Chapter 1 –
Reviews on Python programming

1. Introduction

To install Python, download the installer from python.org for your platform, run the executable, and follow the on-screen instructions to select your desired options, such as "Add Python to PATH" for easier command-line use and specifying an installation directory. After installation, verify by opening a new terminal and typing python --version.

○ Get the Installer

Go to the official source: Visit python.org in your web browser and navigate to the "Downloads" section.
Select your OS: The website will typically detect your operating system or provide a button to download the appropriate installer for Windows, macOS, or Linux.
Download the installer: Click the button to download the latest version of Python.

○ Run the Installer

Launch the installer: Once the download is complete, double-click the downloaded file to start the installation process.
Customize the installation: "Add Python to PATH": This is a crucial step, especially on Windows, as it allows you to run Python directly from the command line.
Customize installation location: You can choose to install Python in a custom folder, such as under a _tools or Python directory.
Optional components: Ensure that components like pip (the package installer), the IDLE development environment, and the test suite are selected.

○ Verify the Installation

Open a new terminal: Open PowerShell or Command Prompt on Windows, or the Terminal on macOS/Linux.
Check the version: Type python --version and press Enter.
Confirm the output: If the installation was successful, the command will display the installed Python version.

○ Important Notes

For Windows: Using the official python.org installer is generally recommended over the Microsoft Store version, which can be more restrictive for advanced use.
Virtual Environments: It is recommended to create and use virtual environments for projects to manage dependencies separately and avoid conflicts between different projects.

2. Operating systems

An **operating system (OS)** is system software that manages computer hardware and software resources, and provides common services for computer programs.

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, peripherals, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware,[1][2] although the application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers.

As of September 2024, Android is the most popular operating system with a 46% market share, followed by Microsoft Windows at 26%, iOS and iPadOS at 18%, macOS at 5%, and Linux at 1%. Android, iOS, and iPadOS are mobile operating systems, while Windows, macOS, and Linux are desktop operating systems.[3] Linux distributions are dominant in the server and supercomputing sectors. Other specialized classes of operating systems (special-purpose operating systems),[4][5] such as embedded and real-time systems, exist for many applications. Security-focused operating systems also exist. Some operating systems have low system requirements (e.g. light-weight Linux distribution). Others may have higher system requirements.

Some operating systems require installation or may come pre-installed with purchased computers (OEM-installation), whereas others may run directly from media (i.e. live CD) or flash memory (i.e. a LiveUSB from a USB stick).

○ Definition and purpose

An operating system is difficult to define, but has been called "the layer of software that manages a computer's resources for its users and their applications". Operating systems include the software that is always running, called a kernel—but can include other software as well. The two other types of programs that can run on a computer are system programs—which are associated with the operating system, but may not be part of the kernel—and applications—all other software.

There are three main purposes that an operating system fulfills:

- Operating systems allocate resources between different applications, deciding when they will receive central processing unit (CPU) time or space in memory. On modern personal computers, users often want to run several applications at once. In order to ensure that one program cannot monopolize the computer's limited hardware resources, the operating system gives each application a share of the resource, either in time (CPU) or space (memory). The operating system also must isolate applications from each other to protect them from errors and security vulnerabilities in another application's code, but enable communications between different applications.

- Operating systems provide an interface that abstracts the details of accessing hardware details (such as physical memory) to make things easier for programmers. Virtualization also enables the operating system to mask limited hardware resources; for example, virtual memory can provide a program with the illusion of nearly unlimited memory that exceeds the computer's actual memory.
- Operating systems provide common services, such as an interface for accessing network and disk devices. This enables an application to be run on different hardware without needing to be rewritten. Which services to include in an operating system varies greatly, and this functionality makes up the great majority of code for most operating systems.

○ Types of operating systems

Multicomputer operating systems- With multiprocessors multiple CPUs share memory. A multicomputer or cluster computer has multiple CPUs, each of which has its own memory. Multicomputers were developed because large multiprocessors are difficult to engineer and prohibitively expensive; they are universal in cloud computing because of the size of the machine needed.
The different CPUs often need to send and receive messages to each other; to ensure good performance, the operating systems for these machines need to minimize this copying of packets. Newer systems are often multiqueue—separating groups of users into separate queues—to reduce the need for packet copying and support more concurrent users.
Another technique is remote direct memory access, which enables each CPU to access memory belonging to other CPUs. Multicomputer operating systems often support remote procedure calls where a CPU can call a procedure on another CPU, or distributed shared memory, in which the operating system uses virtualization to generate shared memory that does not physically exist.

Distributed systems - A distributed system is a group of distinct, networked computers—each of which might have their own operating system and file system. Unlike multicomputers, they may be dispersed anywhere in the world.[24] Middleware, an additional software layer between the operating system and applications, is often used to improve consistency. Although it functions similarly to an operating system, it is not a true operating system.

Embedded - Embedded operating systems are designed to be used in embedded computer systems, whether they are internet of things objects or not connected to a network. Embedded systems include many household appliances. The distinguishing factor is that they do not load user-installed software. Consequently, they do not need protection between different applications, enabling simpler designs. Very small operating systems might run in less than 10 kilobytes, and the smallest are for smart cards. Examples include Embedded Linux, QNX, VxWorks, and the extra-small systems RIOT and TinyOS.

Real-time - A real-time operating system is an operating system that guarantees to process events or data by or at a specific moment in time. Hard real-time systems require exact timing and are common in manufacturing, avionics, military, and other similar uses. With soft real-time systems, the occasional missed event is acceptable; this category often includes audio or multimedia systems, as well as smartphones. In order for hard real-time systems be sufficiently exact in their timing, often they are just a library with no protection between applications, such as eCos.

Hypervisor - A hypervisor is an operating system that runs a virtual machine. The virtual machine is unaware that it is an application and operates as if it had its own hardware. Virtual machines can be paused, saved, and resumed, making them useful for operating systems research, development, and debugging. They also enhance portability by enabling applications to be run on a computer even if they are not compatible with the base operating system.

Library- A *library operating system* (libOS) is one in which the services that a typical operating system provides, such as networking, are provided in the form of libraries and composed with a single application and configuration code to construct a unikernel: a specialized (only the absolute necessary pieces of code are extracted from libraries and bound together), single address space, machine image that can be deployed to cloud or embedded environments.

The operating system code and application code are not executed in separated protection domains (there is only a single application running, at least conceptually, so there is no need to prevent interference between applications) and OS services are accessed via simple library calls (potentially inlining them based on compiler thresholds), without the usual overhead of context switches, in a way similarly to embedded and real-time OSes.

Note that this overhead is not negligible: to the direct cost of mode switching it's necessary to add the indirect pollution of important processor structures (like CPU caches, the instruction pipeline, and so on) which affects both user-mode and kernel-mode performance.

- **Role**

An operating system's role is to act as an intermediary between the user and the computer's hardware, managing all hardware and software resources to provide a convenient and efficient platform for executing programs.
It serves as the central hub for a computer's operation, coordinating everything from memory and processor allocation to file and device management, allowing users to interact with the computer and run applications without needing to understand its complex internal workings.

Key Roles and Functions

- *Hardware/Software Interface*: The OS provides a crucial interface, translating user or application commands into instructions that the computer hardware can understand and execute.

- *Resource Management*: It manages and allocates essential computer resources, including the CPU (processor time), memory, input/output devices, and storage.

- *Process Management*: The OS controls the execution of all programs and tasks, scheduling them and ensuring they receive the resources they need to run efficiently.

- *File Management*: It handles operations like creating, reading, writing, and deleting files, organizing data in a structured way.

- *Security*: The OS implements security measures to protect the system and its data from unauthorized access.

- *User Interface*: It provides the user interface (like a desktop or command line) that allows users to interact with the computer.

- *System Stability*: By coordinating all components, the OS ensures the stable and smooth operation of the entire computer system. A detailed comparison between Linux and windows is illustrated in the below table.

Table 1. Comparison between Linux and Windows

S. No	Linux	Windows
1.	Linux is an **open-source** operating system.	Windows is **not** an open-source operating system.
2.	Linux is **free of cost**.	Windows is **paid** and requires a license.
3.	**File names are case-sensitive**, meaning file.txt and File.txt are different.	**File names are case-insensitive**, meaning file.txt and File.txt are treated the same.
4.	Uses a **monolithic kernel**.	Uses a **hybrid kernel**.
5.	**More efficient and stable**, especially for servers and developers.	**Less efficient** due to resource-intensive processes.
6.	Uses **forward slash (/)** for directory separation.	Uses **backslash (\)** for directory separation.
7.	**More secure** with better user control and fewer vulnerabilities.	**Less secure** due to higher susceptibility to malware and viruses.
8.	Preferred by **hackers and security experts** due to its open-source nature and control.	**Not widely used for hacking** as it lacks built-in security tools.
9.	Has **3 types of user accounts**: (1) Regular, (2) Root, (3) Service Account.	Has **4 types of user accounts**: (1) Administrator, (2) Standard, (3) Child, (4) Guest.
10.	**Root user** has all administrative privileges.	**Administrator user** has all administrative privileges.
11.	In Linux, you **can have two files with the same name** but different cases (File.txt and file.txt).	In Windows, **you cannot have two files with the same name** in the same folder.

3. Basic programming modes.

What is the Difference between Interactive and Script Mode in Python Programming?

Python is a programming language that lets you work quickly and integrate systems more efficiently. It is a widely-used general-purpose, high-level programming language. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code. In the Python programming language, there are two ways in which we can run our code: Interactive and Script mode

○ Interactive mode

Interactive etymologically means "working simultaneously and creating impact of our work on the other's work". Interactive mode is based on this ideology only. In the interactive mode as we enter a command and press enter, the very next step we get the output. The output of the code in the interactive mode is influenced by the last command we give. Interactive mode is very convenient for writing very short lines of code. In python it is also known as *REPL* which stands for *Read Evaluate Print Loop.*

Here, the read function reads the input from the user and stores it in memory. Eval function evaluates the input to get the desired output. Print function outputs the evaluated result. The loop function executes the loop during the execution of the entire program and terminates when our program ends. This mode is very suitable for beginners in programming as it helps them evaluate their code line by line and understand the execution of code well.

○ How to run python code in Interactive mode?

In order to run our program in the interactive mode, we can use command prompt in windows, terminal in Linux, and macOS.

Let us see understand the execution of python code in the command prompt with the help of an example:

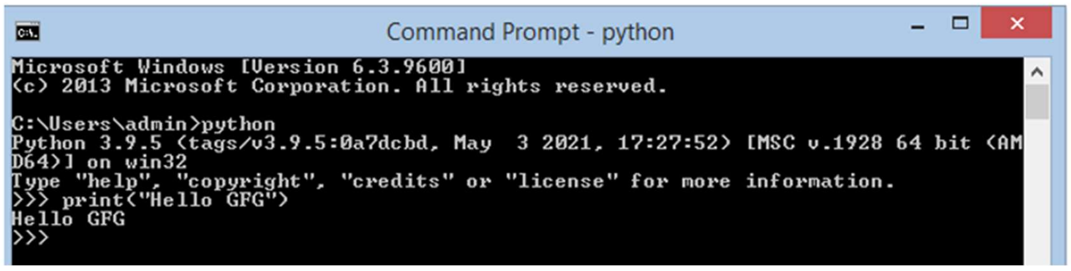
Example 1

To run python in command prompt type "python". Then simply type the Python statement on >>> prompt. As we type and press enter we can see the output in the very next line.

# Python program to display "Hello GFG"

print("Hello GFG")

Output



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>python
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello GFG")
Hello GFG
>>>
```

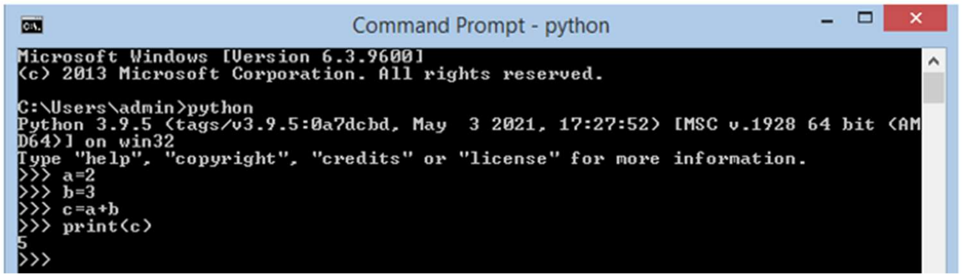
Example 2

Let us take another example in which we need to perform addition on two numbers and we want to get its output. We will declare two variables a and b and store the result in a third variable c. We further print c. All this is done in the command prompt.

# *Python program to add two numbers*
a = 2
b = 3
c = a + b # *Adding a and b and storing result in c*

# *Printing value of c*
print(c)

Output:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>python
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a=2
>>> b=3
>>> c=a+b
>>> print(c)
5
>>>
```

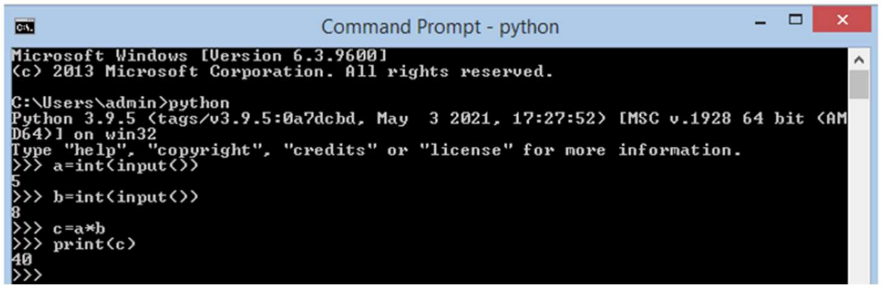
We can see the desired output on the screen. This kind of program is a very short program and can be easily executed in interactive mode.

Example 3

In this example, we will multiply two numbers and take the numbers as an input for two users. You will see that when you execute the input command, you need to give input in the very next line, i.e. code is interpreted line by line.

# *Python program to take input from user*
a = int(input()) # *Taking input from user*
b = int(input()) # *Taking input from user*
c = a * b # *Multiplying and storing result*
print(c) # *Printing the result*

Output



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>python
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a=int(input())
5
>>> b=int(input())
8
>>> c=a*b
>>> print(c)
40
>>>
```

- Disadvantages of interactive mode

- The interactive mode is not suitable for large programs.
- The interactive mode doesn't save the statements. Once we make a program it is for that time itself, we cannot use it in the future. In order to use it in the future, we need to retype all the statements.
- Editing the code written in interactive mode is a tedious task. We need to revisit all our previous commands and if still, we could not edit we need to type everything again.

- **Script Mode**

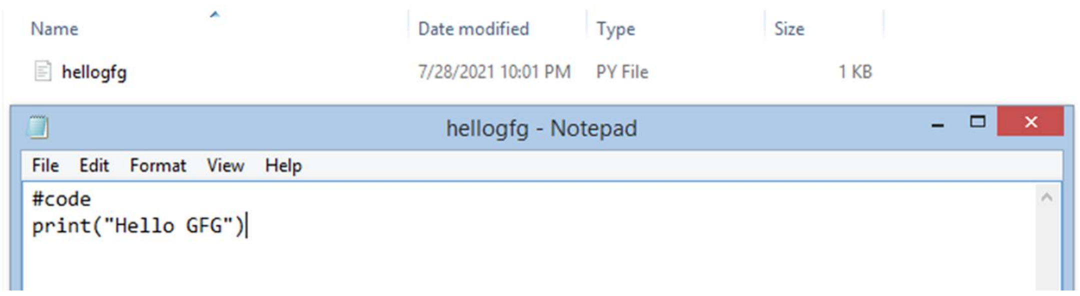
Script etymologically means a system of writing. In the script mode, a python program can be written in a file. This file can then be saved and executed using the command prompt. We can view the code at any time by opening the file and editing becomes quite easy as we can open and view the entire code as many times as we want. Script mode is very suitable for writing long pieces of code. It is much preferred over interactive mode by experts in the program. The file made in the script mode is by default saved in the Python installation folder and the extension to save a python file is ".py".

- How to run python code in script mode?

In order to run a code in script mode follow the following steps.

Step 1: Make a file using a text editor. You can use any text editor of your choice(Here I use notepad).
Step 2: After writing the code save the file using ".py" extension.
Step 3: Now open the command prompt and command directory to the one where your file is stored.
Step 4: Type python "filename.py" and press enter.
Step 5: You will see the output on your command prompt.
Let us understand these steps with the help of the examples:

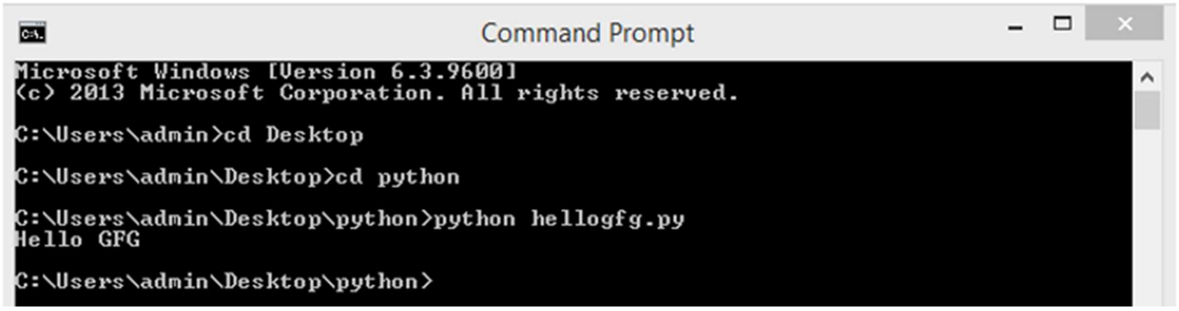
Example 4
In order to execute "Hello gfg" using script mode we first make a file and save it.



```
#code
print("Hello GFG")
```

Now we use the command prompt to execute this file.

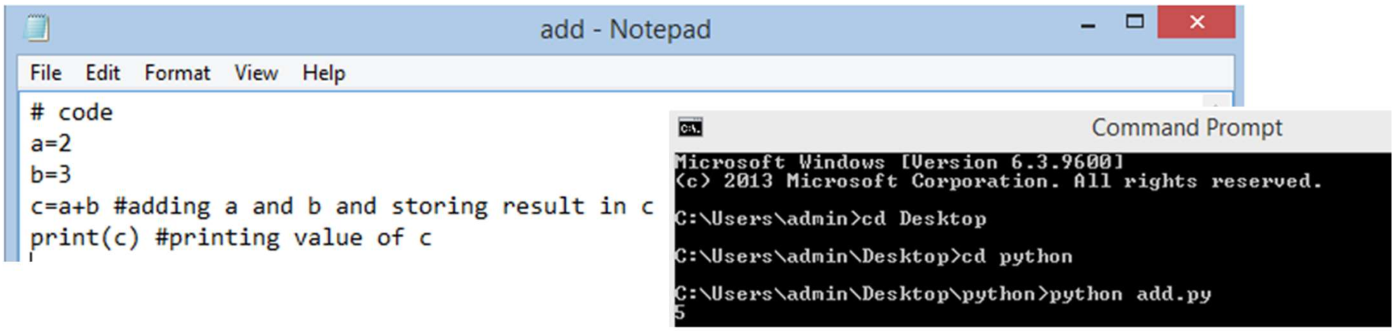
Output:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd Desktop

C:\Users\admin\Desktop>cd python

C:\Users\admin\Desktop\python>python hellogfg.py
Hello GFG

C:\Users\admin\Desktop\python>
```

Example 5

Our second example is the same addition of two numbers as we saw in the interactive mode. But in this case, we first make a file and write the entire code in that file. We then save it and execute it using the command prompt.

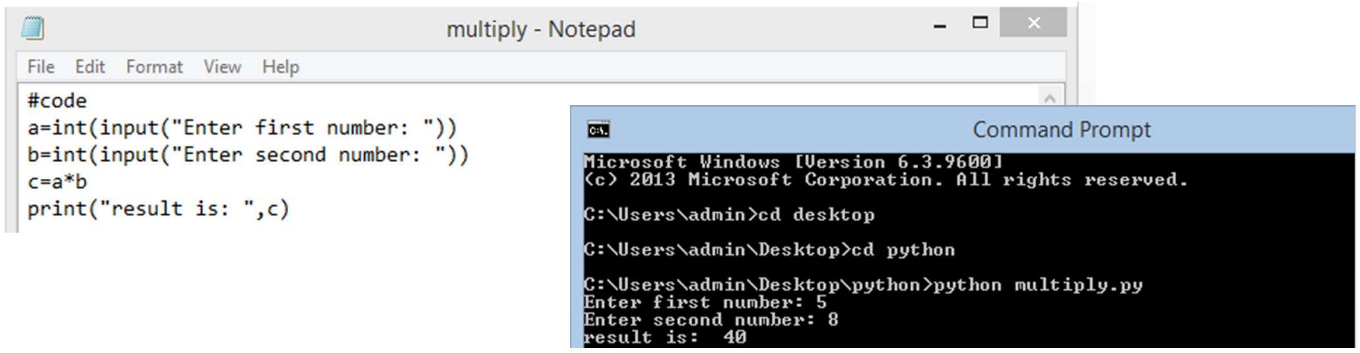


```
# code
a=2
b=3
c=a+b #adding a and b and storing result in c
print(c) #printing value of c
```

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd Desktop

C:\Users\admin\Desktop>cd python

C:\Users\admin\Desktop\python>python add.py
5
```

Example 6

In this example, we write the code for multiplying two numbers. And the numbers which are to be multiplied are taken by the user as an input. In the interactive mode, we saw that as we write the command so does it asks for the input in the very next line. But in script mode we first code the entire program save and then run it in command prompt. The python interpreter executes the code line by line and gives us the result accordingly.



```
#code
a=int(input("Enter first number: "))
b=int(input("Enter second number: "))
c=a*b
print("result is: ",c)
```

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd desktop

C:\Users\admin\Desktop>cd python

C:\Users\admin\Desktop\python>python multiply.py
Enter first number: 5
Enter second number: 8
result is:  40
```

In this example, we see that the whole program is compiled and the code is executed line by line. The output on the shell is entirely different from the interactive mode.

Table 2- Difference between Interactive mode and Script mode.

Interactive Mode	**Script Mode**
It is a way of executing a Python program in which statements are written in command prompt and result is obtained on the same.	In the script mode, the Python program is written in a file. Python interpreter reads the file and then executes it and provides the desired result. The program is compiled in the command prompt,
The interactive mode is more suitable for writing very short programs.	Script mode is more suitable for writing long programs.
Editing of code can be done but it is a tedious task.	Editing of code can be easily done in script mode.
We get output for every single line of code in interactive mode i.e. result is obtained after execution of each line of code.	In script mode entire program is first compiled and then executed.
Code cannot be saved and used in the future.	Code can be saved and can be used in the future.
It is more preferred by beginners.	It is more preferred by experts than the beginners to use script mode.

4. Variables

A ***variable*** in Python is a symbolic name that acts as a reference or a label for an object stored in the computer's memory. It is a container for storing a data value, allowing you to name, access, and manipulate that information throughout your program.

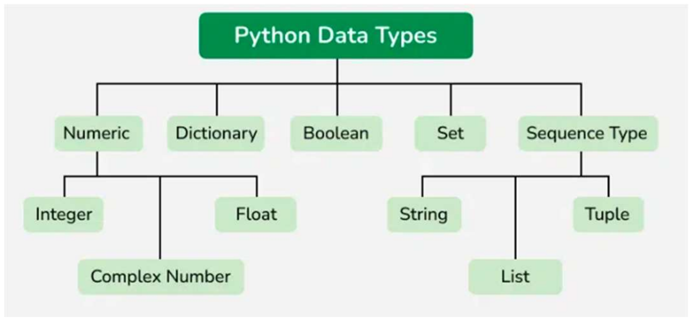
- **Key Concepts**

- ***No Declaration Required***: Unlike many other programming languages (like C++ or Java), you do not need to explicitly declare a variable or its data type beforehand. A variable is created the moment you first assign a value to it using the assignment operator (=).
- ***Dynamic Typing***: Python is a dynamically typed language. This means the type of the value a variable holds is determined at runtime, and the same variable can be reassigned a value of a different type later

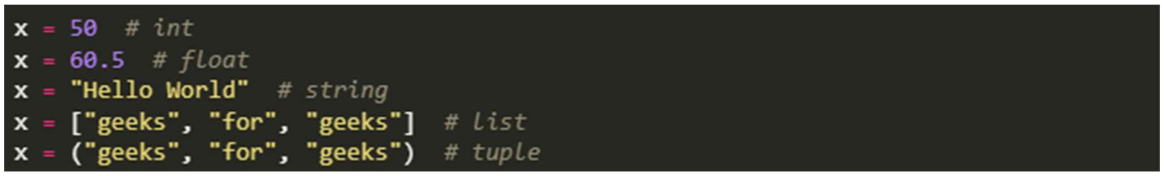
5. Data Types

Data types in Python are a way to classify data items. They represent the kind of value, which determines what operations can be performed on that data. Since everything is an object in Python programming, Python data types are classes and variables are instances (objects) of these classes. The following are standard or built-in data types in Python:

- **Numeric:** int, float, complex
- **Sequence Type:** string, list, tuple
- **Mapping Type:** dict
- **Boolean:** bool
- **Set Type:** set, frozenset
- **Binary Types:** bytes, bytearray, memoryview



As example, the below code assigns variable 'x' different values of few Python data types - int, float, list, tuple and string. Each assignment replaces previous value, making 'x' take on data type and value of most recent assignment.

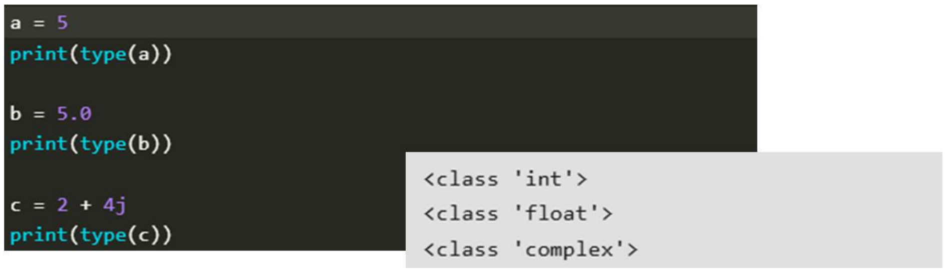


```
x = 50  # int
x = 60.5  # float
x = "Hello World"  # string
x = ["geeks", "for", "geeks"]  # list
x = ("geeks", "for", "geeks")  # tuple
```

○ Numeric Data Types

Python numbers represent data that has a numeric value. A numeric value can be an integer, a floating number or even a complex number. These values are defined as int, float and complex classes.

- ***Integers***: value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). There is no limit to how long an integer value can be.
- ***Float***: value is represented by float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- ***Complex Numbers***: It is represented by a complex class. It is specified as (real part) + (imaginary part) j. For example - 2+3j.



```
a = 5
print(type(a))

b = 5.0
print(type(b))

c = 2 + 4j
print(type(c))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

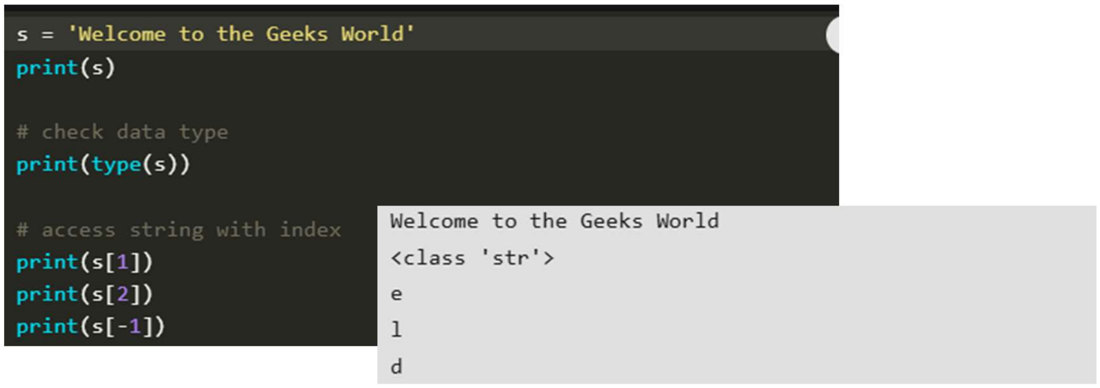
○ Sequence Data Types

A sequence is an ordered collection of items, which can be of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence data types of Python:

○ String (text) Data Type

Python Strings are arrays of bytes representing Unicode characters. In Python, there is no character data type, a character is a string of length one. It is represented by str class.

Strings in Python can be created using single quotes, double quotes or even triple quotes. We can access individual characters of a String using index.



```python
s = 'Welcome to the Geeks World'
print(s)

# check data type
print(type(s))

# access string with index
print(s[1])
print(s[2])
print(s[-1])
```

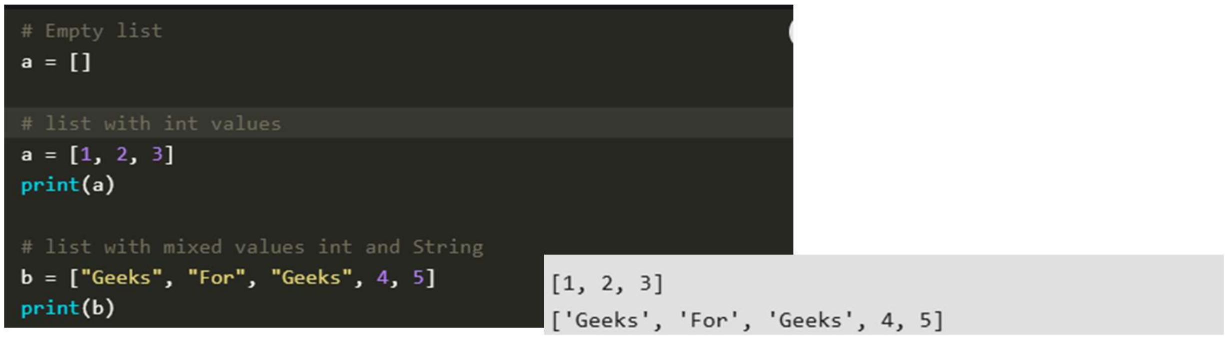
```
Welcome to the Geeks World
<class 'str'>
e
l
d
```

- **List Data Type**

Lists are similar to arrays found in other languages. They are an ordered and mutable collection of items. It is very flexible as items in a list do not need to be of the same type.

  - Creating a List in Python

Lists in Python can be created by just placing sequence inside the square brackets[].

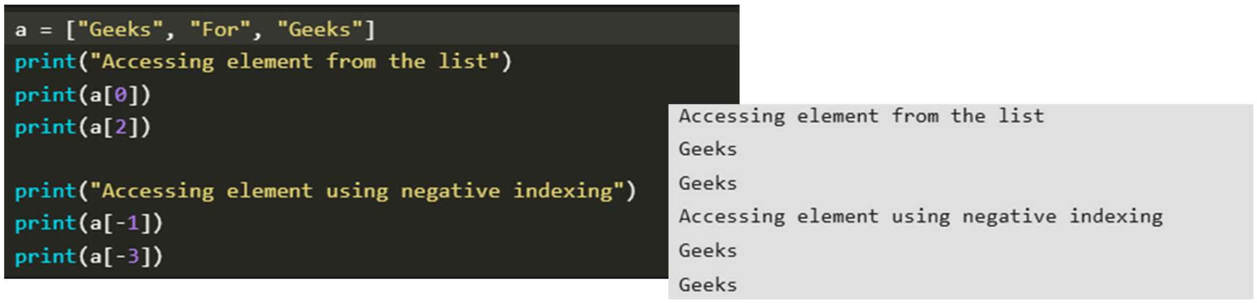


```python
# Empty list
a = []

# list with int values
a = [1, 2, 3]
print(a)

# list with mixed values int and String
b = ["Geeks", "For", "Geeks", 4, 5]
print(b)
```

```
[1, 2, 3]
['Geeks', 'For', 'Geeks', 4, 5]
```

  - **Access List Items**

In order to access the list items refer to index number. In Python, negative sequence indexes represent positions from end of the array. Instead of having to compute offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from end, -1 refers to last item, -2 refers to second-last item, etc.



```python
a = ["Geeks", "For", "Geeks"]
print("Accessing element from the list")
print(a[0])
print(a[2])

print("Accessing element using negative indexing")
print(a[-1])
print(a[-3])
```

```
Accessing element from the list
Geeks
Geeks
Accessing element using negative indexing
Geeks
Geeks
```

- **Tuple Data Type**

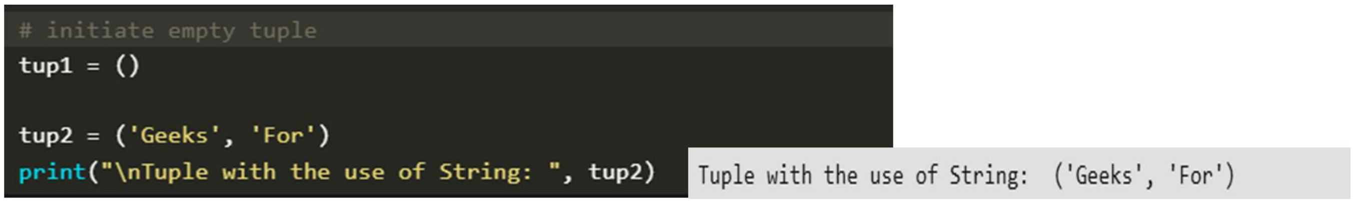
Tuple is an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable. Tuples cannot be modified after it is created.

- **Creating a Tuple in Python**

In Python, tuples are created by placing a sequence of values separated by a 'comma' with or without the use of parentheses for grouping data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.).

Note

*Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing **'comma'** to make it a tuple.*



```
# initiate empty tuple
tup1 = ()

tup2 = ('Geeks', 'For')
print("\nTuple with the use of String: ", tup2)
```

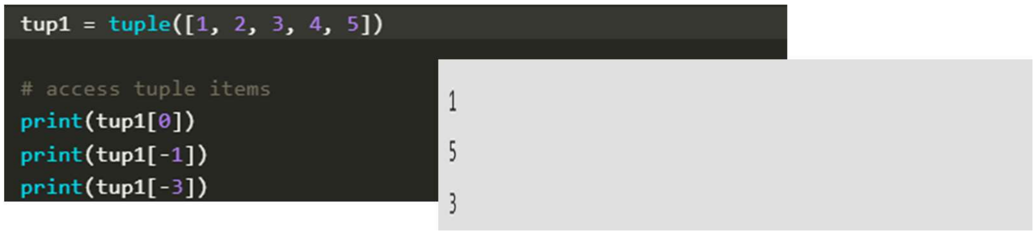
```
Tuple with the use of String:  ('Geeks', 'For')
```

Note

The creation of a Python tuple without the use of parentheses is known as Tuple Packing.

- **Access Tuple Items**

In order to access tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple.



```
tup1 = tuple([1, 2, 3, 4, 5])

# access tuple items
print(tup1[0])
print(tup1[-1])
print(tup1[-3])
```

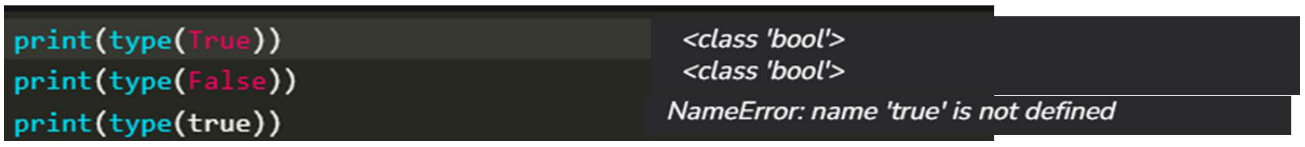
```
1
5
3
```

- **Boolean Data Type in Python**

Python Boolean Data type is one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true) and those equal to False are falsy (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false.

Example

First two lines will print type of the boolean values True and False, which is <class 'bool'>. Third line will cause an error, because true is not a valid keyword. Python is case-sensitive, which means it distinguishes between uppercase and lowercase letters.



```
print(type(True))
print(type(False))
print(type(true))
```

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

- **Set Data Type in Python**

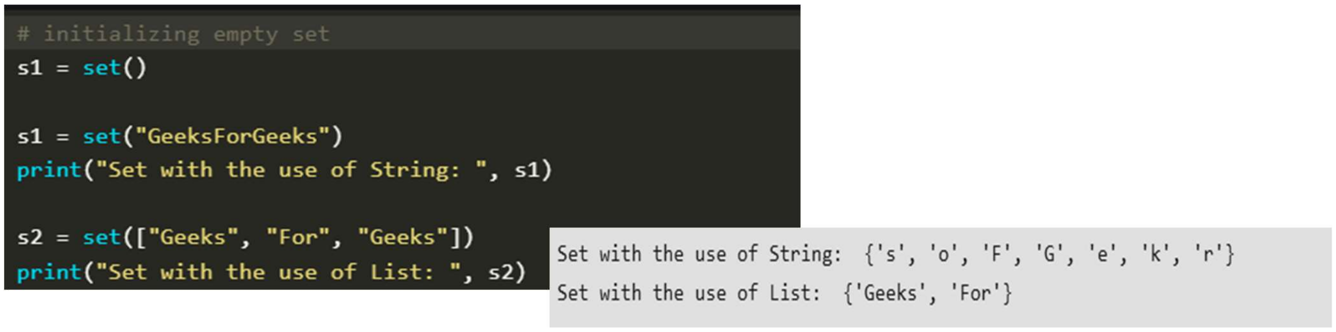
In Python Data Types, Set is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

- **Create a Set in Python**

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

Example

The code is an example of how to create sets using different types of values, such as strings, lists and mixed values

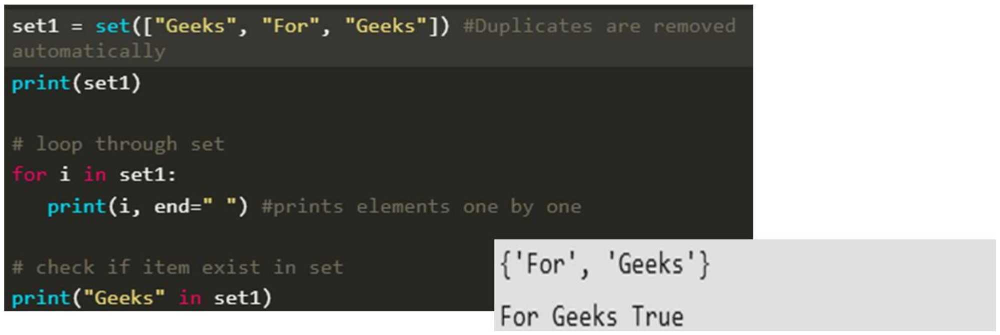


```python
# initializing empty set
s1 = set()

s1 = set("GeeksForGeeks")
print("Set with the use of String: ", s1)

s2 = set(["Geeks", "For", "Geeks"])
print("Set with the use of List: ", s2)
```

```
Set with the use of String:  {'s', 'o', 'F', 'G', 'e', 'k', 'r'}
Set with the use of List:  {'Geeks', 'For'}
```

- **Access Set Items**

Set items cannot be accessed by referring to an index, since sets are unordered the items have no index. But we can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the keyword



```python
set1 = set(["Geeks", "For", "Geeks"]) #Duplicates are removed automatically
print(set1)

# loop through set
for i in set1:
   print(i, end=" ") #prints elements one by one

# check if item exist in set
print("Geeks" in set1)
```

```
{'For', 'Geeks'}
For Geeks True
```

- **Dictionaries**

A dictionary in Python is a collection of data values, used to store data values like a map, unlike other Python Data Types, a Dictionary holds a key: value pair. Key-value is provided in dictionary to make it more optimized.

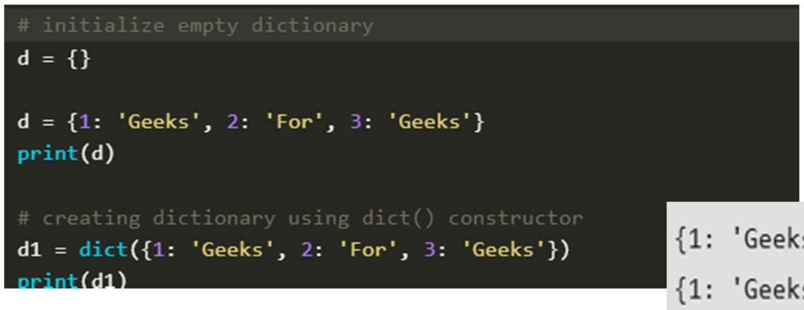
Each key-value pair in a Dictionary is separated by a colon: , whereas each key is separated by a 'comma'.

- **Create a Dictionary in Python**

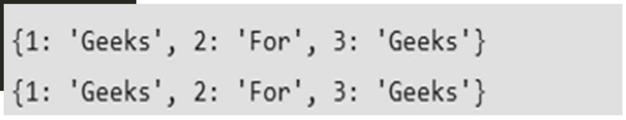
Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. The dictionary can also be created by the built-in function **dict().**

Note

Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.



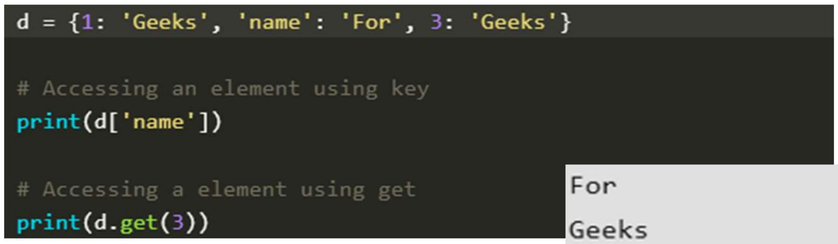
```
# initialize empty dictionary
d = {}

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

# creating dictionary using dict() constructor
d1 = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})
print(d1)
```



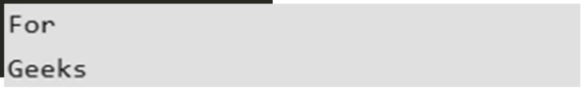
```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

- **Accessing Key-value in Dictionary**

In order to access items of a dictionary refer to its key name. Key can be used inside square brackets. Using get() method we can access dictionary elements.



```
d = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Accessing an element using key
print(d['name'])

# Accessing a element using get
print(d.get(3))
```



```
For
Geeks
```

- **Operators**

Python operators are special symbols that perform operations on variables and values. There are several categories of operators in Python:

- **Arithmetic Operators**

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

- ## Comparison (Relational) Operators

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

- ## Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assigne value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator,	c **= a is equivalent to c = c ** a

	Performs exponential (power) calculation on operators and assign value to the left operand	
//=	Floor Dividion and assigns a value, Performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

- **Bitwise operator.**

Bitwise operator works on bits and perform bit by bit operation

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(a | b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

- **Logical operators**

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

- **Membership Operators -** In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below :

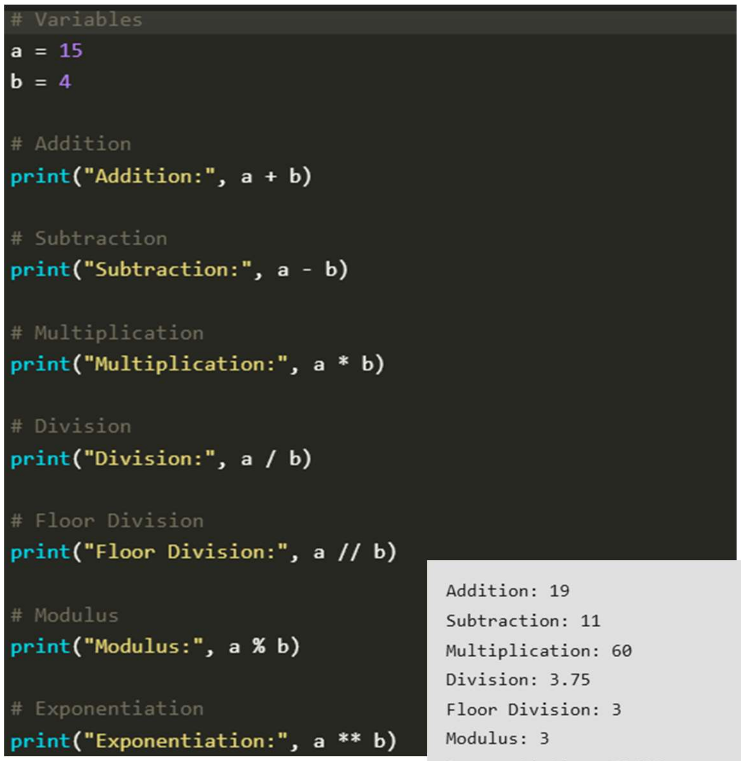
Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here **not in** results in a 1 if x is not a member of sequence y.

- **Identity Operators:**

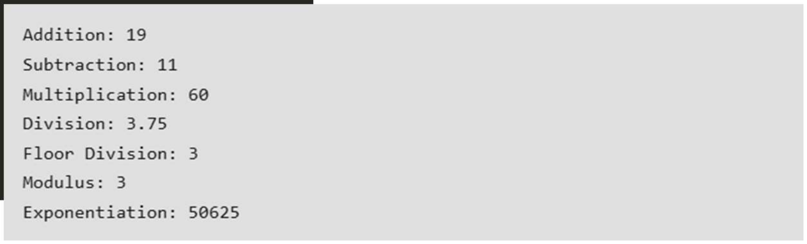
Identity operators compare the memory locations of two objects. There are two Identity operators explained below

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Example

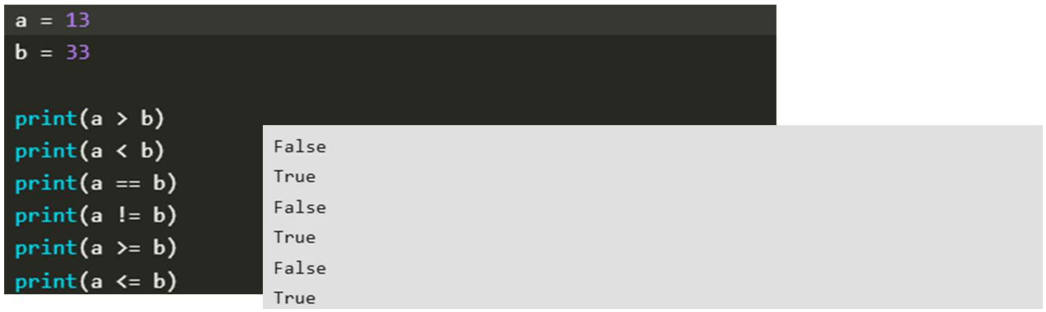


```python
# Variables
a = 15
b = 4

# Addition
print("Addition:", a + b)

# Subtraction
print("Subtraction:", a - b)

# Multiplication
print("Multiplication:", a * b)

# Division
print("Division:", a / b)

# Floor Division
print("Floor Division:", a // b)

# Modulus
print("Modulus:", a % b)

# Exponentiation
print("Exponentiation:", a ** b)
```



```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Floor Division: 3
Modulus: 3
Exponentiation: 50625
```

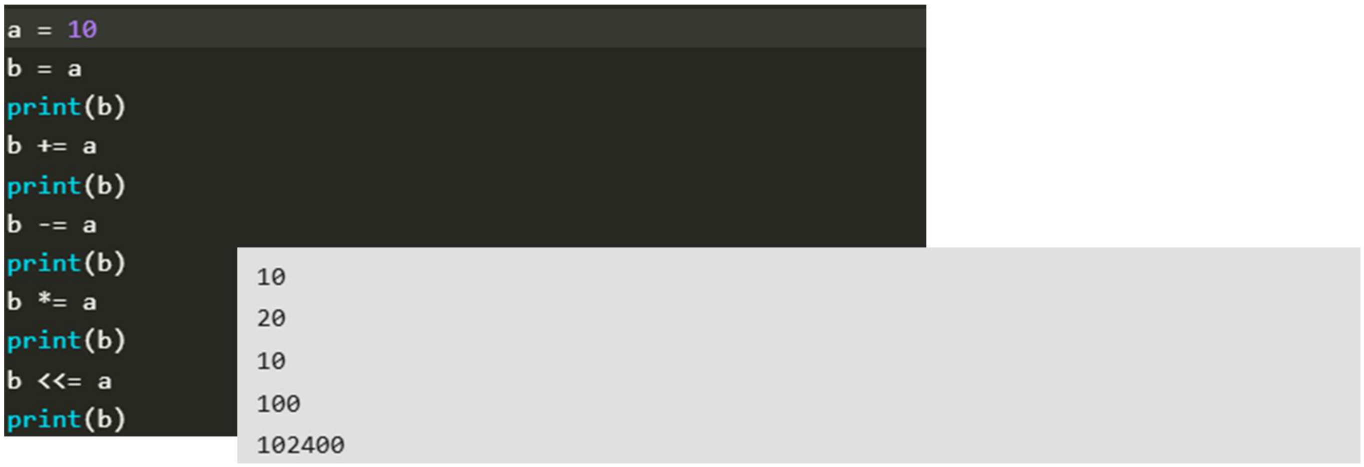
Example 2



```python
a = 13
b = 33

print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

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

Example



```python
a = 10
b = a
print(b)
b += a
print(b)
b -= a
print(b)
b *= a
print(b)
b <<= a
print(b)
```

```
10
20
10
100
102400
```

- **Conditional Structures and loops (if, for, while).**

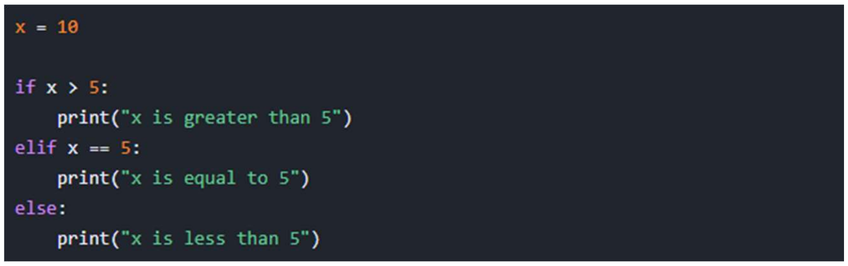
Loops in Python are used to repeat actions efficiently. The main types are, IF loops, For loops (counting through items) and While loops (based on conditions)..

- **iF loop**

In Python, an "if loop" is not a distinct control flow structure. Instead, the term likely refers to the use of an if statement within a for or while loop, or the use of an if statement for conditional execution, which is distinct from looping.

  - **if statement for conditional execution**

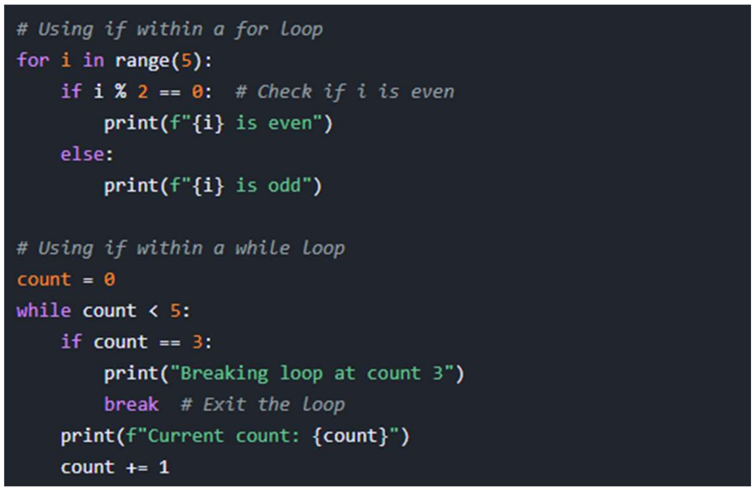
The if statement in Python allows code to be executed only when a specified condition is true. This is a fundamental concept of conditional logic.



```python
x = 10

if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

- **if statement within a loop**

An if statement can be placed inside a for or while loop to conditionally execute code for specific iterations of the loop.



```python
# Using if within a for loop
for i in range(5):
    if i % 2 == 0:  # Check if i is even
        print(f"{i} is even")
    else:
        print(f"{i} is odd")

# Using if within a while loop
count = 0
while count < 5:
    if count == 3:
        print("Breaking loop at count 3")
        break  # Exit the loop
    print(f"Current count: {count}")
    count += 1
```

- **Key points**

- The if, elif (else if), and else keywords are used to define conditional blocks.
- Conditions are expressions that evaluate to a Boolean value (True or False).
- Indentation is crucial in Python to define the scope of the if, elif, and else blocks.
- break and continue statements can be used within loops to control their flow, often in conjunction with if statements. break exits the loop entirely, while continue skips the current iteration and proceeds to the next.

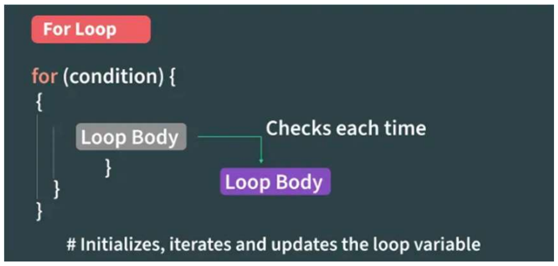
○ **For Loop**

For loops is used to iterate over a sequence such as a list, tuple, string or range. It allow to execute a block of code repeatedly, once for each item in the sequence.

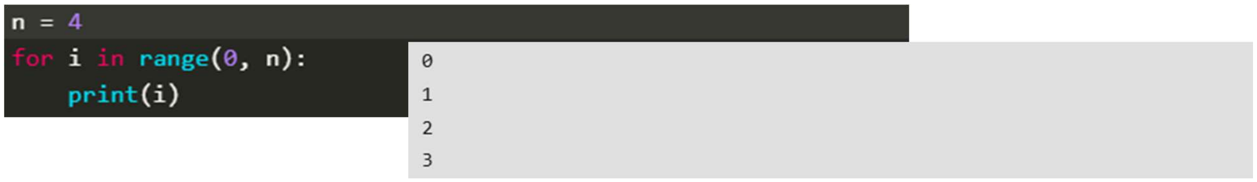
Syntax:

for iterator_var in sequence:

statements(s)



Example



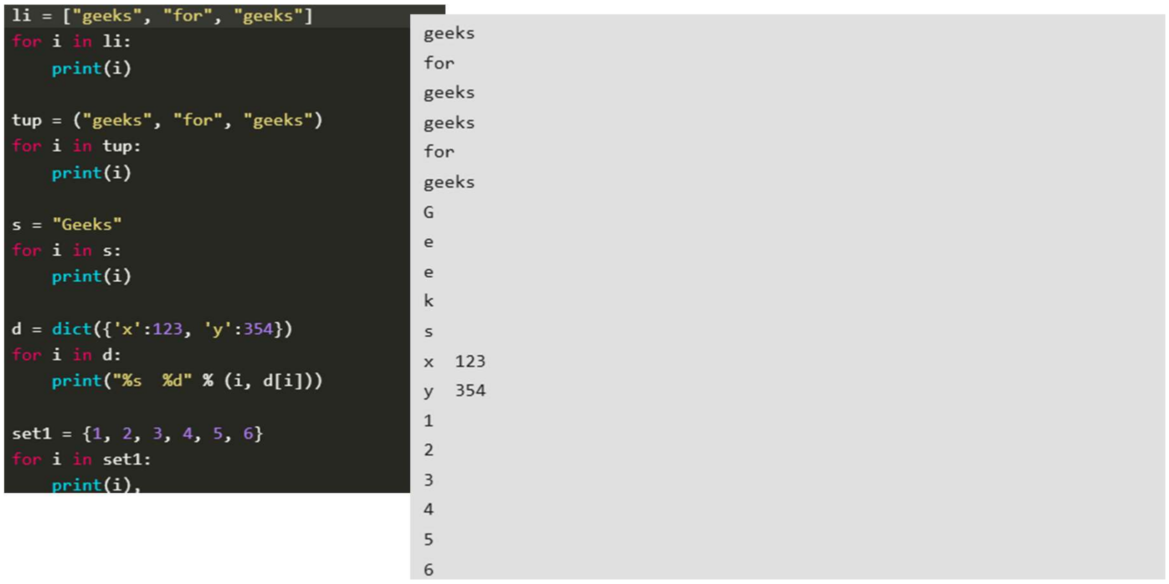
```python
n = 4
for i in range(0, n):
    print(i)
```

```
0
1
2
3
```

Explanation: This code prints the numbers from 0 to 3 (inclusive) using a for loop that iterates over a range from 0 to n-1 (where n = 4).

Example

Iterating Over List, Tuple, String and Dictionary Using for Loops in Python

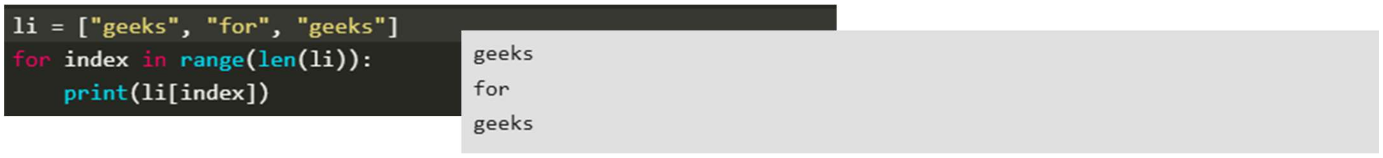


```
li = ["geeks", "for", "geeks"]
for i in li:
    print(i)

tup = ("geeks", "for", "geeks")
for i in tup:
    print(i)

s = "Geeks"
for i in s:
    print(i)

d = dict({'x':123, 'y':354})
for i in d:
    print("%s  %d" % (i, d[i]))

set1 = {1, 2, 3, 4, 5, 6}
for i in set1:
    print(i),
```

```
geeks
for
geeks
geeks
for
geeks
G
e
e
k
s
x  123
y  354
1
2
3
4
5
6
```

- **Iterating by Index of Sequences**

We can also use the index of elements in the sequence to iterate. The key idea is to first calculate the length of the list and then iterate over the sequence within the range of this length.



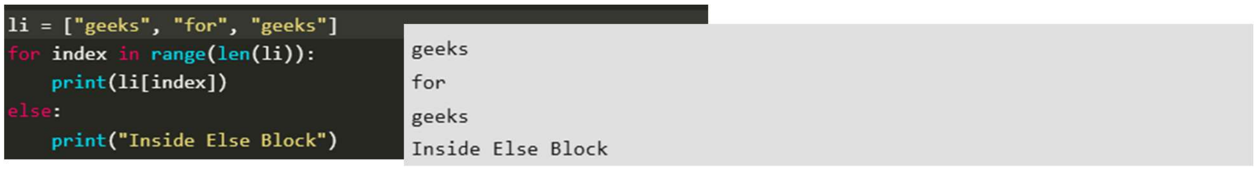
```
li = ["geeks", "for", "geeks"]
for index in range(len(li)):
    print(li[index])
```

```
geeks
for
geeks
```

Explanation- This code iterates through each element of the list using its index and prints each element one by one. The **range(len(list))** generates indices from 0 to the length of the list minus 1.

- **Using else Statement with for Loop**

We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.



```
li = ["geeks", "for", "geeks"]
for index in range(len(li)):
    print(li[index])
else:
    print("Inside Else Block")
```

```
geeks
for
geeks
Inside Else Block
```

Explanation: The code iterates through the list and prints each element. After the loop ends it prints "Inside Else Block" as the else block executes when the loop completes without a break.

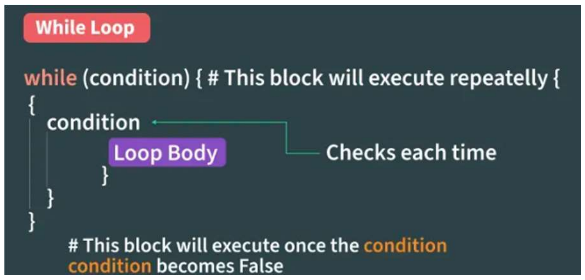
○ While Loop

In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

Syntax:

while expression:

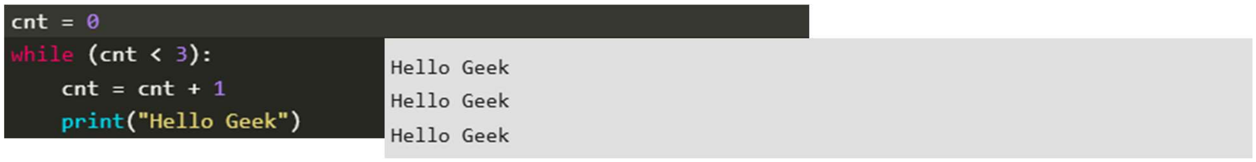
statement(s)


While Loop
while (condition) { # This block will execute repeatedly {
{
condition
Loop Body
}
}
}
Checks each time
# This block will execute once the condition condition becomes False

▪ while loop explanation

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

In below code, loop runs as long as the condition cnt < 3 is true. It increments the counter by 1 on each iteration and prints "Hello Geek" three times.



```python
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello Geek")
```

```
Hello Geek
Hello Geek
Hello Geek
```

▪ Using else statement with While Loop

Else clause is only executed when our while condition becomes false. If we break out of the loop or if an exception is raised then it won't be executed.

Syntax:

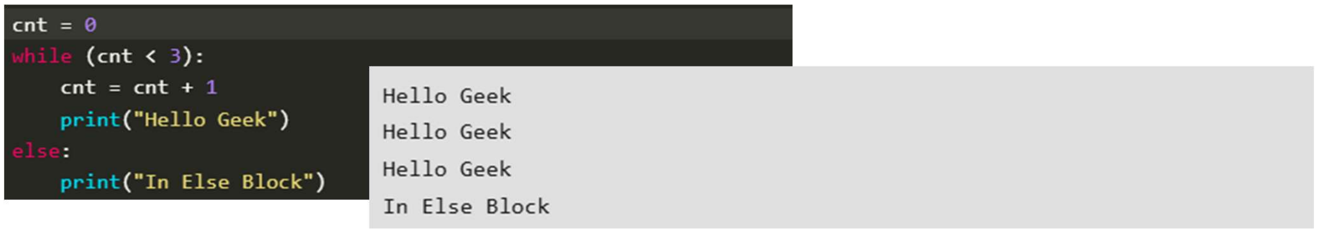
while condition:

# execute these statements

else:

# execute these statements

Code prints "Hello Geek" three times using a 'while' loop and then after the loop it prints "In Else Block" because there is an "else" block associated with the 'while' loop.

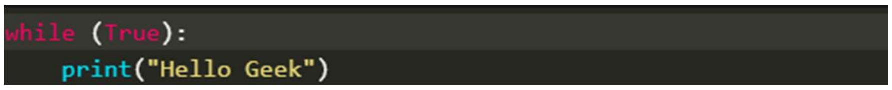


```python
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello Geek")
else:
    print("In Else Block")
```

```
Hello Geek
Hello Geek
Hello Geek
In Else Block
```

- **Infinite While Loop**

If we want a block of code to execute infinite number of times then we can use the while loop in Python to do so.
Code given below uses a 'while' loop with the condition "**True**", which means that the loop will run infinitely until we break out of it using "**break"** keyword or some other logic.

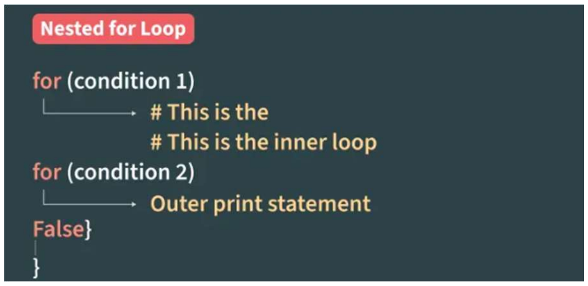


```
while (True):
    print("Hello Geek")
```

Note- ____
it is suggested not to use this type of loop as it is a never-ending infinite loop where the condition is always true and we have to forcefully terminate the compiler.

- **Nested Loops**

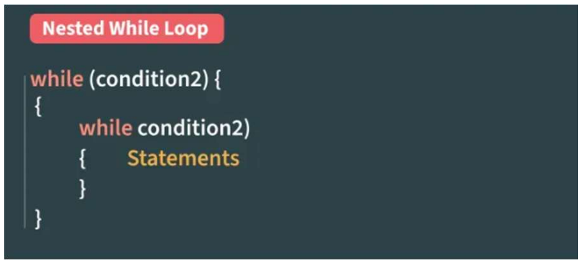
Python programming language allows to use one loop inside another loop which is called nested loop. Following section shows few examples to illustrate the concept.
Syntax:
for iterator_var in sequence:
for iterator_var in sequence:
statements(s)
statements(s)



```
Nested for Loop

for (condition 1)
 └──────────→ # This is the
              # This is the inner loop
for (condition 2)
 └──────────→ Outer print statement
False}
}
```

- **Nested for loop**

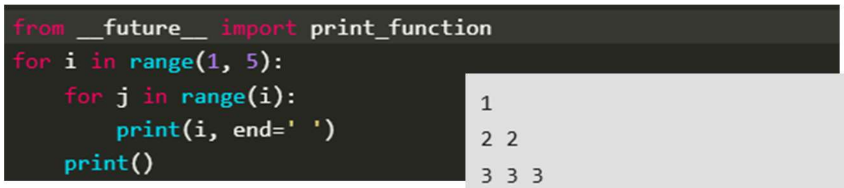
The syntax for a nested while loop statement in the Python programming language is as follows:
while expression:
while expression:
statement(s)
statement(s)



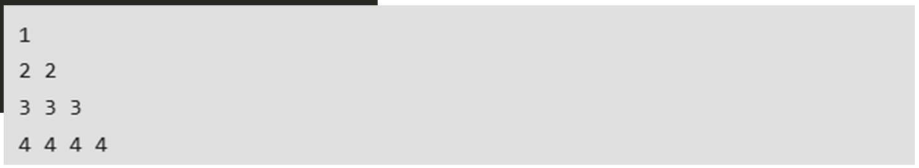
```
Nested While Loop

while (condition2) {
{
     while condition2)
     {     Statements
     }
}
```

- **Nested while loop**

A final note on loop nesting is that we can put any type of loop inside of any other type of loops in Python. For example, a for loop can be inside a while loop or vice versa.



```python
from __future__ import print_function
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```



```
1
2 2
3 3 3
4 4 4 4
```

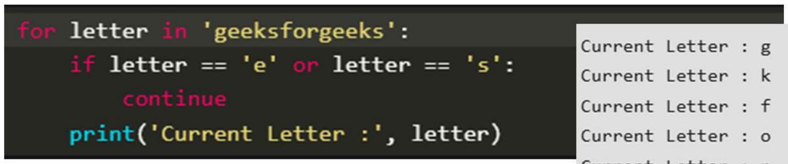
Explanation: In the above code we use nested loops to print the value of i multiple times in each row, where the number of times it prints i increases with each iteration of the outer loop. The print() function prints the value of i and moves to the next line after each row.

- **Loop Control Statements**

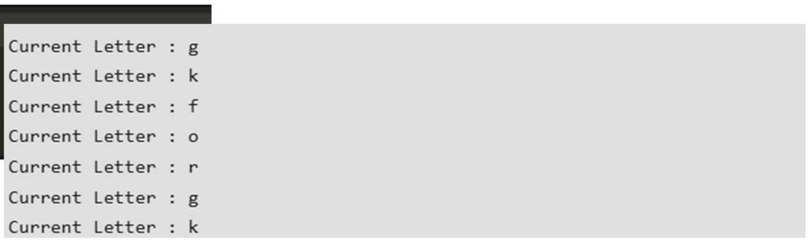
Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Continue Statement

The continue statement in Python returns the control to the beginning of the loop.



```python
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        continue
    print('Current Letter :', letter)
```

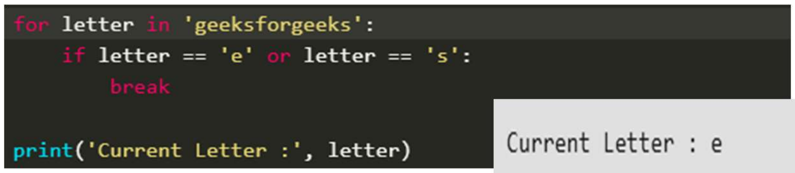


```
Current Letter : g
Current Letter : k
Current Letter : f
Current Letter : o
Current Letter : r
Current Letter : g
Current Letter : k
```

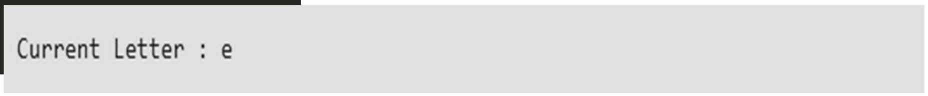
Explanation -The continue statement is used to skip the current iteration of a loop and move to the next iteration. It is useful when we want to bypass certain conditions without terminating the loop.

- **Break Statement**

The break statement in Python brings control out of the loop.



```python
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        break

print('Current Letter :', letter)
```

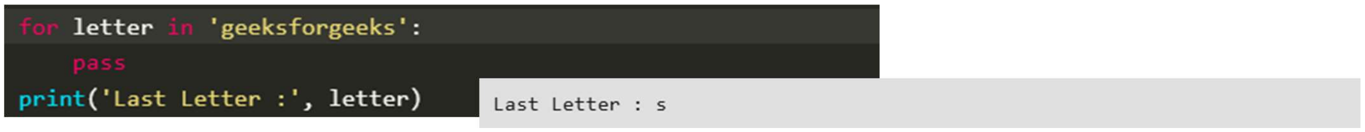


```
Current Letter : e
```

Explanation- break statement is used to exit the loop prematurely when a specified condition is met. In this example, the loop breaks when the letter is either 'e' or 's', stopping further iteration.

- **Pass Statement**

We use pass statement in Python to write empty loops. Pass is also used for empty control statements, functions and classes.



```python
for letter in 'geeksforgeeks':
    pass
print('Last Letter :', letter)
```

```
Last Letter : s
```

Explanation: In this example, the loop iterates over each letter in 'geeksforgeeks' but doesn't perform any operation, and after the loop finishes, the last letter ('s') is printed.

- **Functions**

- **Predefined function**

Predefined functions in Python, often referred to as built-in functions, are functions that are readily available for use without needing to be explicitly imported from a module. They are part of the core Python language and provide fundamental functionalities for various common programming tasks. Their main key characteristics are:

- **Always Available**

Built-in functions are always accessible in any Python program or interactive interpreter session without requiring an import statement.

- **Core Functionality**

  They cover a wide range of essential operations, including:

  - **Mathematical operations:** abs(), round(), pow(), min(), max(), sum().
  - **Type conversions:** int(), float(), str(), list(), tuple(), dict(), set().
  - **Input/Output:** print(), input().
  - **Working with iterables:** len(), range(), enumerate(), map(), filter(), sorted(), zip().
  - **Object inspection and manipulation:** type(), id(), dir(), hasattr(), getattr(), setattr().
  - **Efficiency and Reliability:**

Being implemented in C (for CPython), these functions are generally highly optimized for performance and are thoroughly tested, ensuring reliability.

Examples

- print("Hello, world!") displays output to the console.
- length = len("Python") returns the number of characters in a string.
- absolute_value = abs(-5) returns the positive equivalent of a number.
- numbers = list(range(1, 6)) creates a list of numbers from 1 to 5.

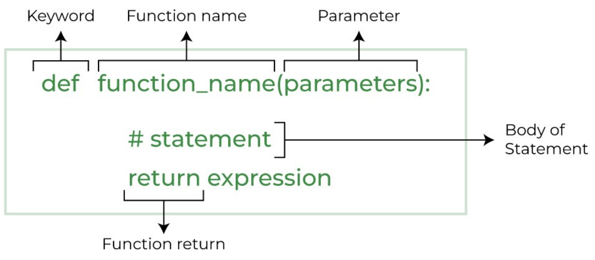
In essence, predefined functions streamline development by providing a readily available set of tools for common programming challenges, allowing developers to focus on higher-level logic rather than re-implementing basic functionalities. A full summary of all predefined function is mentioned in **Annexe1.**

○ Creating a function in Python

Python Functions are a block of statements that does a specific task. The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

▪ Function Declaration

The syntax to declare a function is:



▪ Defining a Function

We can define a function in Python, using the **def keyword**. We can add any type of functionalities and properties to it as we require.
The def keyword stands for define. It is used to create a **user-defined function**. It marks the beginning of a function block and allows you to group a set of statements so they can be reused when the function is called.

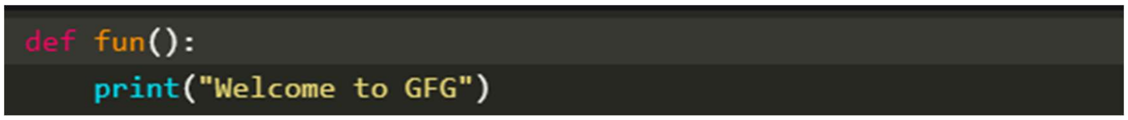
Syntax:

def function_name(parameters):
# function body

Explanation-

- **def:** Starts the function definition.
- **function_name:** Name of the function.
- **parameters:** Inputs passed to the function (inside ()), optional.
- **Indented code:** The function body that runs when called.

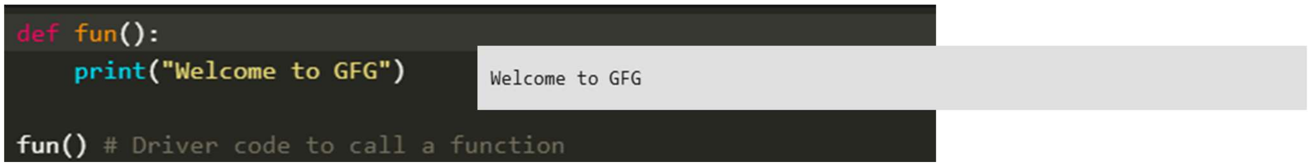
Here, we define a function using def that prints a welcome message when called.



```
def fun():
    print("Welcome to GFG")
```

▪ Calling a Function

After creating a function in Python we can call it by using the name of the functions followed by parenthesis containing parameters of that particular function.



```
def fun():
    print("Welcome to GFG")

fun() # Driver code to call a function
```

```
Welcome to GFG
```

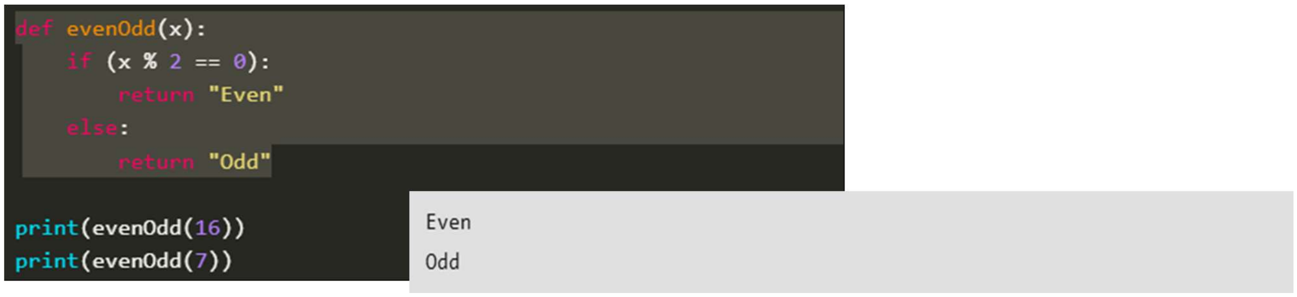
- **Function Arguments**

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Syntax:

def function_name(parameters):
"""Docstring"""
# body of the function
return expression

We will create a simple function in Python to check whether the number passed as an argument to the function is even or odd.



```
def evenOdd(x):
    if (x % 2 == 0):
        return "Even"
    else:
        return "Odd"

print(evenOdd(16))
print(evenOdd(7))
```

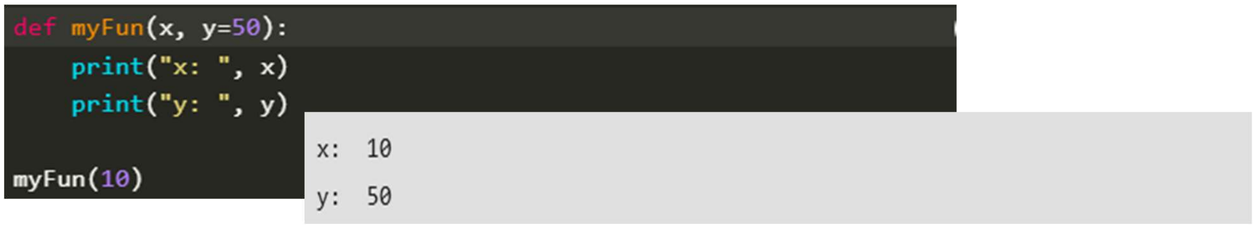
```
Even
Odd
```

- **Types of Function Arguments**

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following function argument types in Python, Let's explore them one by one.

- **Default Arguments**

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

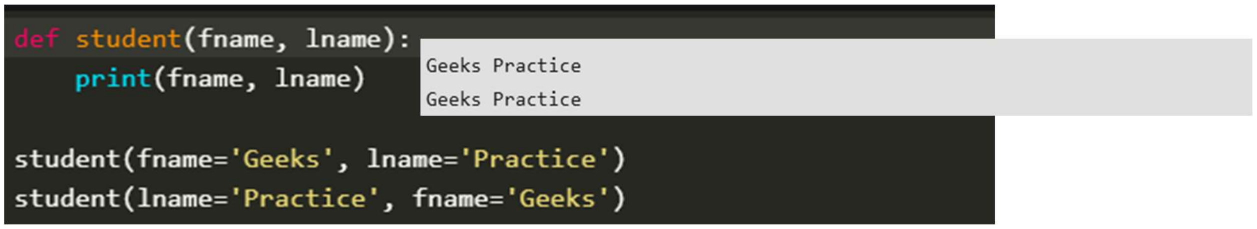


```
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)

myFun(10)
```

```
x:  10
y:  50
```

- **Keyword Arguments**

In keyword arguments, values are passed by explicitly specifying the parameter names, so the order doesn't matter.

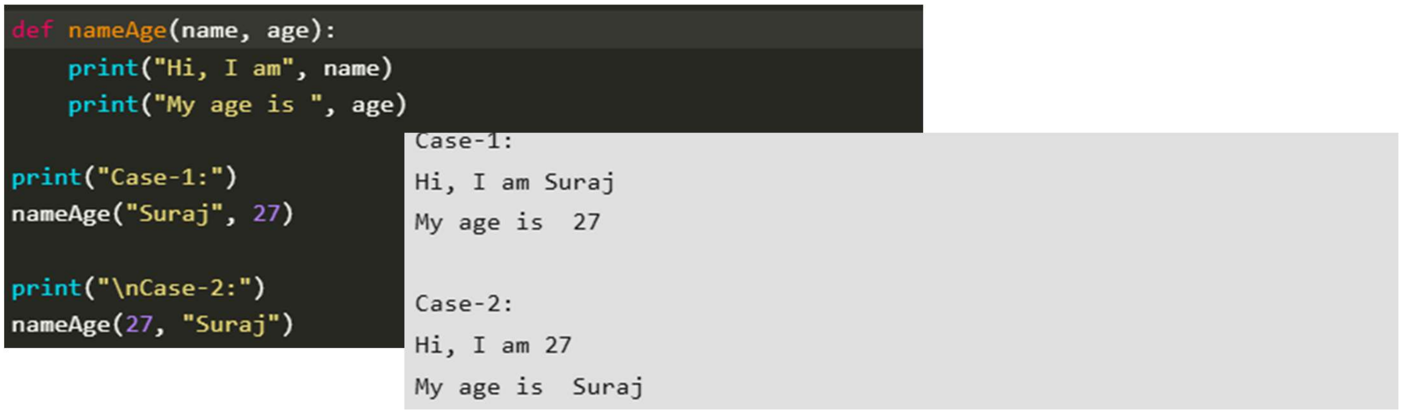


```python
def student(fname, lname):
    print(fname, lname)

student(fname='Geeks', lname='Practice')
student(lname='Practice', fname='Geeks')
```

```
Geeks Practice
Geeks Practice
```

- **Positional Arguments**

In positional arguments, values are assigned to parameters based on their order in the function call.



```python
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)

print("Case-1:")
nameAge("Suraj", 27)

print("\nCase-2:")
nameAge(27, "Suraj")
```

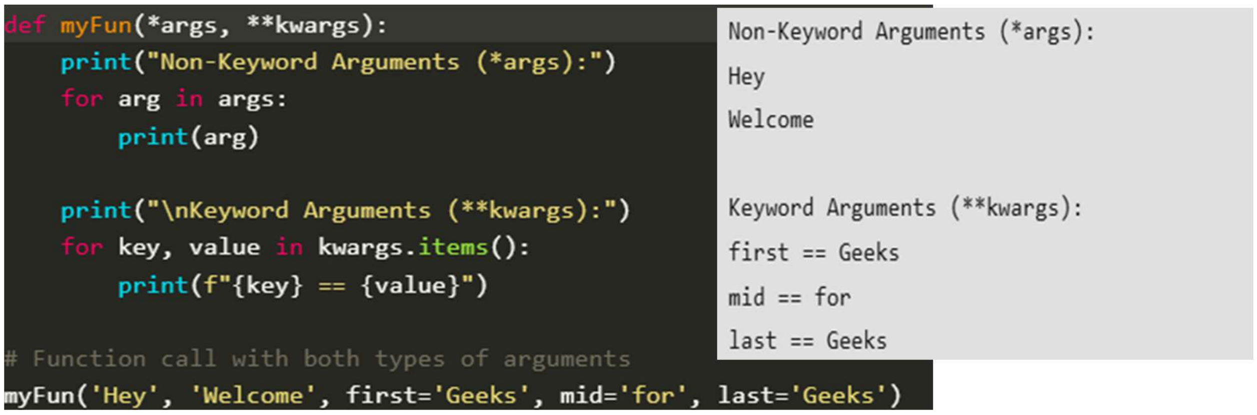
```
Case-1:
Hi, I am Suraj
My age is  27

Case-2:
Hi, I am 27
My age is  Suraj
```

- **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:

- ***args** *in Python (Non-Keyword Arguments)*
- ***kwargs** *in Python (Keyword Arguments)*

This code separately shows non-keyword (*args) and keyword (**kwargs) arguments in the same function.

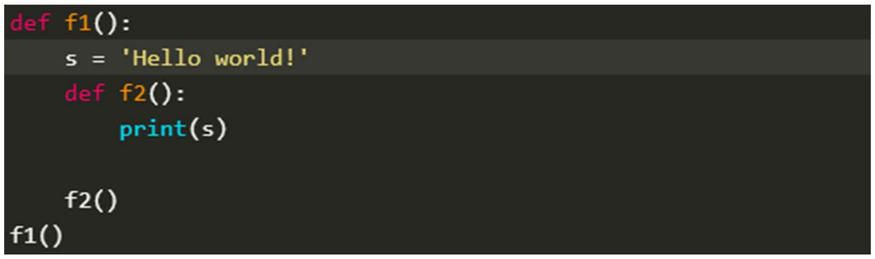


```python
def myFun(*args, **kwargs):
    print("Non-Keyword Arguments (*args):")
    for arg in args:
        print(arg)

    print("\nKeyword Arguments (**kwargs):")
    for key, value in kwargs.items():
        print(f"{key} == {value}")

# Function call with both types of arguments
myFun('Hey', 'Welcome', first='Geeks', mid='for', last='Geeks')
```

```
Non-Keyword Arguments (*args):
Hey
Welcome

Keyword Arguments (**kwargs):
first == Geeks
mid == for
last == Geeks
```

- **Function within Functions**

A function defined inside another function is called an inner function (or nested function). It can access variables from the enclosing function's scope and is often used to keep logic protected and organized.



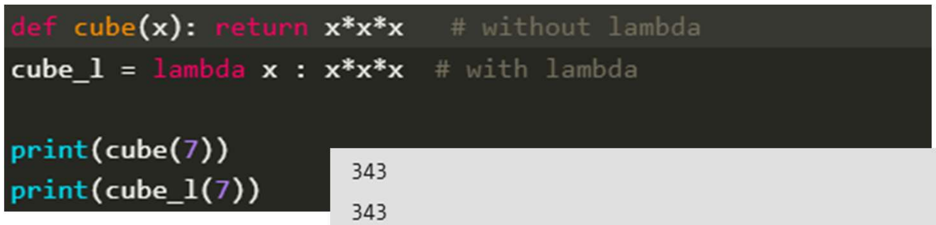
```python
def f1():
    s = 'Hello world!'
    def f2():
        print(s)

    f2()
f1()
```

Output

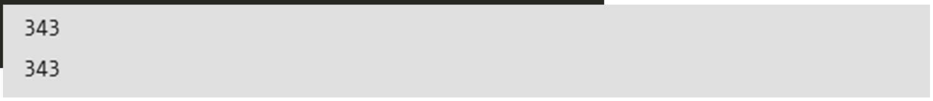
```
Hello world
```

- **Anonymous Functions**

In Python, an anonymous function means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.



```python
def cube(x): return x*x*x    # without lambda
cube_l = lambda x : x*x*x  # with lambda

print(cube(7))
print(cube_l(7))
```



```
343
343
```

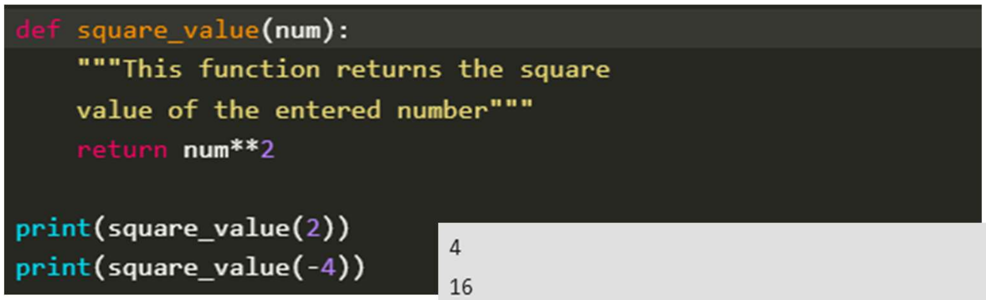
- **Return Statement in Function**

The return statement ends a function and sends a value back to the caller. It can return any data type, multiple values (packed into a tuple), or None if no value is given.

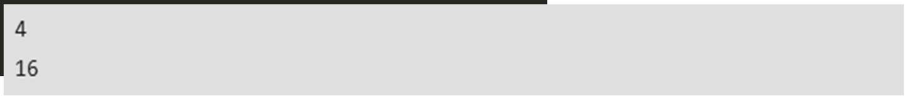
Syntax:

return [expression]

Parameters: return ends the function, **[expression]** is the optional value to return (defaults to None)**.**



```python
def square_value(num):
    """This function returns the square
    value of the entered number"""
    return num**2

print(square_value(2))
print(square_value(-4))
```



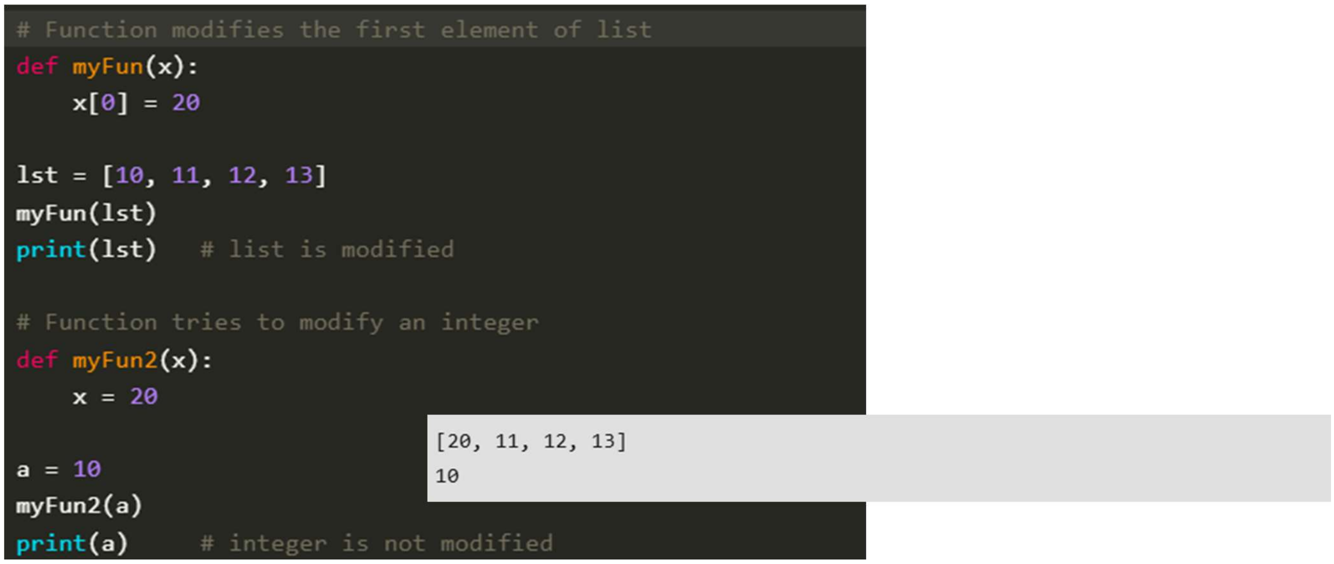
```
4
16
```

- **Pass by Reference and Pass by Value**

In Python, variables are references to objects. When we pass them to a function, the behavior depends on whether the object is mutable (like lists, dictionaries) or immutable (like integers, strings, tuples).

- **Mutable objects:** Changes inside the function affect the original object.

- **Immutable objects:** The original value remains unchanged.



```python
# Function modifies the first element of list
def myFun(x):
    x[0] = 20

lst = [10, 11, 12, 13]
myFun(lst)
print(lst)   # list is modified

# Function tries to modify an integer
def myFun2(x):
    x = 20

a = 10
myFun2(a)
print(a)      # integer is not modified
```

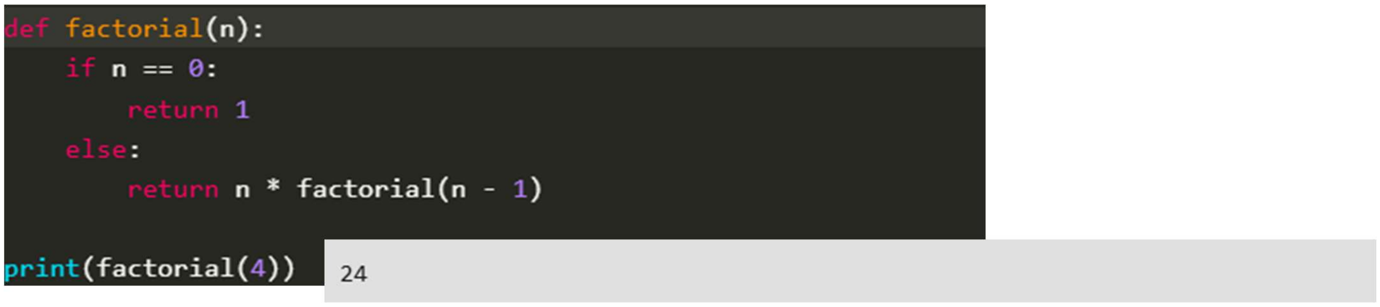
```
[20, 11, 12, 13]
10
```

Note

Technically, Python uses "pass-by-object-reference". Mutable objects behave like pass by reference, while immutable objects behave like pass by value

- **Recursive Functions**

A recursive function is a function that calls itself to solve a problem. It is commonly used in mathematical and divide-and-conquer problems. Always include a base case to avoid infinite recursion.



```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4))
```

```
24
```

- **Standard Modules**

A standard module in Python refers to a module that is part of the Python Standard Library. Here's a breakdown of what that entails:

- **Pre-installed and readily available:**

When you install Python, the entire Standard Library, including all its standard modules, is installed along with it. This means you do not need to install them separately using tools like pip.

- **Collection of pre-written code:**

The Standard Library is a vast collection of modules that provide a wide range of functionalities for common programming tasks. These include modules for working with:

- **Operating system interactions:** os, sys, shutil
- **Data handling:** json, csv, datetime
- **Networking:** socket, http
- **Mathematics:** math, random
- **Text processing:** string, re (regular expressions)

- **And many more specialized areas.**
  - **Importable for use:**

To use the functionalities provided by a standard module in your Python code, you need to explicitly import it. For example, to use the sqrt function from the math module, you would write:

```python
import math
result = math.sqrt(25)
print(result)
```

- **Lambda function**

A lambda function in Python is a small, anonymous function defined using the lambda keyword that can take any number of arguments but has only one expression.

Syntax

The basic syntax for a lambda function is:

```
lambda arguments: expression
```

The result of the expression is automatically returned when the lambda function is called, so you do not use the return keyword.

Examples

Basic Use

You can assign a lambda function to a variable and call it like a normal function.

Add 10 to a number:

```python
add_ten = lambda x: x + 10
print(add_ten(5))
# Output: 15
```

Multiply two numbers:

```python
multiply = lambda a, b: a * b
print(multiply(5, 6))
# Output: 30
```

Conditional logic (if-else)

```python
even_or_odd = lambda x: "Even" if x % 2 == 0 else "Odd"
print(even_or_odd(4))
print(even_or_odd(7))
# Output: Even
# Output: Odd
```

Common Use Cases with Higher-Order Functions

Lambda functions are often used with built-in functions that take other functions as arguments, such as map(), filter(), and sorted().

- map(): Apply a function to each item in an iterable.

```python
numbers = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)
# Output: [2, 4, 6, 8, 10]
```

- filter(): Create an iterable of elements for which a function returns True

```python
numbers = [1, 2, 3, 4, 5, 6]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)
# Output: [1, 3, 5]
```

- sorted(): Sort a list using a specific key.

```python
students = [("Emil", 25), ("Tobias", 22), ("Linus", 28)]
sorted_students = sorted(students, key=lambda x: x[1])
print(sorted_students)
# Output: [('Tobias', 22), ('Emil', 25), ('Linus', 28)]
```

○ What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.
Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

▪ Create a Python Module

To create a Python module, write the desired code and save that in a file with **.py** extension. Let's understand it better with an example:

Example

*Let's create a simple calc.py in which we define two functions, one **add** and another **subtract**.*

```python
# A simple module, calc.py
def add(x, y):
    return (x+y)

def subtract(x, y):
    return (x-y)
```

- **Import module in Python**

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.
When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

Note

A search path is a list of directories that the interpreter searches for importing a module.

For example, to import the module calc.py, we need to put the following command at the top of the script. Syntax to Import Module in Python

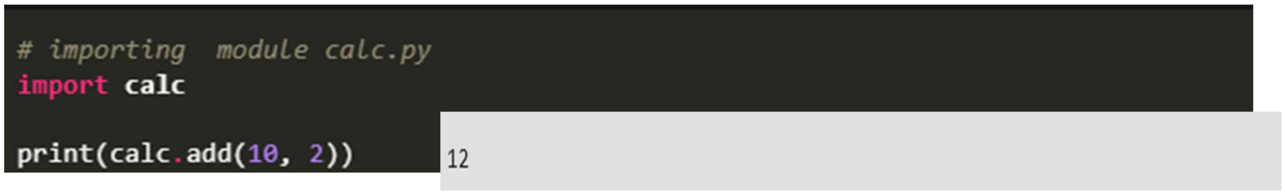
Import module

Note

This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

Importing modules in Python Example

Now, we are importing the calc that we created earlier to perform add operation.



```
# importing  module calc.py
import calc

print(calc.add(10, 2))
```

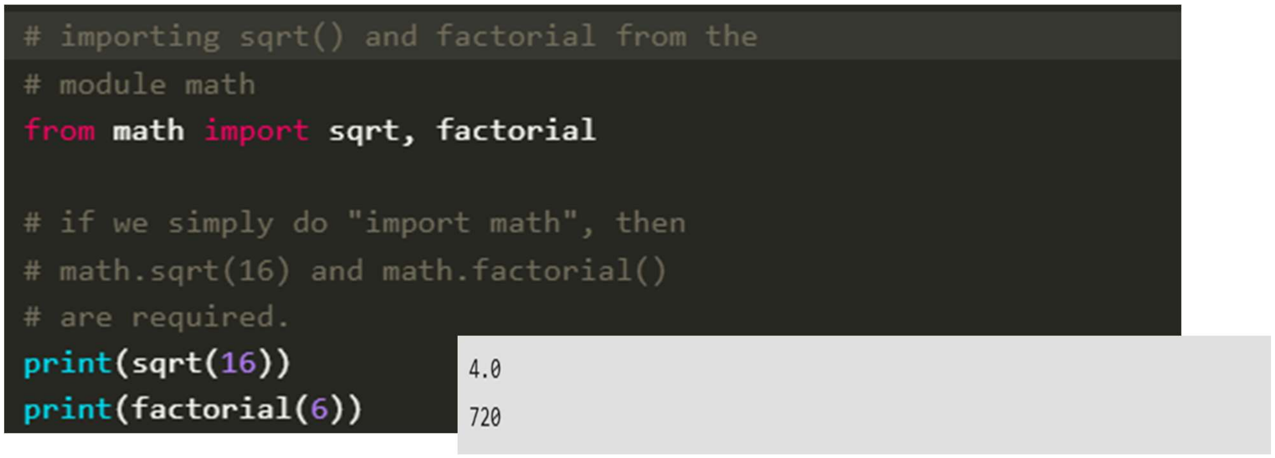
```
12
```

- **Python Import From Module**

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

Import Specific Attributes from a Python module

Here, we are importing specific sqrt and factorial attributes from the math module.



```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

```
4.0
720
```

- **Import all Names**

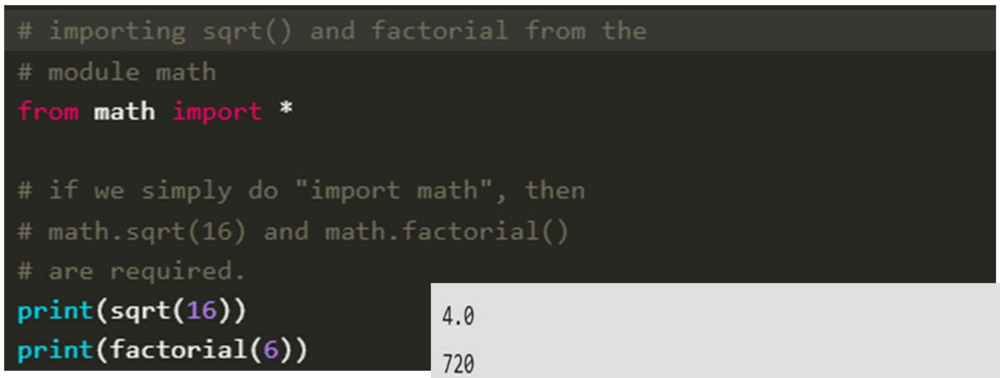
The * symbol used with the import statement is used to import all the names from a module to a current namespace.

Syntax:

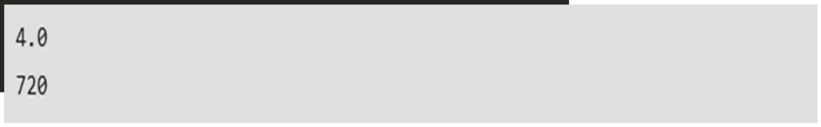
```
from module_name import *
```

- **What does import * do in Python?**

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.



```
# importing sqrt() and factorial from the
# module math
from math import *

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```



```
4.0
720
```

- **Locating Python Modules**

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path. Python interpreter searches for the module in the following manner -

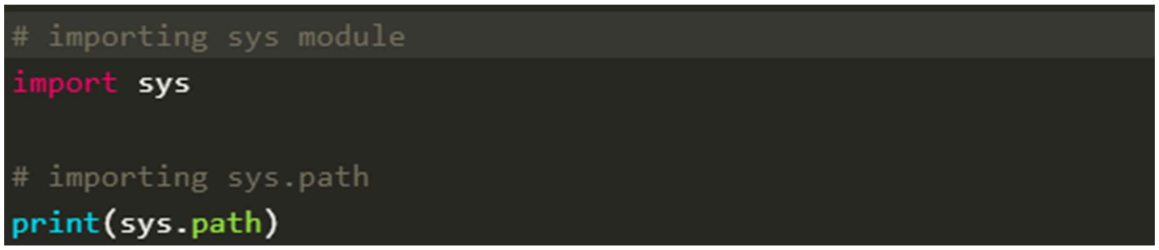
First, it searches for the module in the current directory.

If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.

If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

Directories List for Modules

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.



```
# importing sys module
import sys

# importing sys.path
print(sys.path)
```

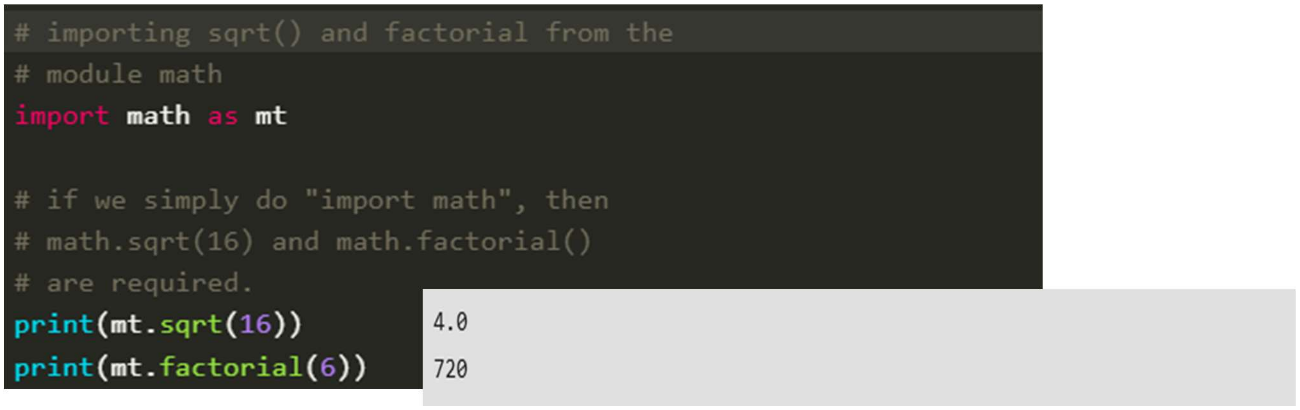
Output -

['/home/nikhil/Desktop/gfg', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload', '', '/home/nikhil/.local/lib/python3.8/site-packages', '/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages', '/usr/local/lib/python3.8/dist-packages/IPython/extensions', '/home/nikhil/.ipython']

- **Renaming the Python Module**

We can rename the module while importing it using the keyword.

Syntax: *Import **Module_name** as **Alias_name***

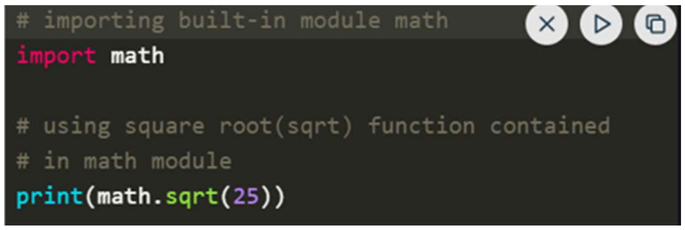


```
# importing sqrt() and factorial from the
# module math
import math as mt

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(mt.sqrt(16))
print(mt.factorial(6))
```

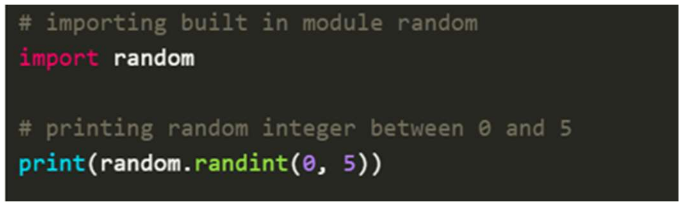
```
4.0
720
```

- **Python Built-in modules**

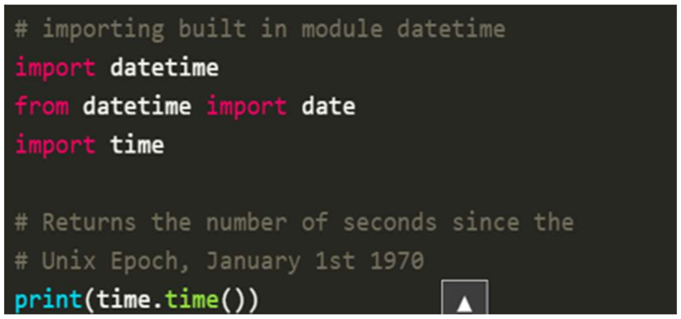
There are several built-in modules in Python, which you can import whenever you like



```
# importing built-in module math
import math

# using square root(sqrt) function contained
# in math module
print(math.sqrt(25))
```



```
# importing built in module random
import random

# printing random integer between 0 and 5
print(random.randint(0, 5))
```



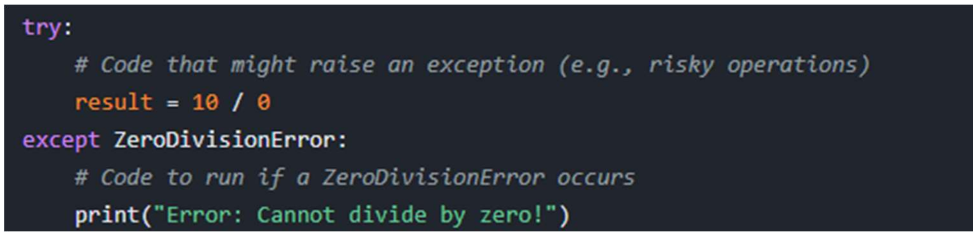
```
# importing built in module datetime
import datetime
from datetime import date
import time

# Returns the number of seconds since the
# Unix Epoch, January 1st 1970
print(time.time())
```

- **Try-except block**

The try...except block in Python is a mechanism for handling runtime errors (exceptions) gracefully, preventing the program from crashing. It allows you to test a block of code for errors and execute specific code to handle the error if one occurs.

- **Basic Syntax and Usage**

The basic structure involves a try block containing the code that might raise an exception, and an except block that catches and handles the error.



```python
try:
    # Code that might raise an exception (e.g., risky operations)
    result = 10 / 0
except ZeroDivisionError:
    # Code to run if a ZeroDivisionError occurs
    print("Error: Cannot divide by zero!")
```

In the above example, the division by zero in the try block raises a ZeroDivisionError, causing the program to jump to the except block and print the error message instead of stopping.
Try and Except statement is used to handle these errors within our code in Python.
The try block is used to check some code for errors i.e the code inside the try block will execute when there is no error in the program.
Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block.

Syntax

try:
# Some Code
except:
# Executed if error in the
# try block

- **Objective**

- Pevents crashes caused by runtime errors.
- Handles specific exceptions like division by zero, file not found, etc.
- Improves code reliability and error tolerance.
- Allows custom error messages and fallback logic. used for robust, debuggable applications.

- **How try() works?**

- First, the try clause is executed i.e. the code between try.
- If there is no exception, then only the try clause will run, except clause is finished.
- If any exception occurs, the try clause will be skipped and except clause will run.
- If any exception occurs, but the except clause within the code doesn't handle it, it is passed on to the outer try statements. If the exception is left unhandled, then the execution stops.
- A try statement can have more than one except clause

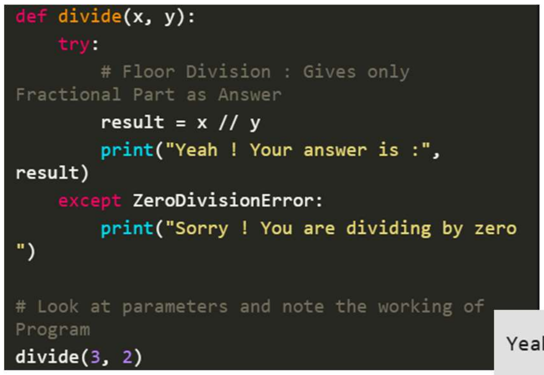
- **Some of the common Exception Errors are**

- **IOError:** if the file can't be opened
- **KeyboardInterrupt:** when an unrequired key is pressed by the user
- **ValueError:** when the built-in function receives a wrong argument

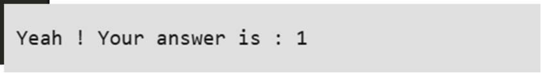
- **EOFError:** if End-Of-File is hit without reading any data
- **ImportError:** if it is unable to find the module

Example 1

*No exception, so the **try** clause will run.*



```
def divide(x, y):
    try:
        # Floor Division : Gives only
Fractional Part as Answer
        result = x // y
        print("Yeah ! Your answer is :",
result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero
")

# Look at parameters and note the working of
Program
divide(3, 2)
```



```
Yeah ! Your answer is : 1
```

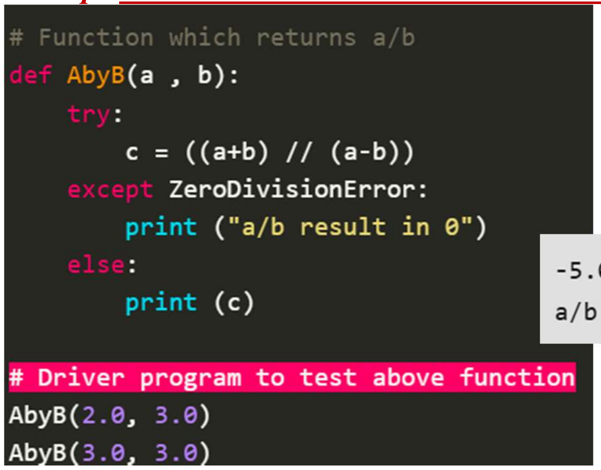
- **Else Clause**

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

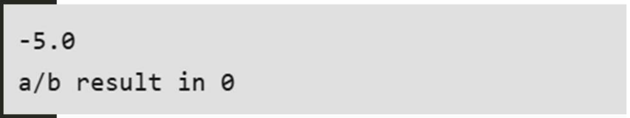
Syntax

try:
# Some Code
except:
# Executed if error in the
# try block
else:
# execute if no exception

Example



```
# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) // (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)

# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```



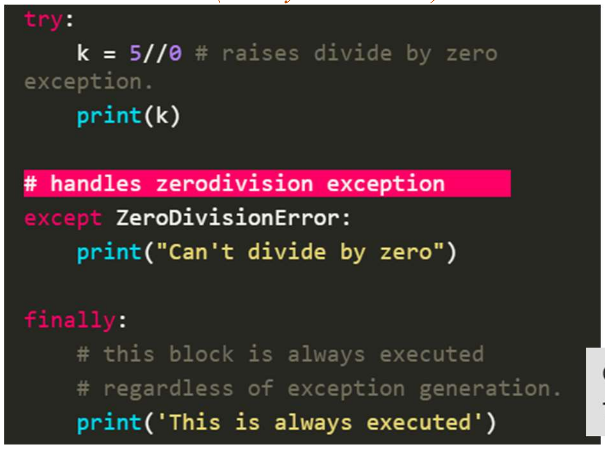
```
-5.0
a/b result in 0
```

- **Finally Keyword in Python**

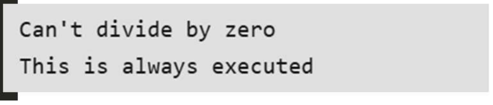
Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exceptions.

Syntax

try:
# Some Code
except:
# Executed if error in the
# try block
else:
# execute if no exception
finally:
# Some code .....(always executed)



```
try:
    k = 5//0 # raises divide by zero
exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

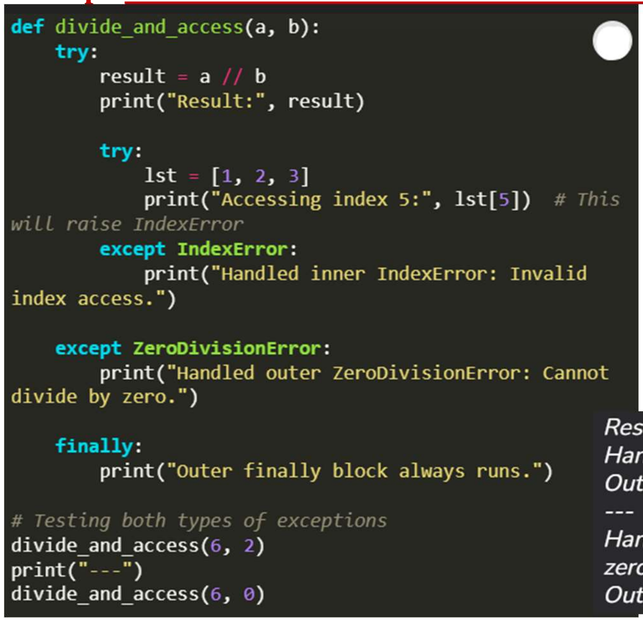


```
Can't divide by zero
This is always executed
```

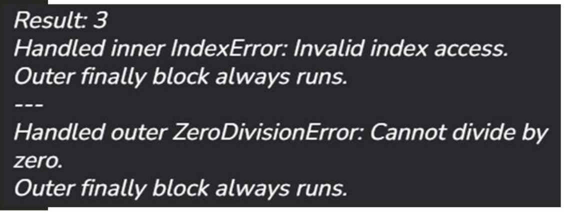
▪ Nested Try-Except Blocks in Python

In Python, you can nest try-except blocks to handle exceptions at **multiple levels**. This is useful when different parts of the code may raise different types of exceptions and need separate handling.

Example



```
def divide_and_access(a, b):
    try:
        result = a // b
        print("Result:", result)

        try:
            lst = [1, 2, 3]
            print("Accessing index 5:", lst[5])  # This
will raise IndexError
        except IndexError:
            print("Handled inner IndexError: Invalid
index access.")

    except ZeroDivisionError:
        print("Handled outer ZeroDivisionError: Cannot
divide by zero.")

    finally:
        print("Outer finally block always runs.")

# Testing both types of exceptions
divide_and_access(6, 2)
print("---")
divide_and_access(6, 0)
```



```
Result: 3
Handled inner IndexError: Invalid index access.
Outer finally block always runs.
---
Handled outer ZeroDivisionError: Cannot divide by
zero.
Outer finally block always runs.
```

Chapter 2
Oriented Object programming

Python Classes/Objects

Python is an object oriented programming language.
Almost everything in Python is an object, with its properties and methods.
A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

ExampleGet your own Python Server

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

Delete Objects

You can delete objects by using the del keyword:

Example

Delete the p1 object:

```
del p1
```

Multiple Objects

You can create multiple objects from the same class:

Example

Create three objects from the MyClass class:

```
p1 = MyClass()
p2 = MyClass()

print(p1.x)
print(p2.x)
```

Note: Each object is independent and has its own copy of the class properties.

The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Example

```
class Person:
  pass
```

Python __init__() Method

All classes have a built-in method called __init__(), which is always executed when the class is being initiated. The __init__() method is used to assign values to object properties, or to perform operations that are necessary when the object is being created.

Example Create a class named Person, use the __init__() method to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("Emil", 36)
print(p1.name)
print(p1.age)
```

Note: The __init__() method is called automatically every time the class is being used to create a new object.

Why Use __init__()?

Without the __init__() method, you would need to set properties manually for each object:

Example- Create a class without __init__():

```
class Person:
  pass

p1 = Person()
```

```
p1.name = "Tobias"
p1.age = 25

print(p1.name)
print(p1.age)
```

Using __init__() makes it easier to create objects with initial values:

Example- With __init__(), you can set initial values when creating the object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Linus", 28)

print(p1.name)
print(p1.age)
```

Default Values in __init__()

You can also set default values for parameters in the __init__() method:

Example - Set a default value for the age parameter:

```
class Person:
  def __init__(self, name, age=18):
    self.name = name
    self.age = age

p1 = Person("Emil")
p2 = Person("Tobias", 25)

print(p1.name, p1.age)
print(p2.name, p2.age)
```

Multiple Parameters

The __init__() method can have as many parameters as you need:

Example- Create a Person class with multiple parameters:

```
class Person:
  def __init__(self, name, age, city, country):
    self.name = name
    self.age = age
    self.city = city
    self.country = country
```

```
p1 = Person("Linus", 30, "Oslo", "Norway")

print(p1.name)
print(p1.age)
print(p1.city)
print(p1.country)
```

The self Parameter

The self parameter is a reference to the current instance of the class. It is used to access properties and methods that belong to the class.

Example - Use self to access class properties:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def greet(self):
    print("Hello, my name is " + self.name)

p1 = Person("Emil", 25)
p1.greet()
```

Note: The self parameter must be the first parameter of any method in the class.

Why Use self?

Without self, Python would not know which object's properties you want to access:

Example - The self parameter links the method to the specific object:

```
class Person:
  def __init__(self, name):
    self.name = name

  def printname(self):
    print(self.name)
p1 = Person("Tobias")
p2 = Person("Linus")
p1.printname()
p2.printname()
```

---

self Does Not Have to Be Named "self"

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any method in the class:

Example

Use the words *myobject* and *abc* instead of *self*:

```
class Person:
  def __init__(myobject, name, age):
    myobject.name = name
    myobject.age = age

  def greet(abc):
    print("Hello, my name is " + abc.name)

p1 = Person("Emil", 36)
p1.greet()
```

Note: While you *can* use a different name, it is strongly recommended to use self as it is the convention in Python and makes your code more readable to others.

Accessing Properties with self

You can access any property of the class using self:

Example

Access multiple properties using self:

```
class Car:
  def __init__(self, brand, model, year):
    self.brand = brand
    self.model = model
    self.year = year

  def display_info(self):
    print(f"{self.year} {self.brand} {self.model}")

car1 = Car("Toyota", "Corolla", 2020)
car1.display_info()
```

Calling Methods with self

You can also call other methods within the class using self:

Example

Call one method from another method using self:

```
class Person:
  def __init__(self, name):
    self.name = name
  def greet(self):
    return "Hello, " + self.name
  def welcome(self):
    message = self.greet()
    print(message + "! Welcome to our website.")
p1 = Person("Tobias")
p1.welcome()
```

Class Properties

Properties are variables that belong to a class. They store data for each object created from the class.

Example Create a class with properties:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("Emil", 36)

print(p1.name)
print(p1.age)
```

Access Properties

You can access object properties using dot notation:

Example

Access the properties of an object:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

car1 = Car("Toyota", "Corolla")

print(car1.brand)
print(car1.model)
```

Modify Properties

You can modify the value of properties on objects:

Example

Change the age property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Tobias", 25)
print(p1.age)

p1.age = 26
print(p1.age)
```

Delete Properties

You can delete properties from objects using the del keyword:

Example

Delete the age property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Linus", 30)

del p1.age

print(p1.name) # This works
# print(p1.age) # This would cause an error
```

Class Properties vs Object Properties

Properties defined inside __init__() belong to each object (instance properties).

Properties defined outside methods belong to the class itself (class properties) and are shared by all objects:

Example

Class property vs instance property:

```
class Person:
  species = "Human" # Class property

  def __init__(self, name):
    self.name = name # Instance property

p1 = Person("Emil")
p2 = Person("Tobias")

print(p1.name)
print(p2.name)
print(p1.species)
print(p2.species)
```

Modifying Class Properties

When you modify a class property, it affects all objects:

Example
Change a class property:

```
class Person:
  lastname = ""
  def __init__(self, name):
    self.name = name
p1 = Person("Linus")
p2 = Person("Emil")
Person.lastname = "Refsnes"
print(p1.lastname)
print(p2.lastname)
```

Add New Properties

You can add new properties to existing objects:

Example - Add a new property to an object:

```
class Person:
  def __init__(self, name):
    self.name = name
p1 = Person("Tobias")
p1.age = 25
p1.city = "Oslo"

print(p1.name)
print(p1.age)
print(p1.city)
```

Note: Adding properties this way only adds them to that specific object, not to all objects of the class.

Python Class Methods

Methods are functions that belong to a class. They define the behavior of objects created from the class.

Example--Create a method in a class:

```
class Person:
  def __init__(self, name):
    self.name = name
  def greet(self):
    print("Hello, my name is " + self.name)

p1 = Person("Emil")
p1.greet()
```

Note: All methods must have self as the first parameter.

Methods with Parameters

Methods can accept parameters just like regular functions:

Example- Create a method with parameters:

```
class Calculator:
  def add(self, a, b):
    return a + b
  def multiply(self, a, b):
    return a * b

calc = Calculator()
print(calc.add(5, 3))
print(calc.multiply(4, 7))
```

Methods Accessing Properties

Methods can access and modify object properties using self:

Example--A method that accesses object properties:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def get_info(self):
    return f"{self.name} is {self.age} years old"

p1 = Person("Tobias", 28)
print(p1.get_info())
```

Methods Modifying Properties

Methods can modify the properties of an object:

Example-A method that changes a property value:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def celebrate_birthday(self):
    self.age += 1
    print(f"Happy birthday! You are now {self.age}")
p1 = Person("Linus", 25)
p1.celebrate_birthday()
p1.celebrate_birthday()
```

The __str__() Method

The __str__() method is a special method that controls what is returned when the object is printed:

Example--Without the __str__() method:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Emil", 36)
print(p1)
```

Example--With the __str__() method:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name} ({self.age})"
```

```
p1 = Person("Tobias", 36)
print(p1)
```

Multiple Methods

A class can have multiple methods that work together:

Example- Create multiple methods in a class:

```
class Playlist:
  def __init__(self, name):
    self.name = name
    self.songs = []

  def add_song(self, song):
    self.songs.append(song)
    print(f"Added: {song}")

  def remove_song(self, song):
    if song in self.songs:
      self.songs.remove(song)
      print(f"Removed: {song}")

  def show_songs(self):
    print(f"Playlist '{self.name}':")
    for song in self.songs:
      print(f"- {song}")

my_playlist = Playlist("Favorites")
my_playlist.add_song("Bohemian Rhapsody")
my_playlist.add_song("Stairway to Heaven")
my_playlist.show_songs()
```

Delete Methods

You can delete methods from a class using the del keyword:

Example- Delete a method from a class:

```
class Person:
  def __init__(self, name):
    self.name = name
  def greet(self):
    print("Hello!")

p1 = Person("Emil")

del Person.greet

p1.greet() # This will cause an error
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example- Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example- Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
  pass
```

Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example-Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent. We want to add the __init__() function to the child class (instead of the pass keyword).

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Example- Add the __init__() function to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.

Note: The child's __init__() function **overrides** the inheritance of the parent's __init__() function.

To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the __init__() function.

Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Example

Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function:

Example

Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Example

Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Function Polymorphism

An example of a Python function that can be used on different objects is the len() function.
String
For strings len() returns the number of characters:
Example
```
x = "Hello World!"
print(len(x))
```
Tuple
For tuples len() returns the number of items in the tuple:
Example
```
mytuple = ("apple", "banana", "cherry")

print(len(mytuple))
```
Dictionary

For dictionaries len() returns the number of key/value pairs in the dictionary:

Example
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(len(thisdict))
```

Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

Example-Different classes with the same method:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Drive!")
class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Sail!")
class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747")     #Create a Plane object

for x in (car1, boat1, plane1):
  x.move()
```

Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.

Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there? Yes. If we use the example above and make a parent class called Vehicle, and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them:

Example- Create a class called Vehicle and make Car, Boat, Plane child classes of Vehicle:

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Move!")
class Car(Vehicle):
  pass
class Boat(Vehicle):
  def move(self):
    print("Sail!")
class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747")     #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle.The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method. Because of polymorphism we can execute the same method for all classes.

Python Encapsulation

Encapsulation is about protecting data inside a class.

It means keeping data (properties) and methods together in a class, while controlling how the data can be accessed from outside the class.This prevents accidental changes to your data and hides the internal details of how your class works.

Private Properties

In Python, you can make properties private by using a double underscore __ prefix:

Example

Create a private class property named __age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age # Private property

p1 = Person("Emil", 25)
print(p1.name)
print(p1.__age) # This will cause an error
```

Note: Private properties cannot be accessed directly from outside the class.

Get Private Property Value

To access a private property, you can create a getter method:

Example-Use a getter method to access a private property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age

  def get_age(self):
    return self.__age

p1 = Person("Tobias", 25)
print(p1.get_age())
```

Set Private Property Value

To modify a private property, you can create a setter method. The setter method can also validate the value before setting it:

Example-Use a setter method to change a private property:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age
  def get_age(self):
    return self.__age
  def set_age(self, age):
    if age > 0:
      self.__age = age
    else:
      print("Age must be positive")
p1 = Person("Tobias", 25)
print(p1.get_age())

p1.set_age(26)
print(p1.get_age())
```

Why Use Encapsulation?

Encapsulation provides several benefits:

- **Data Protection:** Prevents accidental modification of data
- **Validation:** You can validate data before setting it
- **Flexibility:** Internal implementation can change without affecting external code
- **Control:** You have full control over how data is accessed and modified

Example - Use encapsulation to protect and validate data:

```
class Student:
  def __init__(self, name):
    self.name = name
    self.__grade = 0
  def set_grade(self, grade):
    if 0 <= grade <= 100:
      self.__grade = grade
    else:
      print("Grade must be between 0 and 100")
  def get_grade(self):
    return self.__grade

  def get_status(self):
    if self.__grade >= 60:
      return "Passed"
    else:
      return "Failed"

student = Student("Emil")
student.set_grade(85)
print(student.get_grade())
print(student.get_status())
```

Protected Properties

Python also has a convention for protected properties using a single underscore _ prefix:

Example-Create a protected property:

```
class Person:
  def __init__(self, name, salary):
    self.name = name
    self._salary = salary # Protected property

p1 = Person("Linus", 50000)
print(p1.name)
print(p1._salary) # Can access, but shouldn't
```

Note: A single underscore _ is just a convention. It tells other programmers that the property is intended for internal use, but Python doesn't enforce this restriction.

Private Methods

You can also make methods private using the double underscore prefix:

Example

Create a private method:

```
class Calculator:
  def __init__(self):
    self.result = 0
  def __validate(self, num):
    if not isinstance(num, (int, float)):
      return False
    return True
  def add(self, num):
    if self.__validate(num):
      self.result += num
    else:
      print("Invalid number")
calc = Calculator()
```

```
calc.add(10)
calc.add(5)
print(calc.result)
# calc.__validate(5) # This would cause an error
```

Note: Just like private properties with double underscores, private methods cannot be called directly from outside the class. The __validate method can only be used by other methods inside the class.

Name Mangling

Name mangling is how Python implements private properties and methods.
When you use double underscores __, Python automatically renames it internally by adding _ClassName in front. For example, __age becomes _Person__age.

Example- See how Python mangles the name:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age
p1 = Person("Emil", 30)

# This is how Python mangles the name:
print(p1._Person__age) # Not recommended!
```

Note: While you *can* access private properties using the mangled name, it's not recommended. It defeats the purpose of encapsulation

Python Inner Classes

An inner class is a class defined inside another class. The inner class can access the properties and methods of the outer class.
Inner classes are useful for grouping classes that are only used in one place, making your code more organized.

Example -Create an inner class:

```
class Outer:
  def __init__(self):
    self.name = "Outer Class"
  class Inner:
    def __init__(self):
      self.name = "Inner Class"
    def display(self):
      print("This is the inner class")
```

```
outer = Outer()
print(outer.name)
```

Accessing Inner Class from the Outside

To access the inner class, create an object of the outer class, and then create an object of the inner class:

Example - Access the inner class and create an object:

```
class Outer:
  def __init__(self):
    self.name = "Outer"
  class Inner:
    def __init__(self):
      self.name = "Inner"
    def display(self):
      print("Hello from inner class")
outer = Outer()
inner = outer.Inner()
inner.display()
```

Accessing Outer Class from Inner Class

Inner classes in Python do not automatically have access to the outer class instance.
If you want the inner class to access the outer class, you need to pass the outer class instance as a parameter:

Example - Pass the outer class instance to the inner class:

```
class Outer:
  def __init__(self):
    self.name = "Emil"
  class Inner:
    def __init__(self, outer):
      self.outer = outer
    def display(self):
      print(f"Outer class name: {self.outer.name}")
outer = Outer()
inner = outer.Inner(outer)
inner.display()
```

Practical Example

Inner classes are useful for creating helper classes that are only used within the context of the outer class:

Example - Use an inner class to represent a car's engine:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
    self.engine = self.Engine()
  class Engine:
    def __init__(self):
      self.status = "Off"

    def start(self):
      self.status = "Running"
      print("Engine started")
    def stop(self):
      self.status = "Off"
      print("Engine stopped")
  def drive(self):
    if self.engine.status == "Running":
      print(f"Driving the {self.brand} {self.model}")
    else:
      print("Start the engine first!")
car = Car("Toyota", "Corolla")
car.drive()
car.engine.start()
car.drive()
```

Multiple Inner Classes

A class can have multiple inner classes:

Example-Create multiple inner classes:

```
class Computer:
  def __init__(self):
    self.cpu = self.CPU()
    self.ram = self.RAM()
  class CPU:
    def process(self):
      print("Processing data...")
  class RAM:
    def store(self):
      print("Storing data...")
```

```
computer = Computer()
computer.cpu.process()
computer.ram.store()
```

Chapter 3

Python for Control Engineering

1. Introduction to Differential Equations

A differential equation is a mathematical equation that relates some function with its derivatives.

In applications, the functions usually represent physical quantities, the derivatives represent their rates of change, and the differential equation defines a relationship between the two.

Because such relations are extremely common, differential equations play a prominent role in many disciplines including engineering, physics, economics, and biology.

We typically want to solve ordinary differential equations (ODE) of the form:

$$\frac{dy}{dt} = f(t, y), y(t_0) = y_0$$

Example 1. Example of Dynamic System

Given the following differential equation:

$$\dot{x} = -ax + bu$$

Note! $\dot{x}$ is the same as $\frac{dx}{dt}$

We have the following:

- x - Process variable, e.g., Level, Pressure, Temperature, etc.
- u - Input variable, e.g., Control Signal from the Controller
- a, b - Constants

With Python have we can solve these differential equations in many different ways.

We can use so-called ODE solvers or we can make discrete version of the differential equations using discretization methods like Euler, etc.

With ODE solvers Python can solve these equations numerically. Higher order differential equations must be reformulated into a system of first order differential equations.

Example 2 Differential equation example

Given the following differential equation:

$$\dot{x} = ax$$

Where $a = -\frac{1}{T}$, where T is defined as the time constant of the system.

Note! $\dot{x}$ is the same as $\frac{dx}{dt}$

The solution for the differential equation is found to be:

$$x(t) = e^{at}x_0$$

We shall plot the solution for this differential equation using Python.

In our system we can set $T = 5$ and the initial condition $x_0 = x(0) = 1$
Python code:

```
import math as mt
import numpy as np
import matplotlib.pyplot as plt


# Parameters
T = 5
a = -1/T

x0 = 1
t = 0

tstart = 0
tstop = 25

increment = 1

x = []
x = np.zeros(tstop+1)

t = np.arange(tstart ,tstop+1,increment)
```

```python


# Define the Equation
for k in range(tstop):
    x[k] = mt.exp(a*t[k]) * x0


# Plot the Results
plt.plot(t,x)
plt.title('Plotting Differential Equation Solution')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid()
plt.axis([0, 25, 0, 1])
plt.show()
```

This gives the plot shown in Figure 1.

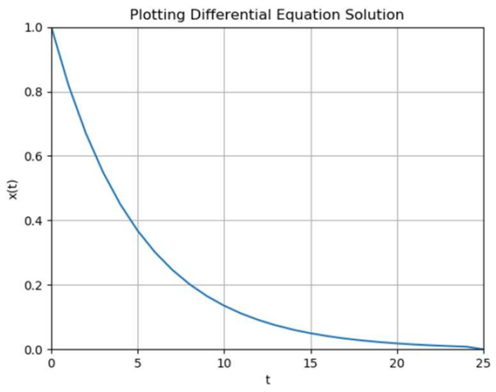


Figure .1: Plotting Difrential Equation Solution

ODE Solvers in Python

The **scipy.integrate** library has two powerful powerful functions **ode()** and **odeint()**, for numerically solving firrst order ordinary differential equations (ODEs). The **ode()** is more flexible, while **odeint()** (ODE integrator) has a simpler Python interface works fine for most problems.
For details, see the **SciPy** documentation:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html
https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.ode.html

Example 2. Using ODE Solver in Python

Given the following differential equation:

$$\dot{x} = ax$$

Where $a = -\frac{1}{T}$, where T is defined as the time constant of the system.

Note! $\dot{x}$ is the same as $\frac{dx}{dt}$

We will use the odeint() function.

The syntax is as follows:

```
x = odeint(functionname, x0, t)
```

Where we have:
functioname: Function that returns derivative values at requested x and t values as dxdt = model(x,t)

x0: Initial conditions of the differential states

t: Time points at which the solution should be reported. Additional internal points are often calculated to maintain accuracy of the solution but are not reported.

Where we first has to define our differential equation:

```
def functionname(x, t):
    dxdt = a * x
    return dxdt
```

The Python code becomes:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Initialization
tstart = 0
tstop = 25
increment = 1

x0 = 1
t = np.arange(tstart,tstop+1,increment)


# Function that returns dx/dt
def mydiff(x, t):
    T = 5
    a = -1/T

    dxdt = a * x

    return dxdt


# Solve ODE
x = odeint(mydiff, x0, t)
print(x)


# Plot the Results
plt.plot(t,x)
plt.title('Plotting Differential Equation Solution')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid()
plt.axis([0, 25, 0, 1])
plt.show()
```

Some modification to the Python code:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Initialization
tstart = 0
tstop = 25
increment = 1

T = 5
a = -1/T
x0 = 1
t = np.arange(tstart,tstop+1,increment)


# Function that returns dx/dt
def mydiff(x, t, a):

    dxdt = a * x

    return dxdt


# Solve ODE
x = odeint(mydiff, x0, t, args=(a,))
print(x)


# Plot the Results
plt.plot(t,x)
plt.title('Plotting Differential Equation Solution')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid()
plt.axis([0, 25, 0, 1])

plt.show()
```

In the modified example we have the parameters used in the differential equation (in this case a) as an input argument. By doing this, it is very easy to changes values for the parameters used in the differential equation without changing the code for the differential equation.
You can also easily run multiple simulations like this.

```
a = -0.2
x = odeint(mydiff, x0, t, args=(a,))

a = -0.1
x = odeint(mydiff, x0, t, args=(a,))
```

Solving Multiple 1. order Differential Equations

In real life we typically have higher order differential equations, or we have a set of 1. order differential equations that describe a given system. How can we solve such equations in Python?
The following set of 1.order Differential Equations. Given the differential equations:

$$\frac{dx}{dt} = -y$$

$$\frac{dy}{dt} = x$$

Assume the initial conditions $x(0) = 1$ and $y(0) = 1$.

The Python code is almost similar as previous examples, but we need to do some small trick to make it work.

Python code:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Initialization
tstart = -1
tstop = 1
increment = 0.1

# Initial condition
z0 = [1,1]


t = np.arange(tstart,tstop+1,increment)


# Function that returns dx/dt
def mydiff(z, t):
    dxdt = -z[1]
    dydt = z[0]

    dzdt = [dxdt,dydt]
    return dzdt

```

```

# Solve ODE
z = odeint(mydiff, z0, t)
print(z)

x = z[:,0]
y = z[:,1]


# Plot the Results
plt.plot(t,x)
plt.plot(t,y)
plt.title('Plotting Differential Equations Solution')
plt.xlabel('t')
plt.ylabel('z(t)')
plt.grid()
plt.axis([-1, 1, -1.5, 1.5])
plt.show()
```

This gives the plot shown in Figure .2.

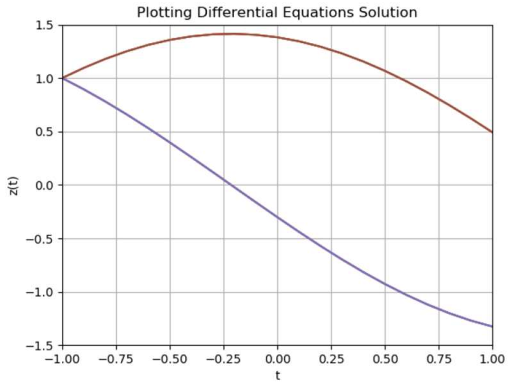


Figure 2

We can also rewrite the differential equations like this:

$$\frac{dx_1}{dt} = -x_2$$

$$\frac{dx_2}{dt} = x_1$$

The Python code then becomes:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Initialization
tstart = -1
tstop = 1
increment = 0.1

# Initial condition
x_init = [1,1]


t = np.arange(tstart,tstop+1,increment)


# Function that returns dx/dt
def mydiff(x, t):
    dx1dt = -x[1]
    dx2dt = x[0]

    dxdt = [dx1dt, dx2dt]
    return dxdt


# Solve ODE
x = odeint(mydiff, x_init, t)
print(x)

x1 = x[:,0]
x2 = x[:,1]


# Plot the Results
plt.plot(t,x1)
plt.plot(t,x2)
plt.title('Plotting Differential Equations Solution')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid()
plt.axis([-1, 1, -1.5, 1.5])
plt.show()
```

Solving Higher order Differential Equations

We shall use Python to solve and plot the results of the following differential equation:

We shall use Python to solve and plot the results of the following differential equation:

$$(1+t^2)\ddot{w} + 2t\dot{w} + 3w = 2$$

Note! Don't be confused that in this example w is used and not x or y. All these are just parameters or variable names.

Note! $\dot{w} = \frac{dw}{dt}$ and $\ddot{w} = \frac{d^2w}{dt^2}$

We will solve the differential equation in the interval [0,5s].

We will use the following initial conditions: $w(t_0) = 0$ and $\dot{w}(t_0) = 1$

First, we should rewrite the equation in order to get the highest derivative alone on the left side of the equation:

$$\ddot{w} = \frac{2 - 2t\dot{w} - 3w}{1+t^2}$$

Note! Higher order differential equations must be reformulated into a system of first order differential equations.

We do the following "trick":

$$w = x_1$$

$$\dot{w} = x_2$$

This gives a set of 1.order differential equations:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = \frac{2 - 2tx_2 - 3x_1}{1 + t^2}$$

Python code

```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Initialization
tstart = 0
tstop = 5
increment = 0.1

# Initial condition
x_init = [0,1]


t = np.arange(tstart,tstop+1,increment)


# Function that returns dx/dt
def mydiff(x, t):
    dx1dt = x[1]
    dx2dt = (2 - t*x[1] - 3*x[0])/(1 + t**2)

    dxdt = [dx1dt, dx2dt]
    return dxdt


# Solve ODE
x = odeint(mydiff, x_init, t)
print(x)

x1 = x[:,0]
x2 = x[:,1]


# Plot the Results
plt.plot(t,x1)
plt.plot(t,x2)
plt.title('Plotting Differential Equations Solution')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid()
plt.axis([0, 5, -1, 2])
plt.show()
```

This gives the plot shown in Figure .3.

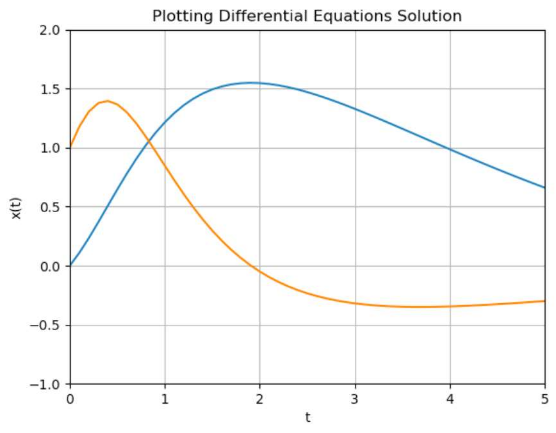


Figure 3

Integration

In Python, you can perform integration on a variable using either symbolic computation with the sympy library or **numerical integration** with the **scipy.integrate** library.

1. Symbolic Integration (Indefinite and Definite)

Use the **sympy** library when you want the exact, analytical result of an integral, possibly as a new mathematical expression involving variables.

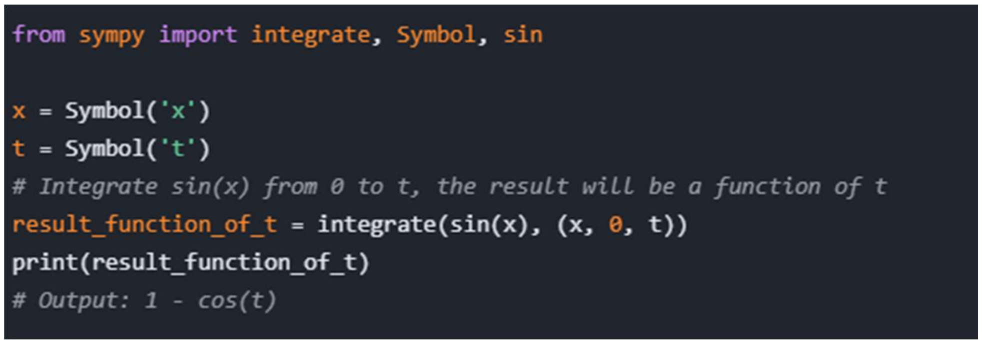
Indefinite Integral

```
from sympy import integrate, Symbol

x = Symbol('x')
f = x**2 + x + 1
indefinite_integral = integrate(f, x)
print(indefinite_integral)
# Output: x**3/3 + x**2/2 + x
```

Definite Integral (with constant limits)

```
from sympy import integrate, Symbol, exp

x = Symbol('x')
f = x * exp(x)
definite_integral = integrate(f, (x, 0, 1))
print(definite_integral)
# Output: 1
```

Definite Integral (with variable limits)

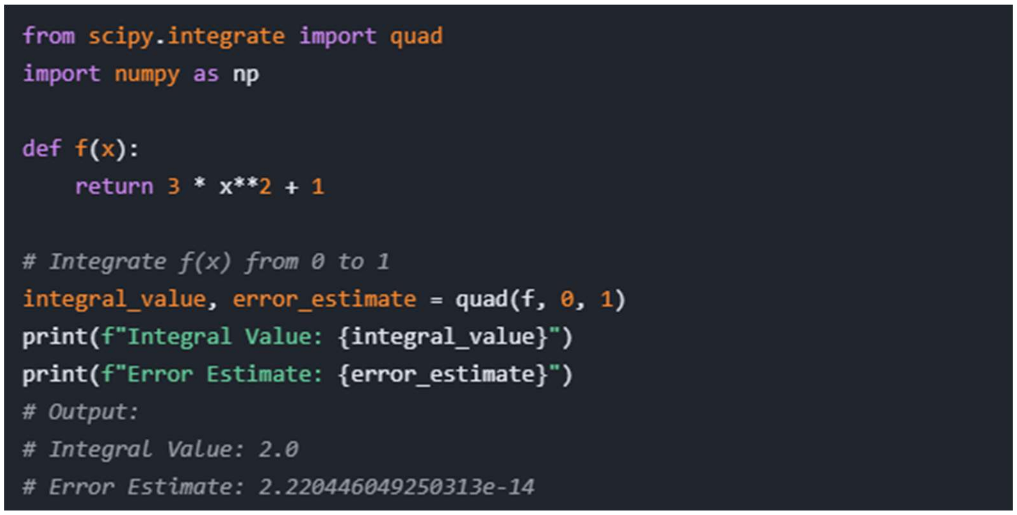


```python
from sympy import integrate, Symbol, sin

x = Symbol('x')
t = Symbol('t')
# Integrate sin(x) from 0 to t, the result will be a function of t
result_function_of_t = integrate(sin(x), (x, 0, t))
print(result_function_of_t)
# Output: 1 - cos(t)
```

2. Numerical Integration (Definite Only)

Use the **scipy.integrate** library when you have a function and need a numerical approximation of its definite integral, or when dealing with discrete data points.

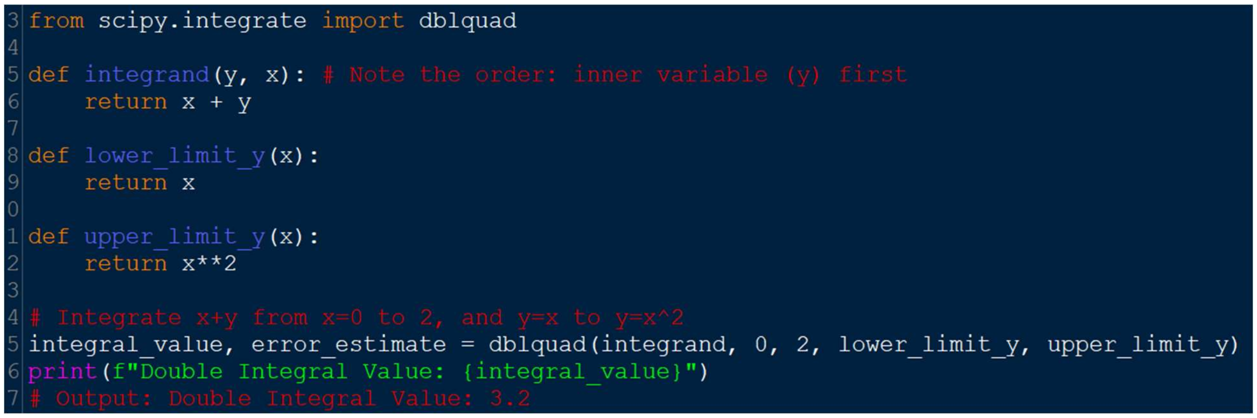
- Single Variable Function (quad): The quad function returns the integral value and an estimate of the absolute error.



```python
from scipy.integrate import quad
import numpy as np

def f(x):
    return 3 * x**2 + 1

# Integrate f(x) from 0 to 1
integral_value, error_estimate = quad(f, 0, 1)
print(f"Integral Value: {integral_value}")
print(f"Error Estimate: {error_estimate}")
# Output:
# Integral Value: 2.0
# Error Estimate: 2.220446049250313e-14
```

Multi-Variable Function with Variable Limits (dblquad):

You can integrate functions of multiple variables by defining the limits for inner integrals as functions of outer variables.



```python
from scipy.integrate import dblquad

def integrand(y, x): # Note the order: inner variable (y) first
    return x + y

def lower_limit_y(x):
    return x

def upper_limit_y(x):
    return x**2

# Integrate x+y from x=0 to 2, and y=x to y=x^2
integral_value, error_estimate = dblquad(integrand, 0, 2, lower_limit_y, upper_limit_y)
print(f"Double Integral Value: {integral_value}")
# Output: Double Integral Value: 3.2
```

Annexe 1

Function	**Description**
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression

exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable

memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
staticmethod()	Converts a method into a static method
str()	Returns a string object
sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class

tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators