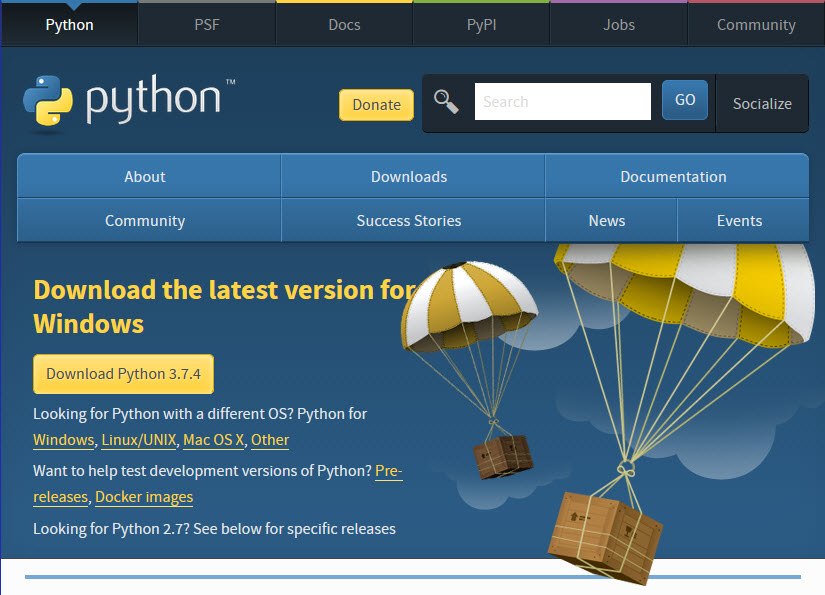
**PYTHON**

* **Introduction to Python** :-
* Python is developed by **Guido van Rossum**. Guido van Rossum started implementing Python in 1989. Python is a very simple programming language so even if you are new to programming, you can learn python without facing any issues.
* **How to Download and Install Python in Windows?**

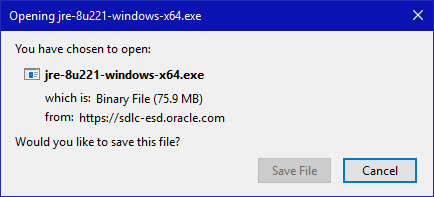
**Downloading**

1. Click [Python Download](https://www.python.org/downloads/). The following page will appear in your browser.s
2. Click the **Windows** link (two lines below the **Download Python 3.7.4** button). The following page will appear in your browser..



1. Click on the **Download Windows x86-64 executable installer** link under the top-left **Stable Releases**.

The following pop-up window titled **Opening python-3.74-amd64.exe** will appear.



Click the **Save File** button.

The file named **python-3.7.4-amd64.exe** should start downloading into your standard download folder. This file is about 30 Mb so it might take a while to download fully if you are on a slow internet connection (it took me about 10 seconds over a cable modem).

The file should appear as

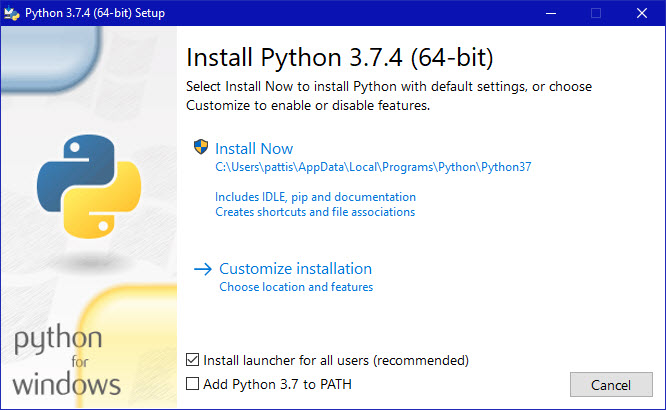
https://www.ics.uci.edu/~pattis/common/handouts/pythoneclipsejava/images/python/exefile.jpg

1. Move this file to a more permanent location, so that you can install Python (and reinstall it easily later, if necessary).
2. Feel free to explore this webpage further; if you want to just continue the installation, you can terminate the tab browsing this webpage.
3. Start the **Installing** instructions directly below.

**Installing**

1. Double-click the icon labeling the file **python-3.7.4-amd64.exe**.

A **Python 3.7.4 (64-bit) Setup** pop-up window will appear.



Ensure that the **Install launcher for all users (recommended)** and the **Add Python 3.7 to PATH** checkboxes at the bottom are checked.

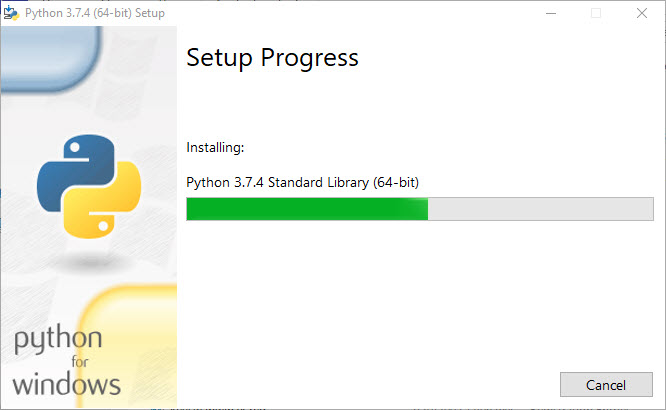
If the Python Installer finds an earlier version of Python installed on your computer, the **Install Now** message may instead appear as **Upgrade Now** (and the checkboxes will not appear).

1. Highlight the **Install Now** (or **Upgrade Now**) message, and then click it.

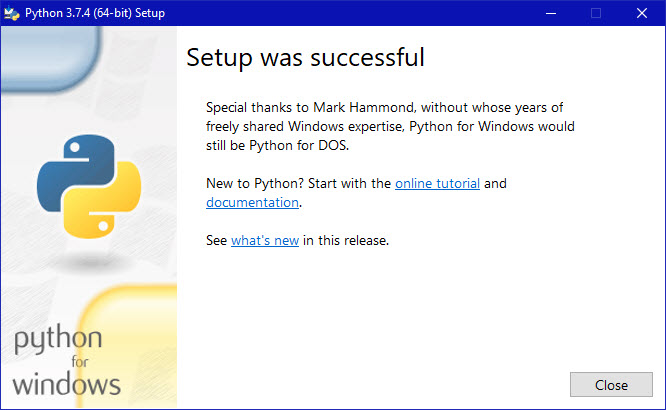
When run, a **User Account Control** pop-up window may appear on your screen. I could not capture its image, but it asks, **Do you want to allow this app to make changes to your device**.

1. Click the **Yes** button.

A new **Python 3.7.4 (64-bit) Setup** pop-up window will appear with a **Setup Progress** message and a progress bar.



During installation, it will show the various components it is installing and move the progress bar towards completion. Soon, a new **Python 3.7.4 (64-bit) Setup** pop-up window will appear with a **Setup was successfuly** message.



1. Click the **Close** button.

Python should now be installed.

What can Python do?

* Python can be used on a server to create web applications.
* Python can be used alongside software to create workflows.
* Python can connect to database systems. It can also read and modify files.
* Python can be used to handle big data and perform complex mathematics.
* Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

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

Python Syntax compared to other programming languages

* Python was designed for readability, and has some similarities to the English language with influence from mathematics.
* Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
* Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.
* **Python syntax –**

>>>print(“Hello, World!)

Hello, World!

This code is use to show the basic program in python.

* **Types of error –**
  + 1. **IndexError** is thrown when trying to access an item at an invalid index.

>>>L1=[1,2,3]  
>>>L1[3]  
Traceback(most recent call last):  
File "<pyshell#18>", line 1, in <module>  
L1[3]  
IndexError: list index out of range

* + 1. **ModuleNotFoundError** is thrown when a module could not be found.

>>> import notamodule  
Traceback (most recent call last):  
File "<pyshell#10>", line 1, in <module>  
import notamodule  
ModuleNotFoundError: No module named 'notamodule'

* + 1. **KeyError** is thrown when a key is not found.

>>> D1={'1':"aa", '2':"bb", '3':"cc"}  
>>> D1['4']  
Traceback (most recent call last):  
File "<pyshell#15>", line 1, in <module>  
D1['4']  
KeyError: '4'

* + 1. **TypeError** is thrown when an operation or function is applied to an object of an inappropriate type.

>>> '2'+2  
Traceback (most recent call last):  
File "<pyshell#23>", line 1, in <module>  
'2'+2  
TypeError: must be str, not int

* + 1. **NameError** is thrown when an object could not be found.

>>> age  
Traceback (most recent call last):  
File "<pyshell#6>", line 1, in <module>  
age  
NameError: name 'age' is not defined

* + 1. **ZeroDivisionError** is thrown when the second operator in the division is zero.

>>> x=100/0  
Traceback (most recent call last):  
File "<pyshell#8>", line 1, in <module>  
x=100/0  
ZeroDivisionError: division by zero

## **Indentation**

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

### Example

if 5 > 2:  
  print("Five is greater than two!")

Python will give you an error if you skip the indentation:

Example

Syntax Error:

if 5 > 2:  
print("Five is greater than two!")

**Reserved Words**

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

|  |  |  |
| --- | --- | --- |
| And | exec | Not |
| Assert | Finally | Or |
| Break | For | Pass |
| Class | From | Print |
| Continue | Global | Raise |
| Def | If | Return |
| Del | Import | Try |
| Elif | In | While |
| Else | Is | With |
| Except | Lambda | Yield |

* **VARIABLES**

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Every value in Python has a datatype. Different data types in Python are Numbers, List, Tuple, Strings, Dictionary, etc. Variables can be declared by any name or even alphabets like a, aa, abc, etc.

## **Creating Variables**

Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

x = 5  
y = "John"  
print(x)  
print(y)

# **Python Number**

## **Python Numbers**

There are three numeric types in Python:

* int
* float
* complex

Variables of numeric types are created when you assign a value to them:

### Example

x = 1     
y = 2.8   
z = 1j  

To verify the type of any object in Python, use the type() function:

### Example

print(type(x))  
print(type(y))  
print(type(z))

## **Int**

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

### Example

Integers:

x = 1  
y = 35656222554887711  
z = -3255522  
  
print(type(x))  
print(type(y))  
print(type(z))

## **Float**

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

### Example

Floats:

x = 1.10  
y = 1.0  
z = -35.59  
  
print(type(x))  
print(type(y))  
print(type(z))

Float can also be scientific numbers with an "e" to indicate the power of 10.

### Example

Floats:

x = 35e3  
y = 12E4  
z = -87.7e100  
  
print(type(x))  
print(type(y))  
print(type(z))

## **Complex**

Complex numbers are written with a "j" as the imaginary part:

### Example

Complex:

x = 3+5j  
y = 5j  
z = -5j  
  
print(type(x))  
print(type(y))  
print(type(z))

## **Type Conversion**

You can convert from one type to another with the int(), float(), and complex() methods:

### Example

Convert from one type to another:

x = 1   
y = 2.8   
z = 1j    
  
a = float(x)  
  
b = int(y)  
  
c = complex(x)  
  
print(a)  
print(b)  
print(c)

# **Python Strings**

A sequence of one or more characters enclosed within either single quotes ‘ or double quotes ” is considered as String in Python. Any letter, a number or a symbol could be a part of the string.

You can display a string literal with the print() function:

Example

print("Hello")  
print('Hello')

Python also supports multi-line strings which require a triple quotation mark at the start and one at the end.

You can assign a multiline string to a variable by using three quotes:

### Example

You can use three double quotes:

a = """Hello GOOD MORNING."""  
print(a)

Or three single quotes:

### Example

a = '''Hello GOOD MORNING.'''  
print(a)

## **Assign String to a Variable**

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

### Example

a = "Hello"  
print(a)

# **Booleans**

The Boolean data type can be one of two values, either **True** or **False**.

The expression is evaluated and Python returns the Boolean answer:

Example

print(10 > 9)  
print(10 == 9)  
print(10 < 9)

When you run a condition in an if statement, Python returns True or False:

### Example

Print a message based on whether the condition is True or False:

a = 200  
b = 33  
  
if b > a:  
  print("b is greater than a")  
else:  
  print("b is not greater than a")

## **Operators**

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

>>> 2+3

5

Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.

Python divides the operators in the following groups:

* Arithmetic operators
* Relational operators
* Logical operators
* Bitwise operators

1.  **Arithmetic operators:** Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

|  |  |  |
| --- | --- | --- |
| **OPERATOR** | **DESCRIPTION** | **SYNTAX** |
| + | Addition: adds two operands | x + y |
| - | Subtraction: subtracts two operands | x - y |
| \* | Multiplication: multiplies two operands | x \* y |
| / | Division (float): divides the first operand by the second | x / y |
| % | Modulus: returns the remainder when first operand is divided by the second | x % y |
|  |  |  |

1. **Relational Operators:**Relational operators compares the values. It either returns **True** or **False** according to the condition.

|  |  |  |
| --- | --- | --- |
| **OPERATOR** | **DESCRIPTION** | **SYNTAX** |
| > | Greater than: True if left operand is greater than the right | x > y |
| < | Less than: True if left operand is less than the right | x < y |
| == | Equal to: True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to: True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to: True if left operand is less than or equal to the right | x <= y |

**3. Logical operators:**Logical operators perform **Logical AND**, **Logical OR** and**Logical NOT** operations.

|  |  |  |
| --- | --- | --- |
| **OPERATOR** | **DESCRIPTION** | **SYNTAX** |
| And | Logical AND: True if both the operands are true | x and y |
| Or | Logical OR: True if either of the operands is true | x or y |
| Not | Logical NOT: True if operand is false | not x |

**4.Bitwise operators:**Bitwise operators acts on bits and performs bit by bit operation.

|  |  |  |
| --- | --- | --- |
| **OPERATOR** | **DESCRIPTION** | **SYNTAX** |
| & | Bitwise AND | x & y |
| | | Bitwise OR | x | y |
| ~ | Bitwise NOT | ~x |
| ^ | Bitwise XOR | x ^ y |
| >> | Bitwise right shift | x>> |
| << | Bitwise left shift | x<< |

# **Lists**

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Example

Create a List:

list = ["apple", "banana", "cherry"]  
print(list)

## **Access Items**

You access the list items by referring to the index number:

### Example

Print the second item of the list:

list = ["apple", "banana", "cherry"]  
print(list[1])

Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

Example

Print the last item of the list:

list = ["apple", "banana", "cherry"]  
print(list[-1])

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(list[2:5])

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

### Example

This example returns the items from the beginning to "orange":

list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(list[:4])

By leaving out the end value, the range will go on to the end of the list:

### Example

This example returns the items from "cherry" and to the end:

list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(list[2:])

### Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list:

### Example

This example returns the items from index -4 (included) to index -1 (excluded)

list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(list[-4:-1])

## **Change Item Value**

To change the value of a specific item, refer to the index number:

### Example

Change the second item:

list = ["apple", "banana", "cherry"]  
list[1] = "blackcurrant"  
print(list)

## **Loop Through a List**

You can loop through the list items by using a for loop:

### Example

Print all items in the list, one by one:

list = ["apple", "banana", "cherry"]  
for x in list:  
  print(x)

## **Check if Item Exists**

To determine if a specified item is present in a list use the in keyword:

### Example

Check if "apple" is present in the list:

list = ["apple", "banana", "cherry"]  
if "apple" in list:  
  print("Yes, 'apple' is in the fruits list")

## **List Length**

To determine how many items a list has, use the len() function:

### Example

Print the number of items in the list:

list = ["apple", "banana", "cherry"]  
print(len(list))

## **Add Items**

To add an item to the end of the list, use the append() method:

### Example

Using the append() method to append an item:

list = ["apple", "banana", "cherry"]  
list.append("orange")  
print(list)

To add an item at the specified index, use the insert() method:

### Example

Insert an item as the second position:

list = ["apple", "banana", "cherry"]  
list.insert(1, "orange")  
print(list)

## **Remove Item**

There are several methods to remove items from a list:

### Example

The remove() method removes the specified item:

list = ["apple", "banana", "cherry"]  
list.remove("banana")  
print(list)

### Example

The pop() method removes the specified index, (or the last item if index is not specified):

list = ["apple", "banana", "cherry"]  
list.pop()  
print(list)

### Example

The del keyword removes the specified index:

list = ["apple", "banana", "cherry"]  
del list[0]  
print(list)

### Example

The del keyword can also delete the list completely:

list = ["apple", "banana", "cherry"]  
del list

### Example

The clear() method empties the list:

list = ["apple", "banana", "cherry"]  
list.clear()  
print(list)

## **Copy a List**

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

### Example

Make a copy of a list with the copy() method:

list = ["apple", "banana", "cherry"]  
mylist = list.copy()  
print(mylist)

Another way to make a copy is to use the built-in method list().

### Example

Make a copy of a list with the list() method:

list = ["apple", "banana", "cherry"]  
mylist = list(list)  
print(mylist)

**Join Two Lists**

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

### Example

Join two list:

list1 = ["a", "b", "c"]  
list2 = [1, 2, 3]  
  
list3 = list1 + list2  
print(list3)

Another way to join two lists are by appending all the items from list2 into list1, one by one:

### Example

Append list2 into list1:

list1 = ["a", "b" , "c"]  
list2 = [1, 2,  3]  
  
for x in list2:  
  list1.append(x)  
  
print(list1)

Or you can use the extend() method, which purpose is to add elements from one list to another list:

### Example

Use the extend() method to add list2 at the end of list1:

list1 = ["a", "b", "c"]  
list2 = [1, 2, 3]  
  
list1.extend(list2)  
print(list1)

## **The list() Constructor**

It is also possible to use the list() constructor to make a new list.

### Example

Using the list() constructor to make a List:

list = list(("apple", "banana", "cherry"))  
print(list)

## **List Methods**

Python has a set of built-in methods that you can use on lists.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [append()](https://www.w3schools.com/python/ref_list_append.asp) | Adds an element at the end of the list |
| [clear()](https://www.w3schools.com/python/ref_list_clear.asp) | Removes all the elements from the list |
| [copy()](https://www.w3schools.com/python/ref_list_copy.asp) | Returns a copy of the list |
| [count()](https://www.w3schools.com/python/ref_list_count.asp) | Returns the number of elements with the specified value |
| [extend()](https://www.w3schools.com/python/ref_list_extend.asp) | Add the elements of a list (or any iterable), to the end of the current list |
| [index()](https://www.w3schools.com/python/ref_list_index.asp) | Returns the index of the first element with the specified value |
| [insert()](https://www.w3schools.com/python/ref_list_insert.asp) | Adds an element at the specified position |
| [pop()](https://www.w3schools.com/python/ref_list_pop.asp) | Removes the element at the specified position |
| [remove()](https://www.w3schools.com/python/ref_list_remove.asp) | Removes the item with the specified value |
| [reverse()](https://www.w3schools.com/python/ref_list_reverse.asp) | Reverses the order of the list |
| [sort()](https://www.w3schools.com/python/ref_list_sort.asp) | Sorts the list |

## **Tuple**

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

### Example

Create a Tuple:

tuple = ("apple", "banana", "cherry")  
print(tuple)

## **Access Tuple Items**

You can access tuple items by referring to the index number, inside square brackets:

### Example

Print the second item in the tuple:

tuple = ("apple", "banana", "cherry")  
print(tuple[1])

### Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

### Example

Print the last item of the tuple:

tuple = ("apple", "banana", "cherry")  
print(tuple[-1])

### Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

### Example

Return the third, fourth, and fifth item:

tuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(tuple[2:5])

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

### Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

### Example

This example returns the items from index -4 (included) to index -1 (excluded)

tuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(tuple[-4:-1]

## **Change Tuple Values**

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

### Example

Convert the tuple into a list to be able to change it:

x = ("apple", "banana", "cherry")  
y = list(x)  
y[1] = "kiwi"  
x = tuple(y)  
  
print(x)

## **Loop Through a Tuple**

You can loop through the tuple items by using a for loop.

### Example

Iterate through the items and print the values:

tuple = ("apple", "banana", "cherry")  
for x in tuple:  
  print(x)

## **Check if Item Exists**

To determine if a specified item is present in a tuple use the in keyword:

### Example

Check if "apple" is present in the tuple:

tuple = ("apple", "banana", "cherry")  
if "apple" in tuple:  
  print("Yes, 'apple' is in the fruits tuple")

## **Tuple Length**

To determine how many items a tuple has, use the len() method:

### Example

Print the number of items in the tuple:

tuple = ("apple", "banana", "cherry")  
print(len(tuple))

## **Add Items**

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

### Example

You cannot add items to a tuple:

tuple = ("apple", "banana", "cherry")  
tuple[3] = "orange"  
print(tuple)

## **Create Tuple With One Item**

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

### Example

One item tuple, remember the comma:

tuple = ("apple",)  
print(type(tuple))  
  
tuple = ("apple")  
print(type(tuple))

**Remove Items**

**Note:** You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

### Example

The del keyword can delete the tuple completely:

tuple = ("apple", "banana", "cherry")  
del tuple  
print(tuple)

## **Join Two Tuples**

To join two or more tuples you can use the + operator:

### Example

Join two tuples:

tuple1 = ("a", "b" , "c")  
tuple2 = (1, 2, 3)  
  
tuple3 = tuple1 + tuple2  
print(tuple3)

## **The tuple() Constructor**

It is also possible to use the tuple() constructor to make a tuple.

### Example

Using the tuple() method to make a tuple:

tuple = tuple(("apple", "banana", "cherry"))  
print(tuple)

## **Tuple Methods**

Python has two built-in methods that you can use on tuples.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [count()](https://www.w3schools.com/python/ref_tuple_count.asp) | Returns the number of times a specified value occurs in a tuple |
| [index()](https://www.w3schools.com/python/ref_tuple_index.asp) | Searches the tuple for a specified value and returns the position of it was found where i was found |

## **Dictionary**

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

### Example

Create and print a dictionary:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
print(dict)  
**Accessing Items**

You can access the items of a dictionary by referring to its key name, inside square brackets:

### Example

Get the value of the "model" key:

x = dict["model"]  
There is also a method called get() that will give you the same result:

### Example

Get the value of the "model" key:

x = dict.get("model")

## **Change Values**

You can change the value of a specific item by referring to its key name:

### Example

Change the "year" to 2018:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
dict["year"] = 2018

## **Loop Through a Dictionary**

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

### Example

Print all key names in the dictionary, one by one:

for x in dict:  
  print(x)

### Example

Print all values in the dictionary, one by one:

for x in dict:  
  print(dict[x])

### Example

You can also use the values() method to return values of a dictionary:

for x in thisdict.values():  
  print(x)

**Example**

Loop through both keys and values, by using the items() method:

for x, y in dict.items():  
  print(x, y)

## **Check if Key Exists**

To determine if a specified key is present in a dictionary use the in keyword:

### Example

Check if "model" is present in the dictionary:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
if "model" in dict:  
  print("Yes, 'model' is one of the keys in the dict dictionary")

## **Dictionary Length**

To determine how many items (key-value pairs) a dictionary has, use the len() function.

### Example

Print the number of items in the dictionary:

print(len(dict))

## **Adding Items**

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

### Example

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
dict["color"] = "red"  
print(dict)

## **Removing Items**

There are several methods to remove items from a dictionary:

### Example

The pop() method removes the item with the specified key name:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
dict.pop("model")  
print(dict)

### Example

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
dict.popitem()  
print(dict)

### Example

The del keyword removes the item with the specified key name:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
del dict["model"]  
print(dict)

### Example

The del keyword can also delete the dictionary completely:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
del dict  
print(dict)

### Example

The clear() method empties the dictionary:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
dict.clear()  
print(dict)

## **Copy a Dictionary**

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

### Example

Make a copy of a dictionary with the copy() method:

sdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
mydict = dict.copy()  
print(mydict)

Another way to make a copy is to use the built-in function dict().

### Example

Make a copy of a dictionary with the dict() function:

dict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
mydict = dict(dict)  
print(mydict)

## **Nested Dictionaries**

A dictionary can also contain many dictionaries, this is called nested dictionaries.

### Example

Create a dictionary that contain three dictionaries:

myfamily = {  
  "child1" : {  
    "name" : "Emil",  
    "year" : 2004  
  },  
  "child2" : {  
    "name" : "Tobias",  
    "year" : 2007  
  },  
  "child3" : {  
    "name" : "Linus",  
    "year" : 2011  
  }  
}

Or, if you want to nest three dictionaries that already exists as dictionaries:

### Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

child1 = {  
  "name" : "Emil",  
  "year" : 2004  
}  
child2 = {  
  "name" : "Tobias",  
  "year" : 2007  
}  
child3 = {  
  "name" : "Linus",  
  "year" : 2011  
}  
  
myfamily = {  
  "child1" : child1,  
  "child2" : child2,  
  "child3" : child3  
}

## **The dict() Constructor**

It is also possible to use the dict() constructor to make a new dictionary:

### Example

dict = dict(brand="Ford", model="Mustang", year=1964)  
print(dict)

## **Python Conditions and If statements**

Python supports the usual logical conditions from mathematics:

* Equals: a == b
* Not Equals: a != b
* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

### Example

If statement:

a = 33  
b = 200  
if b > a:  
  print("b is greater than a")

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

## **Indentation**

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

### Example

If statement, without indentation (will raise an error):

a = 33  
b = 200  
if b > a:  
print("b is greater than a")

## **Elif**

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

### Example

a = 33  
b = 33  
if b > a:  
  print("b is greater than a")  
elif a == b:  
  print("a and b are equal")

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

## **Else**

The else keyword catches anything which isn't caught by the preceding conditions.

### Example

a = 200  
b = 33  
if b > a:  
  print("b is greater than a")  
elif a == b:  
  print("a and b are equal")  
else:  
  print("a is greater than b")

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

### Example

a = 200  
b = 33  
if b > a:  
  print("b is greater than a")  
else:  
  print("b is not greater than a")

## **Short Hand If**

If you have only one statement to execute, you can put it on the same line as the if statement.

### Example

One line if statement:

if a > b: print("a is greater than b")

## **Short Hand If ... Else**

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

### Example

One line if else statement:

a = 2  
b = 330  
print("A") if a > b else print("B")

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

### Example

One line if else statement, with 3 conditions:

a = 330  
b = 330  
print("A") if a > b else print("=") if a == b else print("B")

## **And**

The and keyword is a logical operator, and is used to combine conditional statements:

### Example

Test if a is greater than b, AND if c is greater than a:

a = 200  
b = 33  
c = 500  
if a > b and c > a:  
  print("Both conditions are True")

## **Or**

The or keyword is a logical operator, and is used to combine conditional statements:

### Example

Test if a is greater than b, OR if a is greater than c:

a = 200  
b = 33  
c = 500  
if a > b or a > c:  
  print("At least one of the conditions is True")

## **Nested If**

You can have if statements inside if statements, this is called nested if statements.

### Example

x = 41  
  
if x > 10:  
  print("Above ten,")  
  if x > 20:  
    print("and also above 20!")  
  else:  
    print("but not above 20.")

## **The pass Statement**

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

### Example

a = 33  
b = 200  
  
if b > a:  
  pass

## **Python Loops**

Python has two primitive loop commands:

* while loops
* for loops

## **The while Loop**

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is −

while expression:

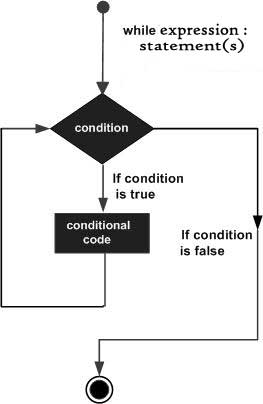
statement(s)

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

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

Flow Diagram



Example

Print i as long as i is less than 6:

i = 1  
while i < 6:  
  print(i)  
  i += 1

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

## **The break Statement**

With the break statement we can stop the loop even if the while condition is true:

### Example

Exit the loop when i is 3:

i = 1  
while i < 6:  
  print(i)  
  if i == 3:  
    break  
  i += 1

## **The continue Statement**

With the continue statement we can stop the current iteration, and continue with the next:

### Example

Continue to the next iteration if i is 3:

i = 0  
while i < 6:  
  i += 1  
  if i == 3:  
    continue  
  print(i)

**The else Statement**

With the else statement we can run a block of code once when the condition no longer is true:

### Example

Print a message once the condition is false:

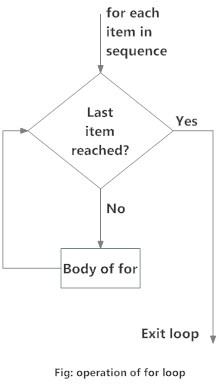
i = 1  
while i < 6:  
  print(i)  
  i += 1  
else:  
  print("i is no longer less than 6")

## **Python For Loops**

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.



Example

Print each fruit in a fruit list:

fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
  print(x)

### FUNCTIONS

A function is a group of related statements that performs a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes the code reusable.

The main advantage of functions is code Reusability.

Python supports 2 types of functions

* + 1. Built-in Functions
    2. User Defined Functions
* **BUILT-IN FUNCTIONS**

The functions which are coming along with Python software automatically, are called built-in functions or pre-defined functions.

Example :-

* + id()
  + type()
  + input()
  + eval() etc..
* **USER-DEFINED FUNCTIONS**

The functions which are developed by programmer explicitly according to business requirements, are called user defined functions.

Syntax to create user defined functions:

def function\_name(parameters):

“ ” ”doc string” “ “

* + - - -
  + - - -

return value

**NOTE-** While creating functions we can use 2 keywords – def (mandatory) & return (optional)

|  |
| --- |
| Example – Write a function to print HELLOO. |
| def greet(): |
| print(“HELLOO”) |
| greet() |

### PARAMETERS

Parameters are inputs to the function. If a function contains parameters, then at the time of calling, compulsory we should provide values otherwise we will get an error.

Example (1) – Write a function to take names of the student as input and print message by name.

|  |
| --- |
| def message(name): |
| print(“Hello”,name,”How Are You?”) |
| message(“Ruchika”) |
| message(“Mansi”) |

Example (2) – Write a function to take number as input and print its square value.

|  |
| --- |
| def squareIt(number): |
| print(“The Square of”,number,”is”,number\*number) |
| squareIt(4) |
| squareIt(5) |

**RETURN STATEMENT:**

Function can take input values as parameters and executes business logic, and returns output to the caller with return statement.

**Example – Write a function to accept 2 numbers as input and return sum**

|  |
| --- |
| def add(x,y): |
| return x+y |
| result=add(10,20) |
| print(“The sum is”,result) |
| print(“The sum is”,add(100,200)) |

If we are not writing return statement then default return value is None

Example (2):

|  |
| --- |
| def f1(): |
| print(“Hello”) |
| f1() |
| print(f1()) |

**Example: Write a function to check whether the given number is even or odd?**

|  |
| --- |
| def even\_odd(num): |
| if num%2==0: |
| print(num,”is Even Number”) |
| else: |
| print(num,”is Odd Number”) |
| even\_odd(10) |
| even\_odd(15) |

**Example: Write a function to find factorial of given number?**

|  |
| --- |
| def fact(num): |
| result=1 |
| while num>=1: |
| result=result\*num |
| num=num-1 |
| return result |
| for i in range(1,5): |
| print(“The Factorial of”,i,”is:”,fact(i)) |

**Returning multiple values from a function.**

In python, a function can return any number of values.

**Example 1:**

|  |
| --- |
| def sum\_sub(a,b): |
| sum=a+b |
| sub=a-b |
| return sum,sub |
| x,y=sum\_sub(100,50) |
| print(“The Sum is:”,x) |
| Print(“The Subtraction is:”,y) |

**Example 2:**

|  |
| --- |
| def calc(a,b): |
| sum=a+b |
| sub=a-b |
| mul=a\*b |
| div=a/b |
| return sum,sub,mul,div |
| t=calc(100,50) |
| print(“The Results are”) |
| for i in t: |
| print(i) |

**Types of arguments**

def f1(a,b):

-----

-----

------

f1(10,20)

a,b are formal arguments where as 10,20 are actual arguments.

There are 4 types of actual arguments are allowed in Python.

1. **Positional arguments**
2. **Keyword arguments**
3. **Default arguments**
4. **Variable length arguments**
   * 1. **Positional arguments:**

These are the arguments passed to function in correct positional order.

def sub(a,b)

print(a-b)

sub(100,200)

sub(200,100)

The number of arguments and position of arguments must be matched. If we change the order then result may be changed.

If we change the number of arguments then we will get error.

**2. Keyword arguments:**

We can pass argument values by keyword i.e by parameter name.

Example:

|  |
| --- |
| def wish(name,msg): |
| print(“Hello”,name,msg) |
| wish(name=”Ruchika”,msg=”I’m good. How are you?”) |
| wish(msg=”I’m good. How are you?”, name=”Ruchika”) |

Here the order of arguments is not important but number of arguments must be matched.

NOTE:

We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments, otherwise we will get syntax error.

def wish(name,msg):

print(“Hello”,name,msg)

wish(“Ruchika”,”I’m good. How are you?”) 🡺 valid

wish(“Ruchika”,msg=”I’m good. How are you?”) 🡺valid

wish(name=”Ruchika”,”I’m good. How are you?”)🡺invalid

Syntax Error: Positional argument follows keyword argument.

1. **Default Arguments:**

Sometimes we can provide default values for our positional arguments.

Example:

|  |
| --- |
| def wish(name=”Mansi”): |
| print(“Hello”,name,”Good Morning”) |
|  |
| wish(“Ruchika”) |
| wish() |

If we are not passing any name then only default value will be considered.

\*\*\*NOTE:

After default arguments we should not take non default arguments.

def wish(name=“Mansi”,msg=”Good Morning”): 🡺 valid

def wish(name,msg=”Good Morning”): 🡺 valid

def wish(name=”Mansi”,msg): 🡺invalid

Syntax Error: non-default arguments follows default argument.

1. **Variable length arguments:**

Sometimes we can pass variable number of arguments to our function, such type of arguments are called variable length arguments.

We can declare a variable length argument with \* symbol as follows.

def f1(\*n):

We can call this function by passing any number of arguments including zero number. Internally all these values represented in the form of tuple.

Example:

|  |
| --- |
| def sum(\*n): |
| total=0 |
| for n1 in n: |
| total=total+n1 |
| print(“The Sum=”,total) |
|  |
| sum() |
| sum(10) |
| sum(10,20) |
| sum(10,20,30,40) |

Example (2):

|  |
| --- |
| def f1(n1,\*s): |
| print(n1) |
| for s1 in s: |
| print(s1) |
|  |
| f1(10) |
| f1(10,20,30,40) |
| f1(10,”A”,30,”B”) |

* + **Note: Function v/s Module v/s Library:**
    1. A group of lines with some name is called a function.
    2. A group of functions saved to a file, is called Module.
    3. A group of Modules is nothing but library.

**Types of Variables:**

Python supports 2 types of variables:

* + - 1. Global variable
      2. Local variable
         1. **Global variable**

The variables which are declared outside of function are called global variable. These variable can be accessed in all functions of that module.

Example :

|  |
| --- |
| a=10 #global variable |
| def f1(): |
| print(a) |
|  |
| def f2(): |
| print(a) |
|  |
| f1() |
| f2() |

**2. Local Variable**

The variables which are declared inside a function are called local variables. Local variables are available only for the function in which we declared it i.e from outside of function we cannot access.

Example :

|  |
| --- |
| def f1(): |
| a=10 |
| print(a) #valid |
|  |
| def f2() |
| print(a) #invalid |
|  |
| f1() |
| f2() |

**Global keyword:**

We can use global keyword for the following 2 purposes:

* + 1. To declare global variable inside function.
    2. To make global variable available to the function so that we can perform required modifications.

Example (1):

|  |
| --- |
| a=10 |
| def f1(): |
| a=777 |
| print(a) |
|  |
| def f2(): |
| print(a): |
|  |
| f1() |
| f2() |

Example (2):

|  |
| --- |
| def f1(): |
| a=10 |
| print(a) |
|  |
| def f2(): |
| print(a) |
|  |
| f1() |
| f2() |

NOTE: If global variable and local variable having the same name then we can access global variable inside a function as follows

|  |
| --- |
| a=10 #global variable |
| def f1(): |
| a=777 #local variable |
| print(a) |
| print(globals()[‘a’]) |
| f1() |

**Recursive Function:**

A function that calls itself is known as Recursive Function.

Example:

factorial(3)=3\*factorial(2)

=3\*2\*factorial(1)

=3\*2\*1\*factorial(0)

=3\*2\*1\*1

=6

factorial(n)=n\*factorial(n-1)

The main advantages of recursive functions are:

* + - 1. We can reduce length of the code and improves readability.
      2. We can solve complex problems very easily.

**Question. Write a Python Function to find factorial of given number with recursion.**

|  |
| --- |
| def factorial(n): |
| if n==0: |
| result=1 |
| else: |
| result=n\*factorial(n-1) |
| return result |
| print(“Factorial of 4 is :”,factorial(4)) |
| print(“Factorial of 5 is :”,factorial(5)) |

**Anonymous Functions:**

Sometimes we can declare a function without any name, such type of nameless functions are called anonymous functions or lambda function.

The main purpose of anonymous function is just for instant use(i.e for one time usage).

**Normal Function:**

We can define by using def keyword.

def squareIt(n):

return n\*n

**Lambda Function:**

We can define by using lambda keyword

Lambda n:n\*n

**Syntax of lambda function:**

lambda argument\_list : expression

NOTE: By using Lambda Functions we can write very concise code so that readability of the program will be improved.

**Question. Write a program to create a lambda function to find square of given number?**

|  |
| --- |
| s=lambda n:n\*n |
| print(“The square of 4 is:”, s(4)) |
| print(“The square of 5 is:”,s(5)) |

**Question. Lambda function to find sum of 2 given numbers.**

|  |
| --- |
| s=lambda a,b:a+b |
| print(“The sum of 10,20 is:”,s(10,20)) |
| print(“The sum of 100,200 s:”,s(100,200)) |

**Question. Lambda function to find biggest of given values.**

|  |
| --- |
| s=lambda a,b:a if a>b else b |
| print(“The biggest of 10,20 is:” ,s(10,20)) |
| print(“The biggest of 100,200 is:” ,s(100,200)) |

NOTE:

Lambda function internally returns expression value and we are not required to write return statement explicitly.

NOTE: Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(), map() and reduce() functions.

**filter() function:**

We can use filter() function to filter values from the given sequence based on some condition.

filter(function,sequence)

where function argument is responsible to perform conditional check sequence can be list or tuple or string.

**Question. Program to filter only even numbers from the list by using filter() function?**

**Without lambda function:**

|  |
| --- |
| def isEven(x): |
| if x%2==0: |
| return True |
| else: |
| return False |
| l=[0,5,10,15,20,25,30] |
| l1=list(filter(isEven,l) |
| print(l1) #[0,10,20,30] |

**With lambda function:**

|  |
| --- |
| l=[0,5,10,15,20,25,30] |
| l1=list(filter(lambda x:x%2==0,l)) |
| print(l1) #[0,10,20,30] |
| l2=list(filter(lambda x:x%2!=0,l)) |
| print(l2) #[5,15,25] |

**map() function:**

For every element present in the given sequence, apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.

Example: for every element present in the list perform double and generate new list of doubles.

**Syntax:**

map(function,sequence)

The function can be applied on each element of sequence and generates new sequence.

**Eg. Without lambda**

|  |
| --- |
| l=[1,2,3,4,5] |
| def doubleIt(x): |
| return 2\*x |
| l1=list(map(doublelt,l)) |
| print(l1) #[2,4,6,8,10] |

**With lambda**

|  |
| --- |
| l=[1,2,3,4,5] |
| l1=list(map{lambda x:2\*x,l)) |
| print(l1) #[2,4,6,8,10] |

**Eg (2). To find square of given numbers.**

|  |
| --- |
| l=[1,2,3,4,5] |
| l1=list(map(lambda x:x\*x,l)) |
| print(l1) #[1,4,9,16,25] |

We can apply map() function on multiple lists also. But make sure all list should have same length.

Syntax: map(lambda x,y:x\*y,l1,l2))

x is from l1 and y is from l2

Example:

|  |
| --- |
| l1=[1,2,3,4] |
| l2=[2,3,4,5] |
| l3=list(map(lambda x,y:x\*y,l1,l2)) |
| print(l3) #[2,6,12,20] |

**reduce() function:**

reduce() function reduces sequences of elements into a single element by applying the specified function.

reduce(function,sequence)

reduce() function present in functools modules and hence we should write import statement.

**Example:**

|  |
| --- |
| from functools import\* |
| l=[10,20,30,40,50] |
| result=reduce(lambda x,y:x+y,l) |
| print(result) #150 |

**Example:**

|  |
| --- |
| result=reduce(lambda x,y:x\*y,l) |
| print(result) #12000000 |

**Example:**

|  |
| --- |
| from functools import\* |
| result=reduce(lambda x,y:x+y,range(1,101)) |
| print(result) #5050 |

NOTE:

* + In python everything is treated as object.
  + Even functions also internally treated as objects only.

**Example:**

|  |
| --- |
| def f1(): |
| print(“Hello”) |
| print(f1) |
| print(id(f1)) |

**Output**

**<function f1 at 0x00419618>**

**4298264**

**Question. What is the difference between the following lines? f1 = outer**

**f2 = outer()**

* In the first case for the outer() function we are providing another name f1(function aliasing).
* But in the second case we calling outer() function, which returns inner function. For that inner function() we are providing another name f1.

NOTE: We can pass function as argument to another function.

Example: filter(function, sequence)

map(function, sequence)

reduce(function, sequence)

**Function Decorators:**

Decorator is a function which can take a function as argument and extend its functionality and returns modified function with extended functionality.

**Input Function new(add some functionality)**

**Decorator**

**Wish() inner()**

**Input Function Decorator Function output function with**

**Extended functionality**

The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

|  |
| --- |
| def wish(name): |
| print(“Hello”,name,”Good Morning”) |

This function can always print same output for any name

Hello Mansi Good Morning

Hello Ruchika Good Morning

But we want to modify this function to provide different message if name is Mansi.

**Example:**

|  |
| --- |
| def decor(func): |
| def inner(name): |
| if name==”Mansi”: |
| print(“Hello Mansi Good Morning”) |
| else: |
| func(name) |
| return inner |
|  |
| def wish(name): |
| print(“Hello”,name,”Good Morning”) |
|  |
| wish(“Mansi”) |
| wish(“Ruchika”) |