not point to anything or anyone in particular.

For example, say, HUMAN is a class, which has suppose -- name, age, gender, city. It does not point to any specific HUMAN out there, but yes, it explains the properties and functions any HUMAN should or any object of class HUMAN should have.

An instance of a class is called **the object.** It is the implementation of the class and exists in real.

An **object** is a collection of data (variables) and methods (functions) that access the data. It is the real implementation of a class.

Example:



Consider this example, here Human is a **class** - It is just a blueprint that defines how Human should be, and not a real implementation.

You may say that "Human" class just exists logically.

- However, **"Ron"** is an object of the Human class (please refer to the image given above for understanding).
- That means, **Ron** is created by using the blueprint of the Human class, and it contains the real data.
- "Ron" exists physically, unlike "Human" (which just exists logically).
- He exists in real, and implements all the properties of the class Human, such as, Ron have a name, he is 15 years old, he is a male, and lives in Delhi.
- Also, Ron implements all the methods of Human class, suppose, Ron can walk, speak, eat, and sleep.
- And many humans can be created using the blueprint of class Human. Such as, we may create 1000s of more humans by referring to the blueprint of the class Human, using objects.

Note:

class = blueprint(suppose an architectural drawing). The Object is an actual thing that is built based on the ‘blueprint’ (suppose a house).

An instance is a virtual copy (but not a real copy) of the object.

When a class is defined, only the blueprint of the object is created, and no memory is allocated to the class.

Memory allocation occurs only when the object or instance is created.

The object or instance contains real data or information.

How to Define a Class in Python?

Classes in Python can be defined by the keyword class, which is followed by the name of the class and a colon.

Syntax:

```
class Human:
    pass
```

'pass' is commonly used as a placeholder, in the place of code whose implementation we may skip for the time being. "pass" allows us to run the code without throwing an error in Python.

GU Interface

- Python offers multiple options for developing GUI (Graphical User Interface).
- Out of all the GUI methods, tkinter is the most commonly used method.
- It is a standard Python interface to the Tk GUI toolkit shipped with Python.
- Python tkinter is the fastest and easiest way to create GUI applications.
- Creating a GUI using tkinter is an easy task.

What is Tkinter?

Tkinter in Python helps in creating GUI Applications with minimum hassle. Among various GUI Frameworks, Tkinter is the only framework that is built-in into Python's Standard Library.

- An important feature in favor of Tkinter is that it is cross-platform, so the same code can easily work on Windows, macOS, and Linux.
- Tkinter is a lightweight module.
- It comes as part of the standard Python installation, so you don't have to install it separately.
- It supports a lot of built-in widgets that can be used directly to create desktop applications.

What are Tcl, Tk, and Tkinter?

Let's try to understand more about the Tkinter module by discussing more about its origin.

- Tkinter is based upon the Tk toolkit, which was originally designed for the Tool Command Language (Tcl). As Tk is very popular thus it has been ported to a variety of other scripting languages, including Perl (Perl/Tk), Ruby (Ruby/Tk), and Python (Tkinter).
- The wide variety of widgets, portability, and flexibility of Tk makes it the right tool which can be used to design and implement a wide variety of simple and complex projects.
- Python with Tkinter provides a faster and more efficient way to build useful desktop applications that would have taken much time if you had to program directly in C/C++ with the help of native OS system libraries.

How to install Tkinter?

- Tkinter comes as **part of standard Python installation**. So if you have installed the latest version of Python, then you do not have to do anything else.
- If you do not have Python installed on your system - Install Python (the current version is **3.11.4** at the time of writing this article) first, and then check for Tkinter.
- You can determine **whether Tkinter is available** for your Python interpreter by attempting to **import the Tkinter module**.

  import tkinter
- If Tkinter is available, then there will be no errors, otherwise, you will see errors in the console.

Using Tkinter to Create Desktop Applications

The basic steps of creating a simple desktop application using the Tkinter module in Python are as follows:

- First of all, import the Tkinter module.
- The second step would be to create a basic window for the desktop application.
- Then you can add different GUI components to the window and functionality to these components or widgets.
- Then enter the main event loop using mainloop() function to run the Desktop application.
- So let's write some code and create a basic desktop application using the Tkinter module in Python.

Hello World tkinter Example

- When you create a desktop application, the first thing that you will have to do is create a new window for the desktop application.
- The main window object is created by the Tk class in Tkinter.
- Once you have a window, you can text, input fields, buttons, etc. to it.

Here's a code example,

```
import tkinter as tk
win = tk.Tk()
win.title('Hello World!')
# you can add widgets here
win.mainloop()
```

The window may look different depending on the operating system.

Tkinter Methods used above:

The two main methods that are used while creating desktop applications in Python are:

1. Tk( )

The syntax for the Tk() method is:

Tk(screenName=None, baseName=None, className='Tk', useTk=1)

This method is used to **create the main window.**

This is how you can use it, just like in the Hello World code example,

win = tkinter.Tk() ## where win indicates name of the main window object

2. The mainloop() Function

This method is used to start the application. The mainloop() function is an **infinite loop** that is used to run the application.

It will wait for **events to occur** and **process the events** as long as the window is not closed.

Common Tkinter Widgets

1. **Button widget**

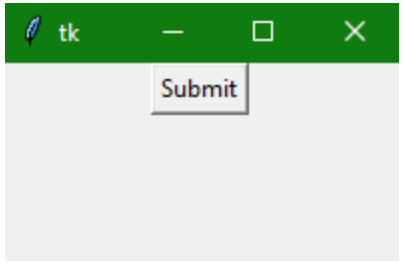
- The **Button widget** in Tkinter is mainly used to **add a button in any GUI Application**.
- In Python, while using the Tkinter button widget, we can easily modify the style of the button like adding a background colors to it, adjusting height and width of button, or the placement of the button, etc. very easily.

- The **syntax** of the button widget is given below,

W = Button(master, options)

Exmple:

```
from tkinter import *
win = Tk()  ## win is a top or parent
    window
win.geometry("200x100")
b = Button(win, text = "Submit")
b.pack()  #using pack() geometry
win.mainloop()
```



Option name	Description
activebackground	This option indicates the background of the button at the time **when the mouse hovers the button**.
bd	This option is used to **represent the width of the border** in pixels.
bg	This option is used to **represent the background color of the button**.
command	The **command option is used to set the function call** which is scheduled at the time **when the function is called**.
activeforeground	This option mainly **represents the font color** of the button when the **mouse hovers the button**.
fg	This option represents the **foreground color of the button**.
font	This option indicates the font of the button.
height	This option indicates the height of the button. This height indicates the **number of text lines** in the case of **text lines** and it indicates **the number of pixels** in the case **of images**.
image	This option indicates the image displayed on the button.
highlightcolor	This option indicates the highlight color when there is a focus on the button
justify	This option is used to indicate the way **by which the multiple text lines are represented**. For left justification, it is set to LEFT and it is set to RIGHT for the right justification, and CENTER for the center justification.
padx	This option indicates the additional padding of the button in the **horizontal direction**.
pady	This option indicates the additional padding of the button in the **vertical direction**.
underline	This option is used to **underline the text of the button**.
width	This option specifies the width of the button. For textual buttons, It exists as a number of letters or for image buttons it indicates the pixels.
wraplength	In the case, if this option's value is **set to a positive number**, the text lines will be wrapped **in order to fit within this length**.
state	This option's value set to **DISABLED** to make the **button unresponsive**. The ACTIVE mainly represents the **active state of the button**.

2. **Label widget**

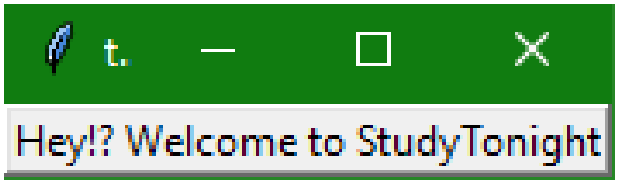
- The label widget in Tkinter is used to display boxes where you can place your images and text.
- The label widget is mainly used to provide a message about the other widgets used in the Python Application to the user.
- You can change or update the text inside the label widget anytime you want.
- This widget uses only one font at the time of displaying some text.

The syntax of the label widget is given below,

W = Label(master,options)

Example:

```
import tkinter
from tkinter import *
win = Tk()
var = StringVar()
label = Label( win, textvariable=var,
    relief=RAISED )
# set label value
var.set("Hey!? Welcome to
    StudyTonight")
label.pack()
win.mainloop()
```



Tkinter Label Widget Options

Following are the options used with label widgets:

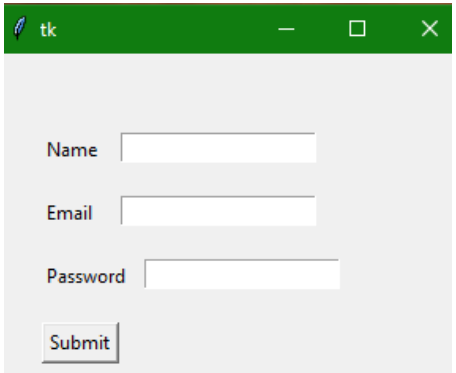
Name of the option	Description
anchor	This option is mainly used for controlling the **position of text in the provided widget size**. The default value is **CENTER which is used to align the text in** center in the provided space.
bd	This option is used **for the border width of the widget**. Its default value is 2 pixels.
bitmap	This option is used to set **the bitmap equals to the graphical object** specified so that now the label can represent the graphics instead of text.
bg	This option is used for the background **color of the widget**.
cursor	This option is used to specify **what type of cursor to show when the mouse is moved over the label.** The default of this option is to use **the standard cursor**.
fg	This option is used to specify the foreground color of the **text that is written inside the widget**.
font	This option **specifies the font type of text** inside the label.
height	This option indicates the **height of the widget**
image	This option indicates the image **that is shown as the label**.

3. **entry widget**

- If you need to get a little bit of text from a user, like a name or an email address or a contact number then use an Entry widget.
- The Entry widget is mainly used to display a small text box that the user can type some text into.
- There are the number of options available to change the styling of the Entry Widget.

The syntax of the entry widget is given below:

w = Entry(master, option=value)



Example:

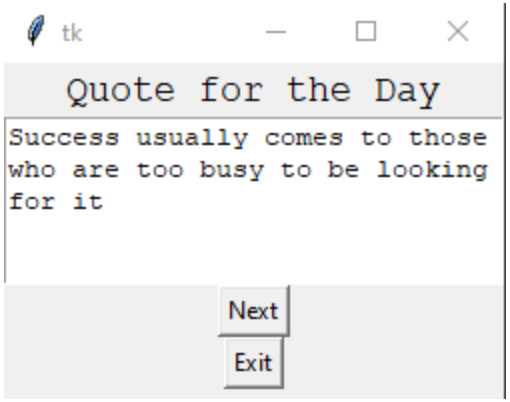
```
from tkinter import *
win = Tk()
win.geometry("400x250")
name = Label(win, text = "Name").place(x = 30,y
    = 50)
email = Label(win, text = "Email").place(x = 30, y
    = 90)
password = Label(win, text = "Password").place(x
    = 30, y = 130)
submitbtn = Button(win, text =
    "Submit",activebackground = "red",
    activeforeground = "blue")
        .place(x = 30, y = 170)
entry1 = Entry(win).place(x = 80, y = 50)
entry2 = Entry(win).place(x = 80, y = 90)
entry3 = Entry(win).place(x = 95, y = 130)
win.mainloop()
```

4. **Text widget**

- The text widget is used to provide a multiline textbox (input box) because in Tkinter single-line textbox is provided using Entry widget.
- You can use various styles and attributes with the Text widget.
- You can also use marks and tabs in the Text widget to locate the specific sections of the text.
- Media files like images and links can also be inserted in the Text Widget.

The syntax of the Text widget is given below:

W = Text(master, options)



Example :

```
import tkinter as tk
from tkinter import *
win = Tk()
#to specify size of window.
win.geometry("250x170")
# To Create a text widget and specify size.
T = Text(win, height = 6, width = 53)
# TO Create label
l = Label(win, text = "Quote for the Day")
l.config(font =("Courier", 14))
Quote = """Success usually comes to those who are
      too busy to be looking for it"""
# Create a button for the next text.
b1 = Button(win, text = "Next", )
# Create an Exit button.
b2 = Button(win, text = "Exit", command =
      win.destroy)
l.pack()
T.pack()
b1.pack()
b2.pack()
# Insert the Quote
T.insert(tk.END, Quote)
tk.mainloop()
```

Name of the option	Description
bd	This option **represents the border width of the widget.**
bg	This option indicates **the background color of the widget.**
exportselection	This option is used **to export the selected text in the selection** of the window manager. If you do not want to export the text **then you can set the value of this option to 0.**
cursor	This option will convert the mouse pointer to **the specified cursor type and it can be set to an** arrow, dot, etc.
font	This option is used **to indicate the font type of the text.**
fg	This option indicates **the text color of the widget**
height	This option indicates **the vertical dimension of the widge**t and it is mainly in the number of lines.
highlightbackground	This option indicates the **highlightcolor at the time when the widget isn't under the focus.**
higlightthickness	This option is used **to indicate the thickness of the highlight**. The default value of **this option is 1.**
highlightcolor	This option **indicates the color of the focus highlight** when the widget is under the focus.
insertbackground	This option is used **to represent the color of the insertion cursor.**

4. **Text:** This can be used for multi-line text fields, to show text, or to take input from the user.

5. **Canvas:** This widget can be used for drawing shapes, images, and custom graphics.

6. **Frame:** This acts as a container widget that can be used to group other widgets together.

7. **Checkbutton:** A checkbox widget used for creating toggle options on or off.

8. **Radiobutton:** This is used to create Radio buttons.

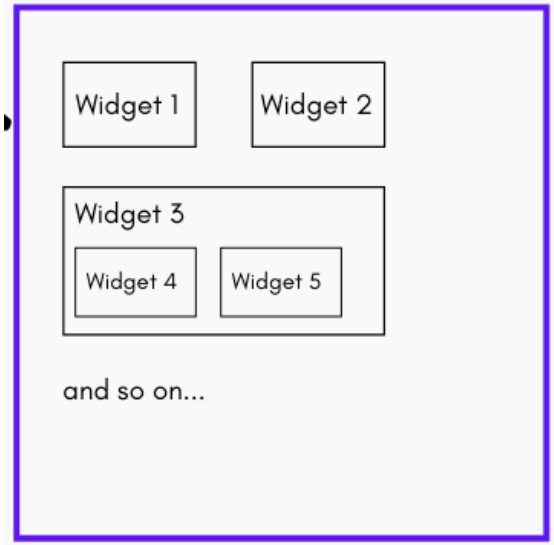
9. **Listbox:** This is used to create a List widget used for displaying a list of values.

10. **Scrollbar:** This widget is used for scrolling content in other widgets like Text and Listbox.

Tkinter Windows

The term "Window" has different meanings in the different contexts, But generally "Window" refers to a rectangular area somewhere on the user's display screen through which you can interact.

- The top-level window object in GUI Programming contains all of the little window objects that will be part of your complete GUI.
- The little window objects can be text labels, buttons, list boxes, etc., and these individual little GUI components are known as Widgets.
- The object that is returned by making a call to tkinter.Tk() is usually referred to as the Root Window.
- The widgets can either be stand-alone or can be containers. If one widget contains other widgets, it is considered the parent of those widgets.
- Similarly, if a widget is contained within another widget, it's known as a child of the parent, the parent is the next immediate enclosing container widget.




```
win = tkinter.Tk()
```

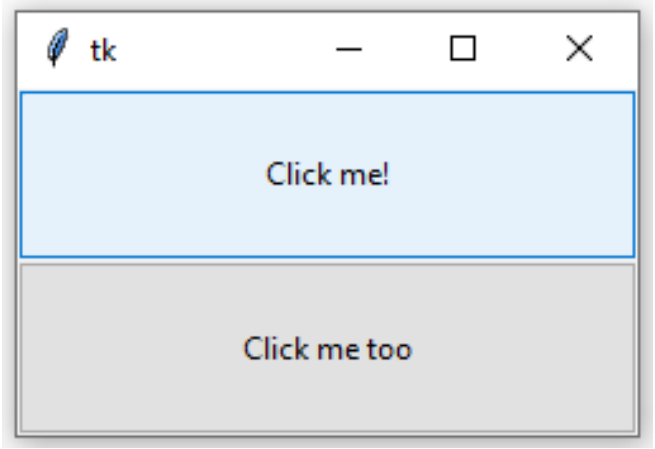
1. **pack() method in Tkinter**

The Pack geometry manager packs widgets relative to the earlier widget.

Tkinter literally packs all the widgets one after the other in a window. We can use options like fill, expand, and side to control this geometry manager.

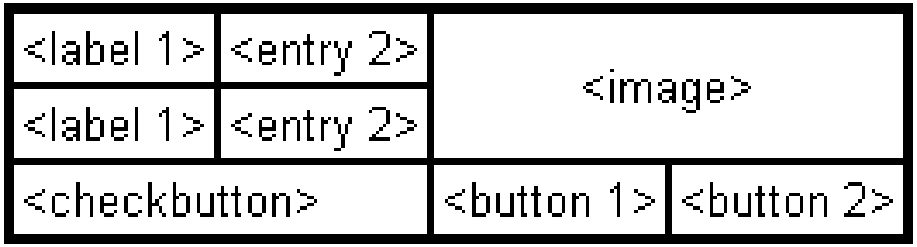
Example;

```
# Importing tkinter module
from tkinter import * from tkinter.ttk import *
# creating Tk window
master = Tk()
# creating a Fra, e which can expand according
# to the size of the window
pane = Frame(master)
pane.pack(fill = BOTH, expand = True)
# button widgets which can also expand and fill
# in the parent widget entirely
# Button 1
b1 = Button(pane, text = "Click me !")
b1.pack(fill = BOTH, expand = True)
# Button 2
b2 = Button(pane, text = "Click me too")
b2.pack(fill = BOTH, expand = True)
# Execute Tkinter
master.mainloop()
```



2. **grid() method in Tkinter**

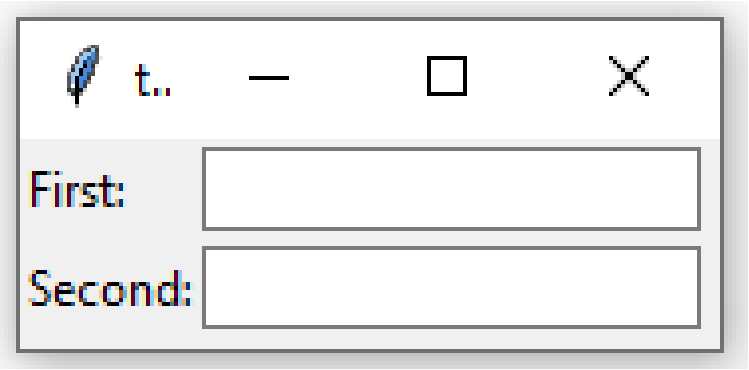
- The Grid geometry manager puts the widgets in a 2-dimensional table.
- The master widget is split into a number of rows and columns, and each “cell” in the resulting table can hold a widget.
- The grid manager is the most flexible of the geometry managers in Tkinter.
- If you don’t want to learn how and when to use all three managers, you should at least make sure to learn this one.



- Creating this layout using the **pack** manager is possible, but it takes a number of extra frame widgets, and a lot of work to make things look good.
- If you use the grid manager instead, you only need one call per widget to get everything laid out properly. Using the **grid** manager is easy.
- Just create the widgets, and use the **grid** method to tell the manager in which row and column to place them. You don’t have to specify the size of the grid beforehand; the manager automatically determines that from the widgets in it.

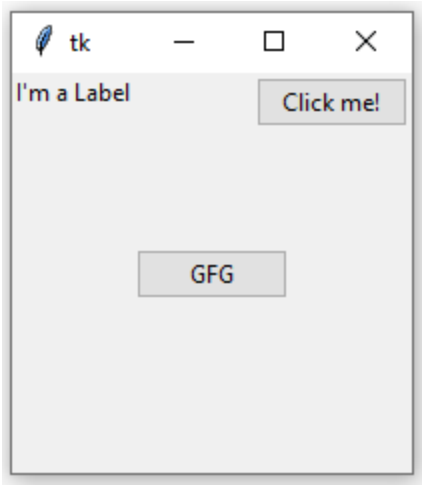
Example:

```
# import tkinter module
from tkinter import * from tkinter.ttk import *

# creating main tkinter window/toplevel
master = Tk()

# this will create a label widget
l1 = Label(master, text = "First:")
l2 = Label(master, text = "Second:")

# grid method to arrange labels in respective
# rows and columns as specified
l1.grid(row = 0, column = 0, sticky = W, pady = 2)
l2.grid(row = 1, column = 0, sticky = W, pady = 2)

# entry widgets, used to take entry from user
e1 = Entry(master)
e2 = Entry(master)

#  this will arrange entry widgets
e1.grid(row = 0, column = 1, pady = 2)
e2.grid(row = 1, column = 1, pady = 2)

# infinite loop which can be terminated by
      keyboard
# or mouse interrupt
mainloop()
```



3. place() method in Tkinter

- The Place geometry manager is the simplest of the three general geometry managers provided in Tkinter.
- It allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window.
- **Syntax:**
- widget.place(relx = 0.5, rely = 0.5, anchor = CENTER)



- Example:

```
# Importing tkinter module
from tkinter import * from tkinter.ttk import *

# creating Tk window
master = Tk()

# setting geometry of tk window
master.geometry("200x200")

# button widget
b1 = Button(master, text = "Click me !")
b1.place(relx = 1, x =-2, y = 2, anchor = NE)

# label widget
l = Label(master, text = "I'm a Label")
l.place(anchor = NW)

# button widget
b2 = Button(master, text = "GFG")
b2.place(relx = 0.5, rely = 0.5, anchor = CENTER)

# infinite loop which is required to
# run tkinter program infinitely
# until an interrupt occurs
mainloop()
```

SQLite3

- Light-weight implementation of a relational DBMS (~340Kb)
  - Includes most of the features of full DBMS
  - Intended to be imbedded in programs
- Used as the data manager in iPhone apps and Firefox (among many others)
- Databases are stored as files in the OS

SQLite3 Data types

- SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container
- Types are:
  - **NULL**: The value is a NULL value.
  - **INTEGER**: The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value
  - **REAL**: The value is a floating point value, stored as an 8-byte IEEE floating point number.
  - **TEXT**. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE). (default max 1,000,000,000 chars)
  - **BLOB**. The value is a blob of data, stored exactly as it was input.

SQLite3 from Python

- SQLite is available as a loadable python library
  - You can use any SQL commands to create, add data, search, update and delete.

Python SQLite3 module

•**Python SQLite3** module is used to integrate the SQLite database with Python.

•It is a standardized Python DBI API 2.0 and provides a straightforward and simple-to-use interface for interacting with SQLite databases.

•There is no need to install this module separately as it comes along with Python after the 2.5x version.

•To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.

SQLite Methods

1. **Connect**

When we connect to a SQLite database, we are accessing data that ultimately resides in a file on our computer. SQLite databases are fully featured SQL engines that can be used for many purposes.

Syntax:

sqlite3.connect(database [,timeout ,other optional arguments])

Example:

```
import sqlite3
connection = sqlite3.connect("aquarium.db")
print(connection.total_changes)
```

Explain:

import sqlite3 gives our Python program access to the sqlite3 module. The sqlite3.connect() function returns a Connection object that we will use to interact with the SQLite database held in the file aquarium.db. The aquarium.db file is created automatically by sqlite3.connect() if aquarium.db does not already exist on our computer.

2. Cursor

- Cursors are created by the connection.cursor() method: they are bound to the connection for the entire lifetime and all the commands are executed in the context of the database session wrapped by the connection.
- Cursors created from different connections can or can not be isolated, depending on the connections' isolation level.
- **Syntax:**
- connection.cursor([cursorClass])

Example:

```
cursor = connection.cursor()
    cursor.execute("CREATE
    TABLE fish (name TEXT,
    species TEXT, tank_number
    INTEGER)")
```

Explain:

connection.cursor() returns a Cursor object. Cursor objects allow us to send SQL statements to a SQLite database using cursor.execute().

3. Execute

A cursor is an object which helps to execute the query and fetch the records from the database. The cursor plays a very important role in executing the query.

Syntax:

```
cursor.execute(operation, params=None, multi=False)
```

Example:

```
rows = cursor.execute("SELECT name, species, tank_number FROM fish").fetchall()
print(rows)
```

Output:

```
[('Sammy', 'shark', 1), ('Jamie', 'cuttlefish', 7)]
```

4. Close

This method closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!

Syntax:

connection.close()

Example:

```
import sqlite3
connection = sqlite3.connect("aquarium.db")
cursor = connection.cursor()
cursor.close()
```

Connect To Database

Following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

Code:

```
#!/usr/bin/python
import sqlite3
conn = sqlite3.connect('test.db')
print "Opened database successfully";
```

explain :

- let's run the above program to create our database test.db in the current directory.
- You can change your path as per your requirement. Keep the above code in sqlite.py file and execute it as shown below.
- If the database is successfully created, then it will display the following message.

output:

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

Create a Table

Following Python program will be used to create a table in the previously created database.

Code:

```
#!/usr/bin/python
import sqlite3
conn = sqlite3.connect('test.db')
print "Opened database successfully";
conn.execute('''CREATE TABLE COMPANY
   (ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL);''')
print "Table created successfully";
conn.close()
```

explain:

When the above program is executed, it will create the COMPANY table in your test.db and it will display the following messages −

output:

```
Opened database successfully
Table created successfully
```

Operations on Tables

1. INSERT Operation

Following Python program shows how to create records in the COMPANY table created in the above example.

Code:

```
#!/usr/bin/python
import sqlite3
conn = sqlite3.connect('test.db')
print "Opened database successfully";
conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (1, 'Paul', 32, 'California', 20000.00 )");
conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");
conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");
conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");
conn.commit()
print "Records created successfully";
conn.close()
```

Output:

```
Opened database successfully
Records created successfully
```

2. SELECT Operation

Following Python program shows how to fetch and display records from the COMPANY table created in the above example.

```
#!/usr/bin/python
import sqlite3
conn = sqlite3.connect('test.db')
print "Opened database successfully";
cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

Output:

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

3. UPDATE Operation

Following Python code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```
#!/usr/bin/python

import sqlite3
conn = sqlite3.connect('test.db')
print "Opened database successfully";
conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID = 1")
conn.commit()
print "Total number of rows updated :", conn.total_changes
cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

OUTPUT:

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 25000.0
ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0
ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0
ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0
Operation done successfully
```

4. DELETE Operation

Following Python code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
#!/usr/bin/python

import sqlite3
conn = sqlite3.connect('test.db')
print "Opened database successfully";
conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print "Total number of rows deleted :", conn.total_changes
cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

Output:
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully

Unit 5

Python Data Visualization

Visualization of Data

The field of data analysis that focuses on visualizing data is called data visualization. It is an effective method of communicating data insights and presents data graphically.

Python Data Visualization

Python provides various libraries that come with different features for visualizing data. All these libraries come with different features and can support various types of graphs. In this tutorial, we will be discussing four such libraries.

- Matplotlib
- Seaborn
- Plotly

Matplotlib

Matplotlib is an easy-to-use, low-level data visualization library that is built on NumPy arrays. It consists of various plots like scatter plot, line plot, pie chart, bar chart, histogram, etc. Matplotlib provides a lot of flexibility.

Installation of Matplotlib

If you have Python and PIP already installed on a system, then installation of Matplotlib is very easy.

Install it using this command:

pip install matplotlib

Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding the import module statement:

import matplotlib

plot() method:

the plot() function is used to draw points (markers) in a diagram.

By default, the plot() function draws a line from point to point.

The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the **x-axis**.

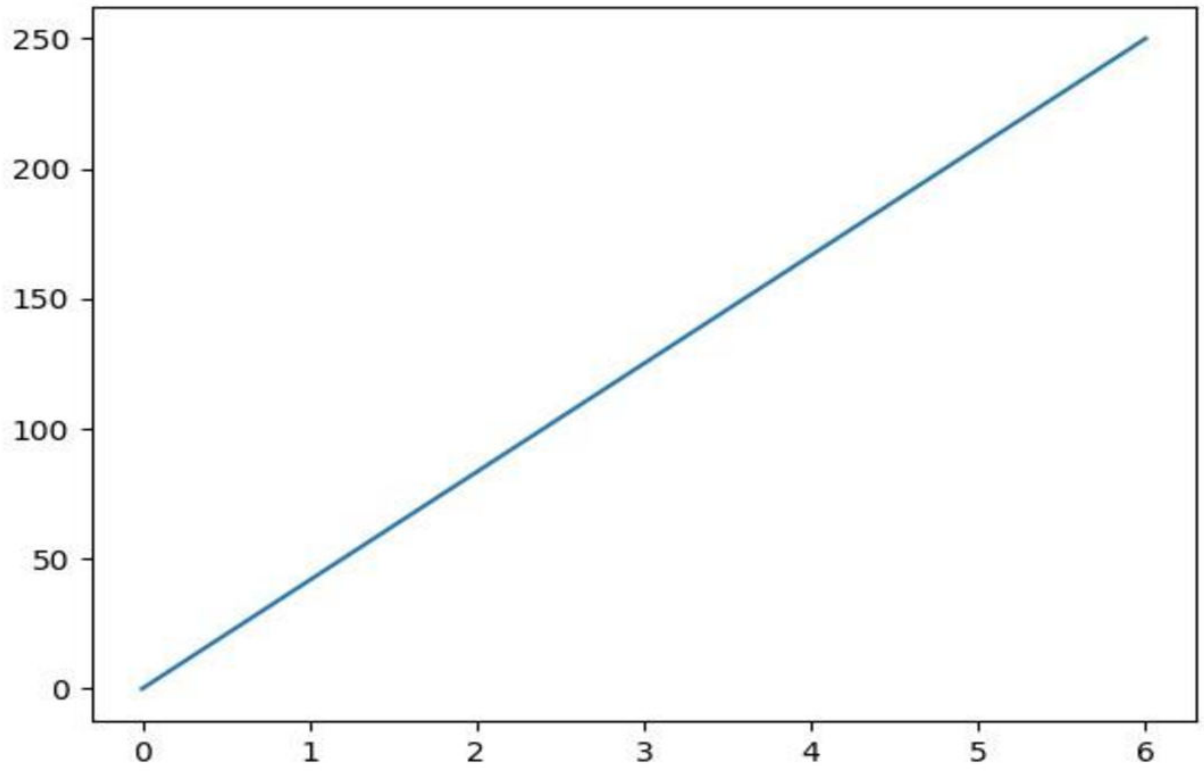
Parameter 2 is an array containing the points on the **y-axis**.

Example:

```
#Draw a line in a diagram from position (0,0) to position (6,250):

import matplotlib.pyplot as plt
import numpy as np
```

```
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```

output:



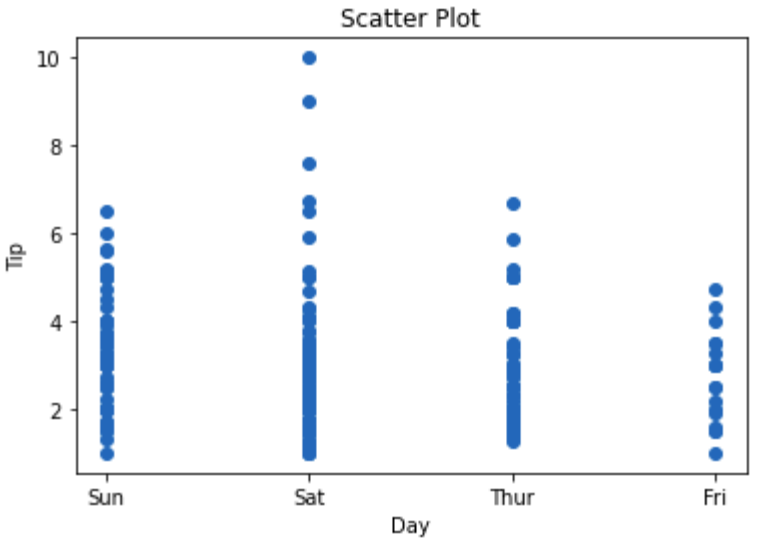
Scatter Plot

- Scatter plots are used to observe relationships between variables and uses dots to represent the relationship between them.
- The scatter() method in the matplotlib library is used to draw a scatter plot.

Exapmle:

```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# Scatter plot with day against tip
plt.scatter(data['day'], data['tip'])
# Adding Title to the Plot
plt.title("Scatter Plot")

# Setting the X and Y labels
plt.xlabel('Day')
plt.ylabel('Tip')
plt.show()
```

Output:



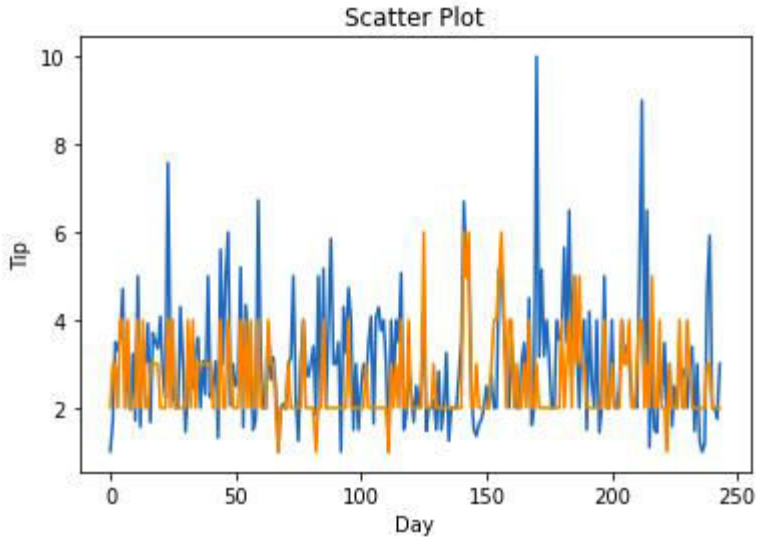
Line Chart

Line Chart is used to represent a relationship between two data X and Y on a different axis. It is plotted using the plot() function. Let's see the below example.

Example:

```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# Scatter plot with day against tip
plt.plot(data['tip'])
plt.plot(data['size'])
# Adding Title to the Plot
plt.title("Scatter Plot")

# Setting the X and Y labels
plt.xlabel('Day')
plt.ylabel('Tip')
plt.show()
```

Output:



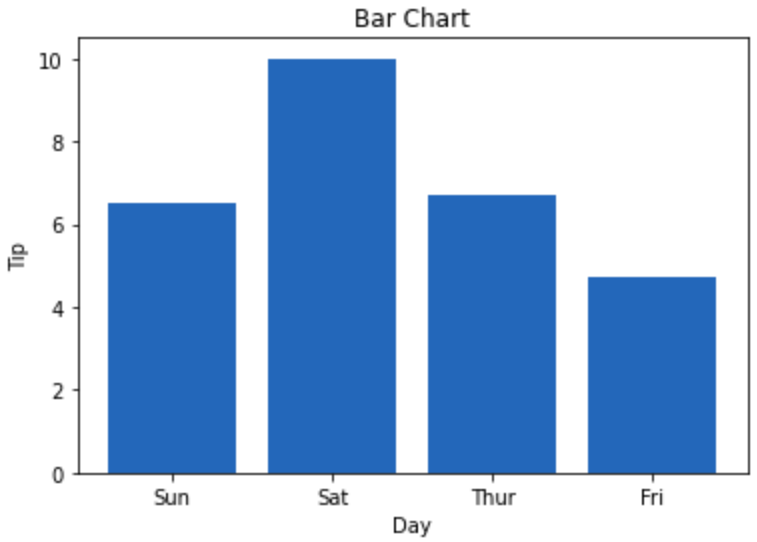
Bar Chart

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. It can be created using the bar() method.

Example:

```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# Bar chart with day against tip
plt.bar(data['day'], data['tip'])
plt.title("Bar Chart")

# Setting the X and Y labels
plt.xlabel('Day')
plt.ylabel('Tip')

# Adding the legends
plt.show()
```

Output:



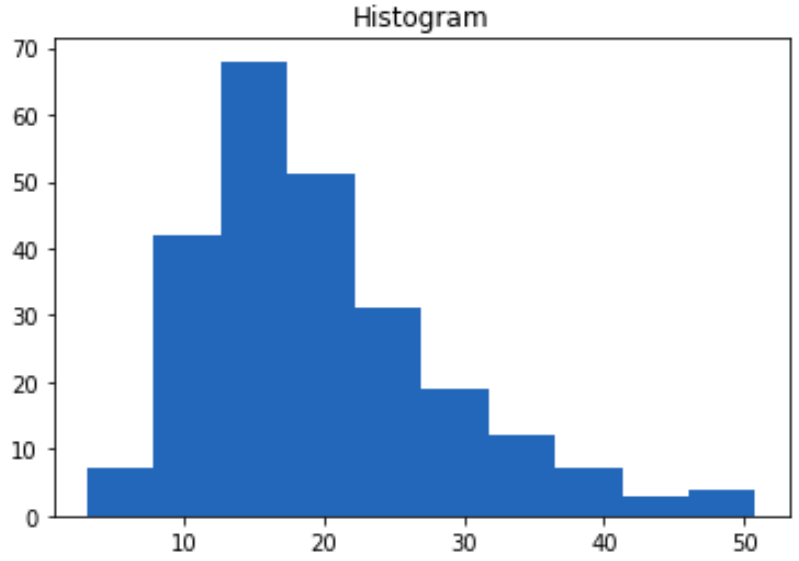
Histogram

- A histogram is basically used to represent data in the form of some groups.
- It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency.
- The hist() function is used to compute and create a histogram. In histogram, if we pass categorical data then it will automatically compute the frequency of that data i.e. how often each value occurred.

Example:

```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# histogram of total_bills
plt.hist(data['total_bill'])

plt.title("Histogram")

# Adding the legends
plt.show()
```

Output:



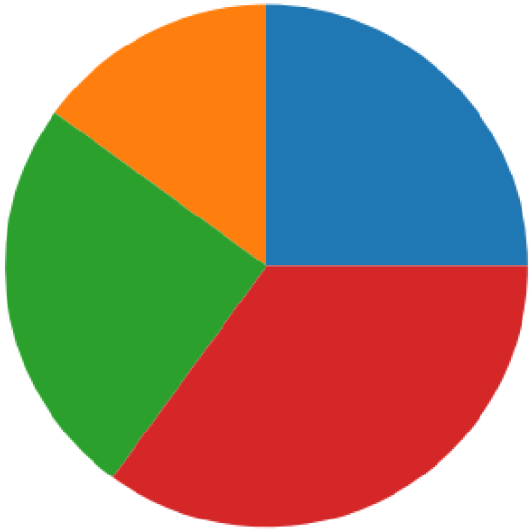
Matplotlib Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

Example

#A simple pie chart:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([25, 15, 25, 35])

plt.pie(y)
plt.show()
```



Labels

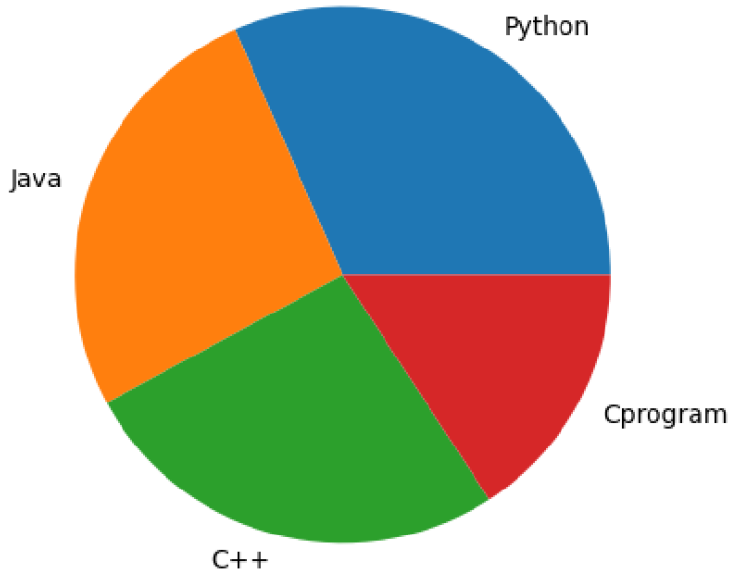
Add labels to the pie chart with the `label` parameter.

The `label` parameter must be an array with one label for each wedge

Example

#A simple pie chart:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([30, 25, 25, 15])
mylabels = ["Python", "Java", "C++", "Cprogram"]

plt.pie(y, labels = mylabels)
plt.show()
```



Colors

You can set the color of each wedge with the colors parameter.

The colors parameter, if specified, must be an array with one value for each wedge:

Example:

```
import matplotlib.pyplot as plt

import numpy as np

y = np.array([30, 25, 25, 15])

mylabels = ["Python", "Java", "C++", "Cprogram"]

mycolours=["hotpink","yellow","red","green"]

plt.pie(y, labels = mylabels,colors=mycolours)

plt.show()
```