inceptez

Python Function Based Programming and OOPS Theory & Hands on

Python is an interpreted, high-level, general-purpose **programming language**. It supports both Object Oriented and Function Based Programming.

Object Oriented Programming
PYTHON
Functional Programming

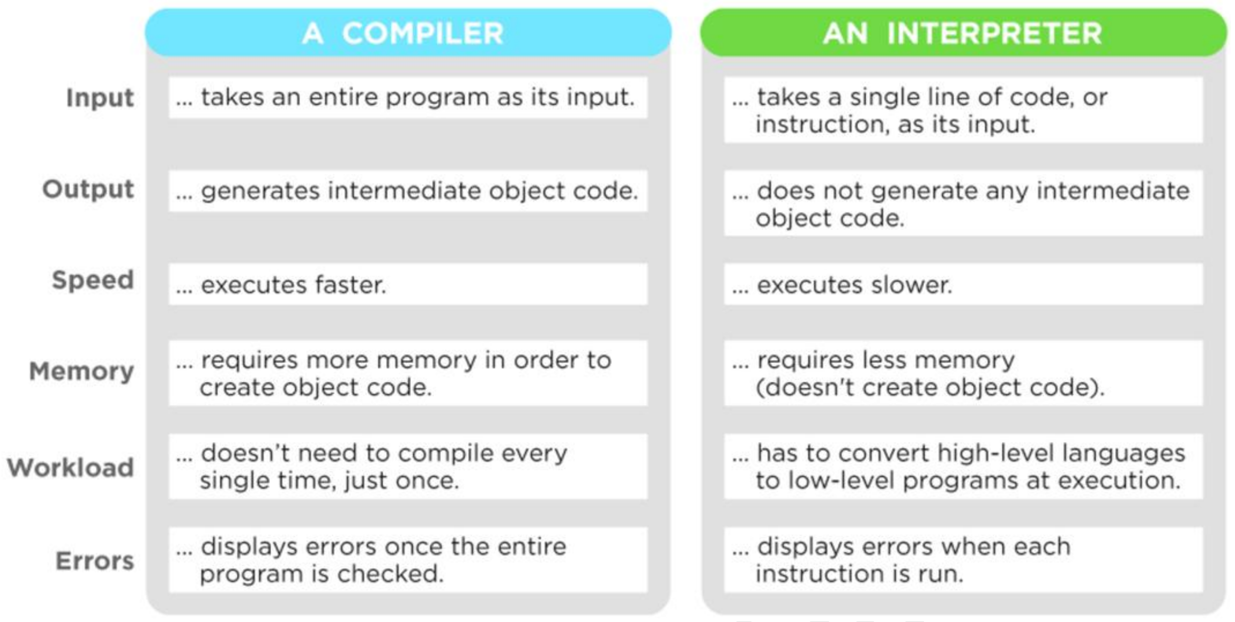
History

- ✓ Who? Invented in the Netherlands ***by Guido van Rossum***
- ✓ When? Python was conceived in the late 1980s and its implementation was started in December 1989
- ✓ How Named? Guido Van Rossum is fan of 'Monty Python's Flying Circus', this is a famous TV show in Netherlands, named after Monty Python
- ✓ Open source? From the beginning it was Open sourced

Why Python?

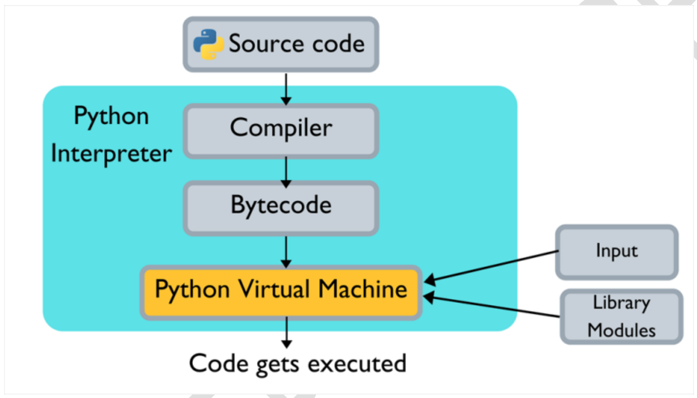
- ➢ Built-in types and Library utilities
- ➢ Dynamic typing
- ➢ Strongly typing
- ➢ Third party utilities (e.g. Pandas, NumPy, SciPy)
- ➢ Automatic memory management

Python is a Compiler or Interpreted Language?



	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

How Python code gets executed?



<u>Writing Python code using different tools</u>

Type	Tool Name	Description	Link
Text Editor	VS Code	Lightweight, extensible editor with Python extension and debugging.	code.visualstudio.com
	Sublime Text	Fast, minimal editor with syntax highlighting and plugin support.	sublimetext.com
IDE	PyCharm	Full-featured Python IDE (Community & Pro versions).	jetbrains.com/pycharm
	Thonny	Beginner-friendly IDE with built-in debugger and clean interface.	thonny.org
Browser-Based	Google Colab	Free cloud-hosted Jupyter notebooks with GPU/TPU support.	colab.research.google.com

	Jupyter (Try)	Online Jupyter environment, no install needed.	jupyter.org/try
Terminal / CLI	Python (REPL)	Default Python shell, installed with Python.	python.org
	IPython	Enhanced interactive Python shell with magic functions.	ipython.readthedocs.io

Step 1: Download Python 3.x Installer

Go to the official Python website:

https://www.python.org/downloads/release/python-3100/

Scroll down and select the Windows installer and Download:

Step 2: Run the Installer

Goto Downloads folder, Double-click and install it

Step 3: Configure Installation

Important: Select Check box:

"Add Python 3.10 to PATH"

Step 4: Complete Installation

Wait for the installation to finish.

Click "Close" when done.

Step 5: Verify Python Installation

Open Command Prompt

```
python --version
python -m pip --version
```

How to Install & Configure PyCharm in windows or mac??

1. Go to the official Pycharm website:

https://www.jetbrains.com/pycharm/download/?section=windows

or

https://www.jetbrains.com/pycharm/download/?section=mac

2. Scroll to the page and find "PyCharm Community Edition" and Click Download button to download

Run the Installer

Goto Downloads folder, Double-click the exe "pycharm-community-2025.1.3.1" and install it

Configure Installation

Choose an installation location (default is fine).

Click Next → then Install.

Launch Pycharm & Create Project

Goto Search -> click Pycharm 2025.2.3 -> Choose the Interpreter -> Create Project

Python Virtual Environment

A **virtual environment** is an isolated space to install project-specific Python packages without changing the global Python setup.

- Prevent package conflicts across projects
- Use different versions of the same library

Creating and Using Virtual Environments

```
python -m venv env_name
.\env_name\Scripts\activate
deactivate
```

Pip Commands

Command	Description
pip install <package_name>	Install a package (e.g., pip install numpy)
pip uninstall <package_name>	Remove a package
pip list	Show installed packages
pip freeze	List installed packages with versions
pip show <package_name>	Show details about a package
pip install -r requirements.txt	Install all packages listed in a file

Functional Programming

What is Functional Programming?

- Functional programming makes it easier to write **concurrent or multithreaded** applications.
- Functional programming languages make it easier to write elegant code, which is **easy to read, understand, and reason** about.

Functions – Characteristics

First Class

FP treats functions as first-class citizens. A function has the same status as a variable or value. It allows a function to be used just like a variable. While in case of any imperative language such as C it treats function and variable differently.

Composable
Functions in functional programming are composable. Function composition is a mathematical and computer science concept of **combining simple functions to create a complex one**.

No Side Effects
A function in functional programming does not have side effects. The result returned by a function depends only on the input arguments to the function**. The behavior of a function does not change with time. It returns the same output every time for a given input, no matter how many times it is called**. In other words, a function does not have a state. It does not depend on or update any global variable.

Simple
Functions in functional programming are **simple**. A function consists of a few lines of code and it does only one thing. A simple function is easy to reason about. Thus, it enables robustness and high quality.

Variables -

A name or an identifier to refer the memory location of an object or a value

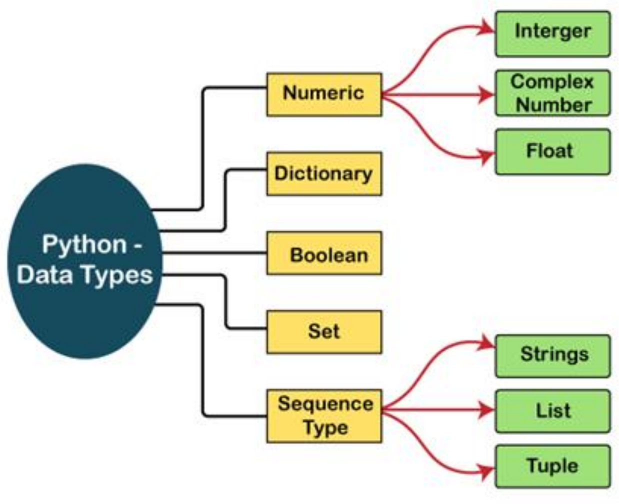
Characteristics:

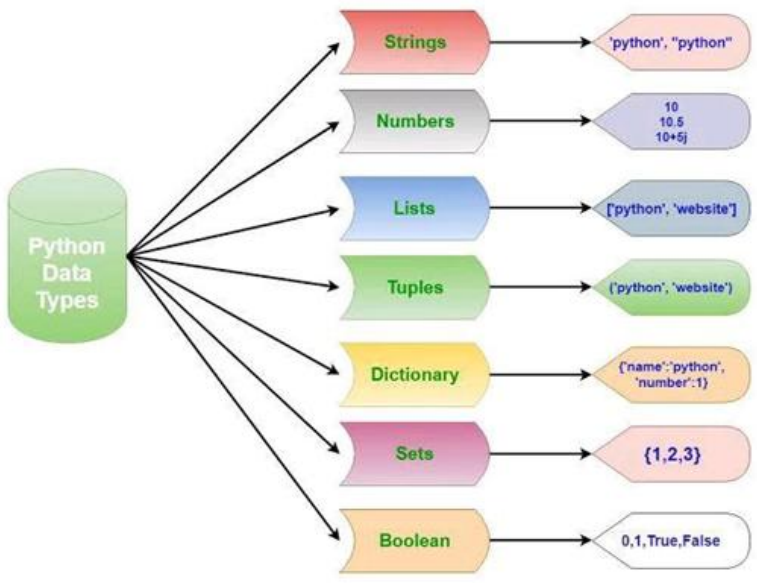
- Dynamically Inferred
- Dynamically Typed
- Strongly Typed

Naming Convention:

- ***Name of a variable shouldn't be a reserved/predefined/builtin function names/keyword.***
- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Python DataTypes:





Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Let’s Explore Python:

Python Exercises:

The easiest way to get started with Python is by using the Python interpreter, which provides an interactive shell for writing Python code. It is a **REPL** (read, evaluate, print, loop) tool, we have other options too such as IDEs, Notebooks, Online Interpreters etc.,

Variables

Case sensitive variable names

```
name =  'Inceptez Technologies '
Name = " 'Inceptez!' "
print (name)
print (Name)
```

Text Type: str

```
x='Hello Inceptez'
print(x)
print(type(x))
```

Usage of single, double, triple quotes

```
name = " 'Inceptez!' "
print (name)
print (type(name))
name =  '''Inceptez
Technologies '''
print (name)
print (type(name))
```

Numeric Types: int, float, complex

```
x=100
print(x)
print(type(x))

x=100.5
print(x)
print(type(x))

x=10e3
print(x)
print(type(x))
```

Boolean Type: bool

```
x=True
print(x)
print(type(x))
```

None Type: None

The None keyword is used to define a null value, or no value at all. None is not the same as 0, False, or an empty string

```
x=None
print(x)
print(type(x))
```

Binary Types: bytes

```
x=100
(bytes(x))
```

Type Casting:

```
x = str(100)
y = int(100)
z = float(100)
```

Collection Types:

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

We will see soon

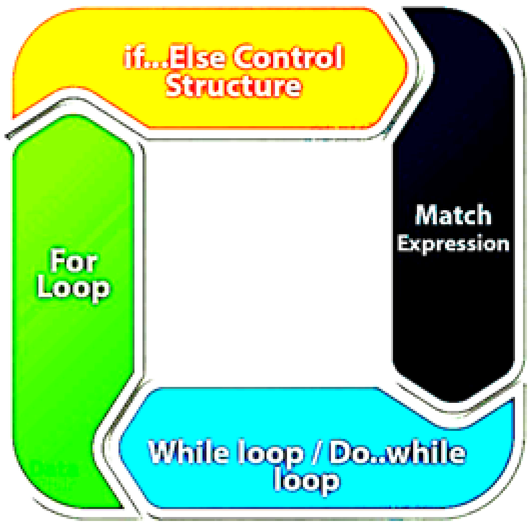
Operators:

== != >

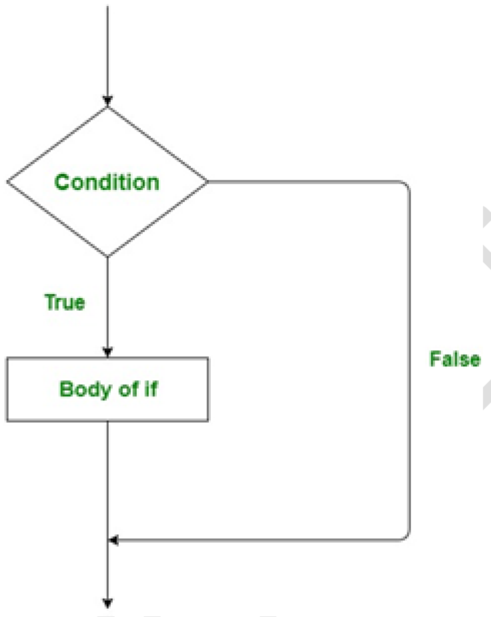
< >= <=

- ✓ Assignment Operators
  - Used to assign or operate on the variable values.
  - Eg: x=10, x=20; x=x+30 or x+=30
- ✓ Arithmetic Operators
  - Used for performing functional/transformational operations on the variables.
  - Eg: +,-,*,/, **
- ✓ Relational Operators
  - Used for performing comparison operations on the variables.
  - Eg: ==, !=, >, <, <=, >=
- ✓ Logical Operators
  - Used to perform conditional operations on the variables.
  - Eg: and, or, not

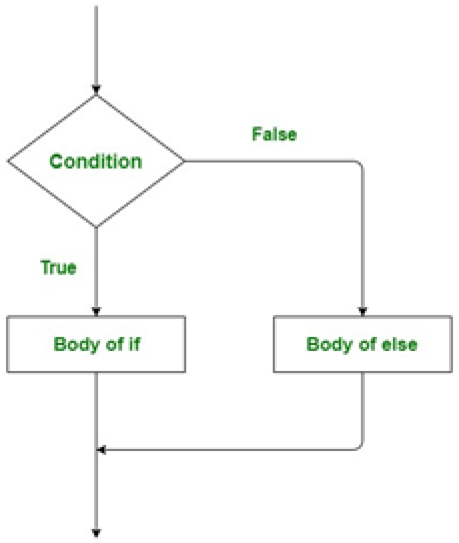
Control Structures:



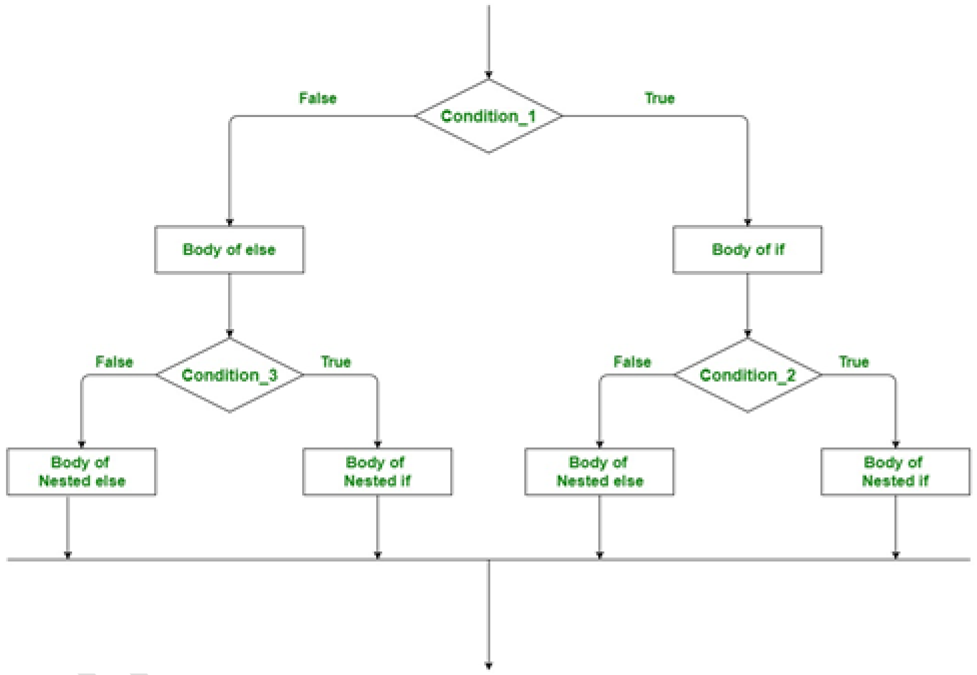
If then



If then else



Nested if then else if else If then else if



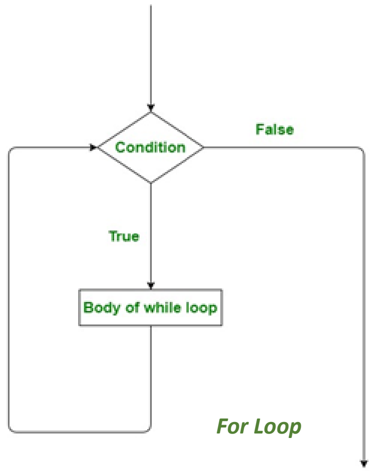
```
x=5
y = 5

if x > y  :
   print ("x is greater")
elif y>x :
   print ("y is greater")
```

```
else:
   print("x equals y")
```

Loops:

For loops:

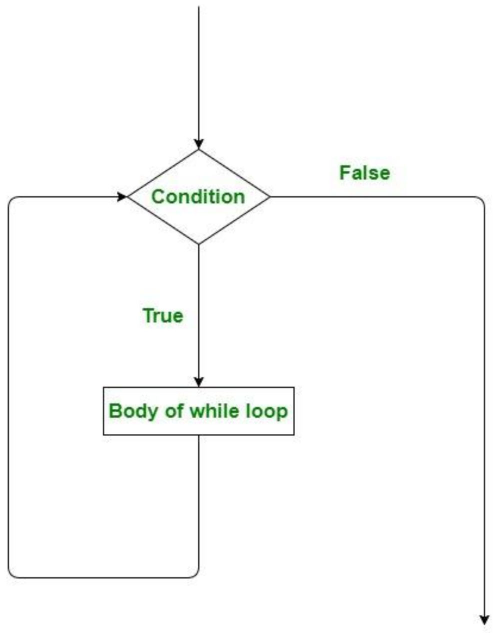
For loops are preferred when the number of times loop statements are to be executed is known beforehand



```
for i in list(range(1,10)) :
 print('squared ' + str(i*i))
```

While loops:

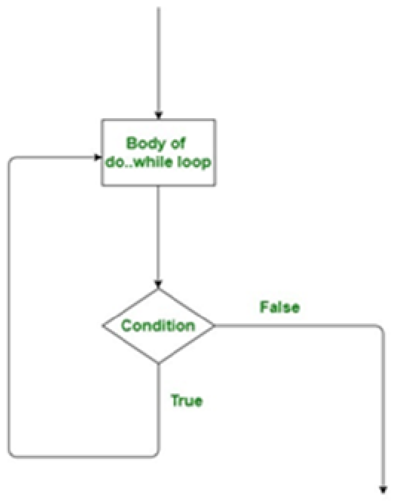
While loops are same as For loop, used when we don’t know the number of times we want the loop to be executed, however we know the termination condition of the loop. It is also known as an ***entry controlled***



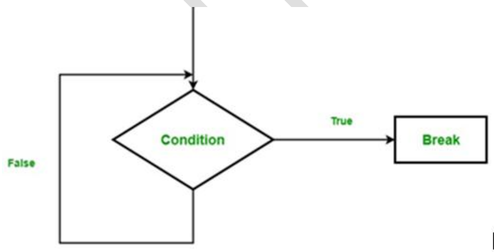
```
x = 10
while x >= 0:
   print(x)
   x -= 1
```

Do While – not available in python

```
i = 1
while True:  #infinate loop (scheduling), Do-While
   print(i)
   if i <= 2:
      break
   i -= 1

print("Do while example loop with break completed here")
```



Break :



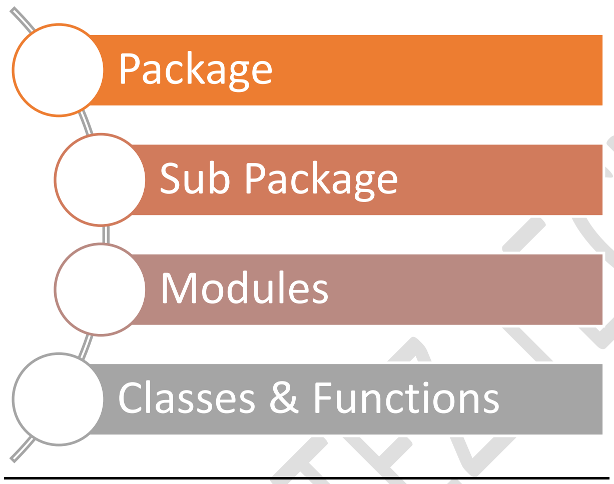
We use break statement to break the execution of the loop in the program

Python Switch Statement

- ✓ Unlike other programming language like Scala or Java, Python does not support case or switch, hence we can use dictionary to achieve it
- ✓ It is a technique for checking a value against a pattern.

```
case_switch_using_dict = { 0: "zero",  1: "one", 2: "two",}

case_switch_using_dict[1]
```

Hierarchy of Programs in Python:

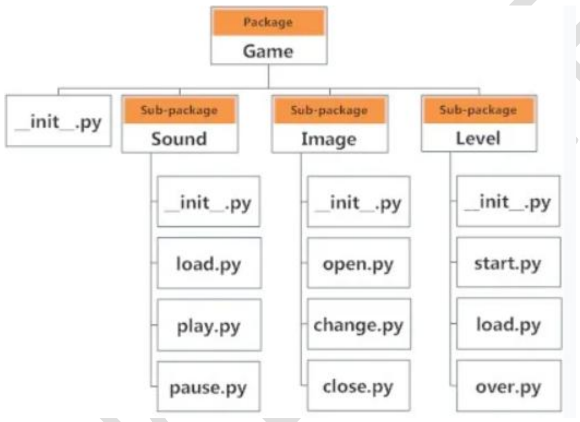


Package:

- *A package is a hierarchical file directory structure to encapsulate/collect/contains group of related Sub-package folders and Modules. Mainly used for segregation of the programs.*
- *Any definitions placed in a package object are considered members of the package itself*
- *__init__.py helps the Python interpreter to recognize the folder as package*
- *Sub package is a sub folder for maintained to organize and categorize the modules efficiently.*

Module:

- *A Python module is a file or files may contain several classes, functions, variables, etc.*
- *Modules are .py files, will help us manage the programs.*
- *In simpler terms a package is folder that contains various modules as files.*



<u>How to invoke a package/subpackage/module/class/method</u>

We can import modules from packages using the dot (.) operator.
For example:

```
import package.subpackage.module

import Game.Level.start
#we must use the full name to reference it.
```

Game.Level.start.select_difficulty()

(or)

from package.subpackage import module

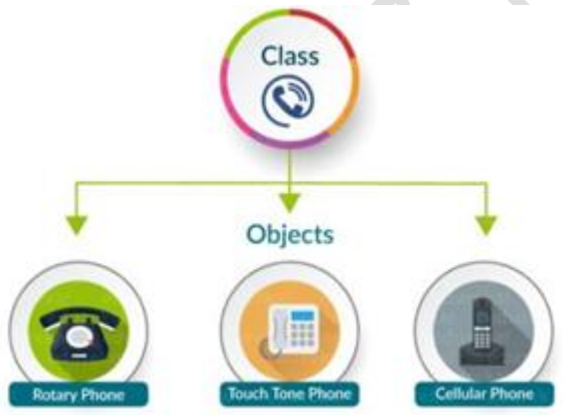
from Game.Level import start

start.select_difficulty(2)

(or)
from package.subpackage.module import class
from package.subpackage.module import *

from Game.Level.start import select_difficulty
select_difficulty(2)

Classes & Objects



- A class is a template or blueprint for creating objects at runtime.
- A class is defined using the keyword class.
- A class is defined in source code.
- A class definition starts with the class name, followed by colon, followed by optional __init__ function to instantiate the class with arguments, followed by the member functions

Class

Init Function

Member Function

Class:

```
class Telephone :
   def __init__(self, model,costperhour):
      self.model=model
      self.costperhour=costperhour
   def installationcost(self,hoursforinstallation):
     return hoursforinstallation*self.costperhour
```

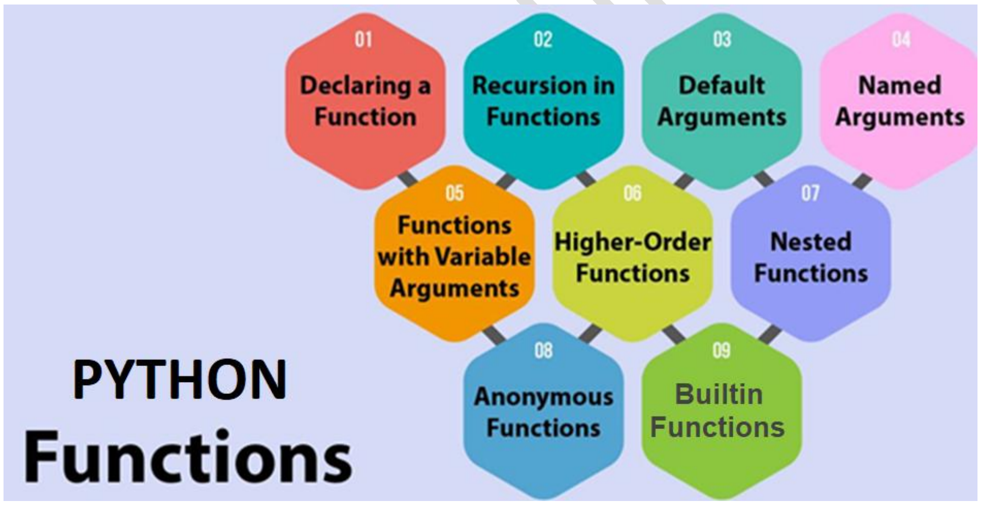
Object:

- ➢ An object is an instance of a class.
- ➢ To load/instantiate/initialize and make use of the class, we have to instantiate in the form of objects

Object

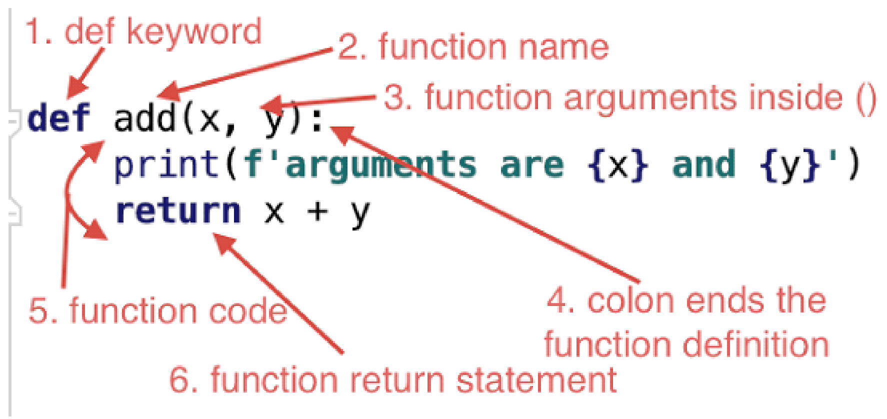
```
rotaryobject= telephone("rotary",10)
rotaryobject.installationcost(4)
```

Functions



- A function/method is a collection of statements that perform a certain task.
- Functions are used to put some common and repeated task into a single function, so instead of writing the same code again and again for different inputs, we can simply call the function.
- Python is assumed as functional programming language so these play an important role. It makes easier to debug and modify the code.
- Naming Convention- function name should be in lower case with underscores for eg. lower_case_with_underscore()

Function Declaration & Definition


1. def keyword
2. function name
3. function arguments inside ()
4. colon ends the function definition
5. function code
6. function return statement

```
def add(x, y):
    print(f'arguments are {x} and {y}')
    return x + y
```

Function declaration & definition have 6 components:

- **def keyword:** “def” keyword is used to declare a function in *Python*.
- **function_name:** It should be valid name in lower case with underscore in between words and cannot use special characters in the function names.
- **parameter_list:** optional comma-separated list of the input parameters are defined, within the enclosed parenthesis.
- : - Function name ends with “:” is a syntax to end the function definition
- **Function body:** Body of the code is developed by the necessary indentation levels.
- **return_type:** return statement with the result can be mentioned while defining function and return type of a function is optional (returns None if not mentioned).

Example1:

```
def return_mail_id(fname,lname): #number of arguments can be any or nothing
   mail_id=fname +  '.' + lname + '@gmail.com'
   return mail_id #optional return statement
```

Example2:

```
def optimal_offer(purchase_amount, offer_percent, max_offer_amt):
   calc_offer_amt = purchase_amount*offer_percent
   if (calc_offer_amt > max_offer_amt):
      print("Calculated offer amount is more than the max offer amount, hence returning the
max_offer_amt applied total amount")
      return purchase_amount-max_offer_amt
   else:
      print("Calculated offer amount is less than the max offer amount, hence returning the
offer_percent applied total amount");
      return purchase_amount -calc_offer_amt
```

Calling functions:

Positional Arguments:

```
print(optimal_offer(1000, .04, 50))
print(optimal_offer(1000, .06, 50))

return_mail_id("inceptez","technologies")
```

Keyword Arguments:

```
return_mail_id(lname="technologies",fname="inceptez")
```

Default Arguments Value:

If we call the function without argument, it uses the default value:

```
def return_mail_id(fname,lname,domainname="@inceptez.com"):
   mail_id=fname +  '.' + lname + domainname
   return mail_id

return_mail_id("inceptez","technologies") #Default arguments
return_mail_id("inceptez","technologies","@gmail.com") #Overriding arguments
```

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

```
def return_mail_id(*name):
   mail_id=name[0] + '.' + name[1] + '@gmail.com'
   return mail_id

return_mail_id("inceptez","technologies")
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly:

```
def return_mail_id(**name):
   mail_id=name["fname"] +  '.' + name["lname"] + '@gmail.com'
   return mail_id

return_mail_id(lname="technologies",fname="Inceptez")
```

Global & Local Variables scope in Python Functions:

The basic rules for local and global keyword:

1. When we create a variable inside a function, it is local by default.
2. When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
3. We use global keyword to read and write a global variable inside a function.
4. Use of global keyword outside a function has no effect.

```
global default_domainname  # no need to define, by default global variable
default_domainname= '@explicit_default_domainname.com' # explicit Global variable
default_global_domainname= '@default_global_domainname.com' # By default Global variable

def return_mail_id(fname,lname):
   local_domain_name= '@gmail.com' # By default local variable
   global global_domain_name
   global_domain_name= '@inceptez.com' # global variable
   local_mail_id=fname +  '.' + lname + local_domain_name
   print(local_mail_id)
   global_mail_id=fname +  '.' + lname + global_domain_name
   print(global_mail_id)

return_mail_id("Inceptez","technologies")
print(default_domainname)
print(default_global_domainname)
```

Special types of Functions:

Higher-Order Functions

A Function that takes another function as an input parameter is called a *higher-order method*.

```
def add_all(numbers):
   return sum(numbers)

def sqrt_hof(input_hof_func,args):
   sum_of_args=input_hof_func(args)
   return sum_of_args*sum_of_args

print("Square feet of the sum of second argument is "+ str(sqrt_hof(add_all,[1,2,3])))
```

Anonymous or Lambda Functions

An anonymous function is a function that is defined without a name and created anonymously for using it within a program scope and not across the application.

While normal functions are defined using the def keyword in Python
Anonymous functions are defined using the lambda keyword, anonymous functions are also called lambda functions.

```
~~def add_all(numbers):~~
~~    return sum(numbers)~~

anonymous_add_all =(lambda a:sum(a))
```

Eg: Display the salary of all employees who is getting more than 10000

```
lst=[10000,20000,5000,25000,9000,15000]
lambda_filter=lambda x:x>10000
list(filter(lambda_filter,lst))
```

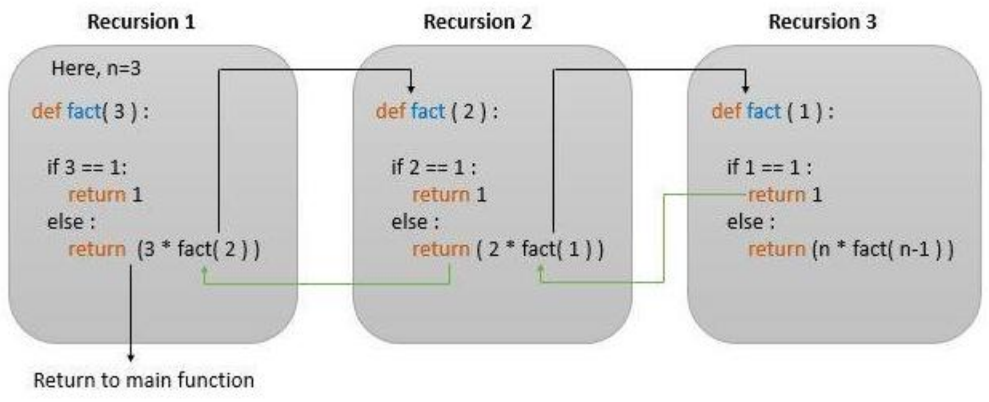
Closures

A closure is a function, whose return value depends on the value of one or more variables declared outside the current function. It can access these variables even after the outer function has completed its execution. Closures can avoid the use of global values and provides some form of data hiding.

```
def sal_hike(sal,hike):
    salhike=sal+hike
    def incentives(incentive):
        return salhike+incentive
    return incentives

print(sal_hike(20000,2000))
mainout = sal_hike(20000,2000)
print(mainout(1000))
```

Python Recursive Functions

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.



```
def recurse_fact(arg):
   if arg == 1:
      return 1
   else:
      return (arg * recurse_fact(arg-1))

recurse_fact(3)
```

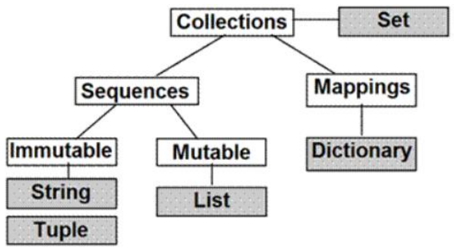
Collections

A collection is a container data structure. It contains zero or more elements. Collections provide a higher- level abstraction for working with data. They enable declarative/function based programming. With an easy-to use interface, they eliminate the need to manually iterate or loop through all the elements.

- **List** is a collection which is ordered and changeable. Allows duplicate members. Append/insert/+/__add__(), pop/remove, lst[index]=value_to_update
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is pre ordered(internally), not updatable but you can remove items and add new items, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered (As of Python version 3.7, dictionaries are pre ordered (internally) based on the key) and changeable. No duplicate members (keys), duplicate values can be accepted.

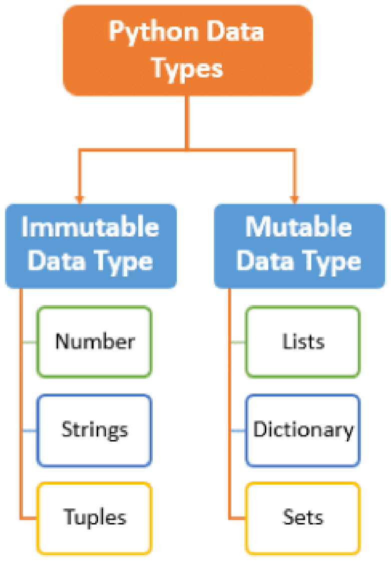
Categories:

- Sequences (list, tuples)
- Dictionary
- Set



Iterable:
Iterators are data structures/functions that allow to loop/iterate over a sequence of elements
Python includes an elegant and powerful collection library. They are easy-to-use, concise and universal.

Python collection is systematically distinguished between ***mutable and immutable***.



Mutable collection can be updated/added/deleted with the elements.

Immutable collection by contrast never changes, but still~~, can be added/deleted/updated by internally recreated.~~

Comparison Chart:

Collection Type (Iterable)	**Example (Syntax)**	**Access (Select)**	**Mutable/Immutable (update)**	**Re sizable (insert/delete)**	**Application (Symantics)**
List Notation: []	lst=[10,20,30] lst[0]	Access using Index starts with 0 lst[0]	Updatable	Insert/delete	Indexed & Sequenced collection of same type elements Used to store events/values in sequence Eg: salary, amount, dates etc.,
Dictionary Notation: {k,v}	kvpair={1:"Inceptez",2:"Apple"} kvpair[1] kvpair[2]="IBM"	Access using key dict[key]	Updatable	Insert/delete	Unordered Collection of Key:Value pairs Key is unique, values can be non unique Used for lookup, enrichment Eg: custno:mobilenum, city:state, acctnum:account balance

Set
Notation: {}
	st={1,2,4,3,5}
	st.add(6)
	st.remove(4)
	st.update({10,20})
		Iteration	Updatable	Insert/delete	Collection of sequenced, unique elements
					Used for performing set operations like union, intersect, difference etc.,
					Eg: Combine 2 set of customer mobile or account numbers

Tuple Notation: ()	tup=(1,"Inceptez",100)	Access using Index starts with 0 tup[0]	Immutable	Non Resizable	Indexed collection of different type elements Used to store collection of different types Eg: record or recordset

Sequences

A *sequence* represents a collection of elements in a specific order. Since the elements have a defined order, they can be accessed by their position in a collection.

For example, you can ask for the *nth* element in a sequence.

List

- ✓ **A *List* is an indexed sequence of elements.**
- ✓ All the elements in a list are of the same type.
- ✓ It is a mutable data structure.
- ✓ You can add/remove/update element to an array after it has been created.
- ✓ Lists can be declared by using square brackets "[]" following the variable name.

```
lst = [10, 20, 30, 40]
```

Tuples

A *tuple* is a container for storing two or more elements of different types. It is immutable, it cannot be modified after it has been created.

```
twoElements = ("10", true)
threeElements = (10, "harry", true)
```

An element in a tuple can be accessed using index starting 0.

```
first = threeElements[0]
second = threeElements[1]
```

Dictionary

- ✓ *Dictionary* is a collection of key-value pairs.
- ✓ *Dictionaries are mutable*
- ✓ You can add the key,value using the update function, if the key is present it will update else insert to a dict after it has been created. Delete can be done using pop function.
- ✓ Dictionary can be declared by using flower brackets "{}" following the variable name.
- ✓ In other languages, it is known as a map, associative array, or hash map.
- ✓ It is an efficient data structure for looking up a value by its key.

```
capitals = {"USA":"Washington D.C.", "England" :"London", "India":"New Delhi"}

capitals["India"] #If the value not found it will throw key error

capitals.get("India","NA") #To avoid key error we can use get function

capitals.update({"India":"Delhi NCR"})  #update the value if key exists

capitals.update({"Malaysia":"KL"})  #add the key,value if key doesn't exists

capitals.pop("England")  #Delete the key and the value
```

Set

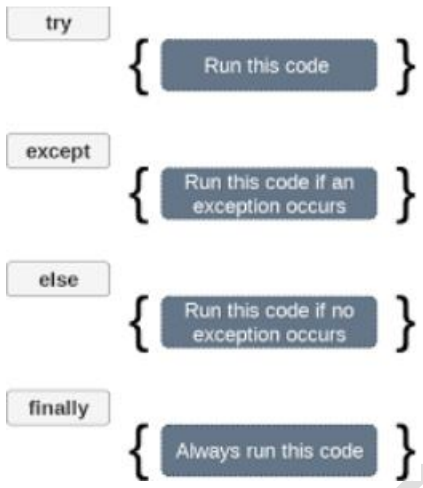
- ✓ Set is a collection that contains no duplicate elements and sorted by default.
- ✓ It is a mutable data structure.
- ✓ You can add/remove elements to a set after it has been created.
- ✓ In order to update we have to remove and add a given value, update function in set is used to merge 2 sets.
- ✓ Set can be declared by using flower brackets "{}" following the variable name.
- ✓ Set is useful in performing set operations like union, intersection, difference etc.,

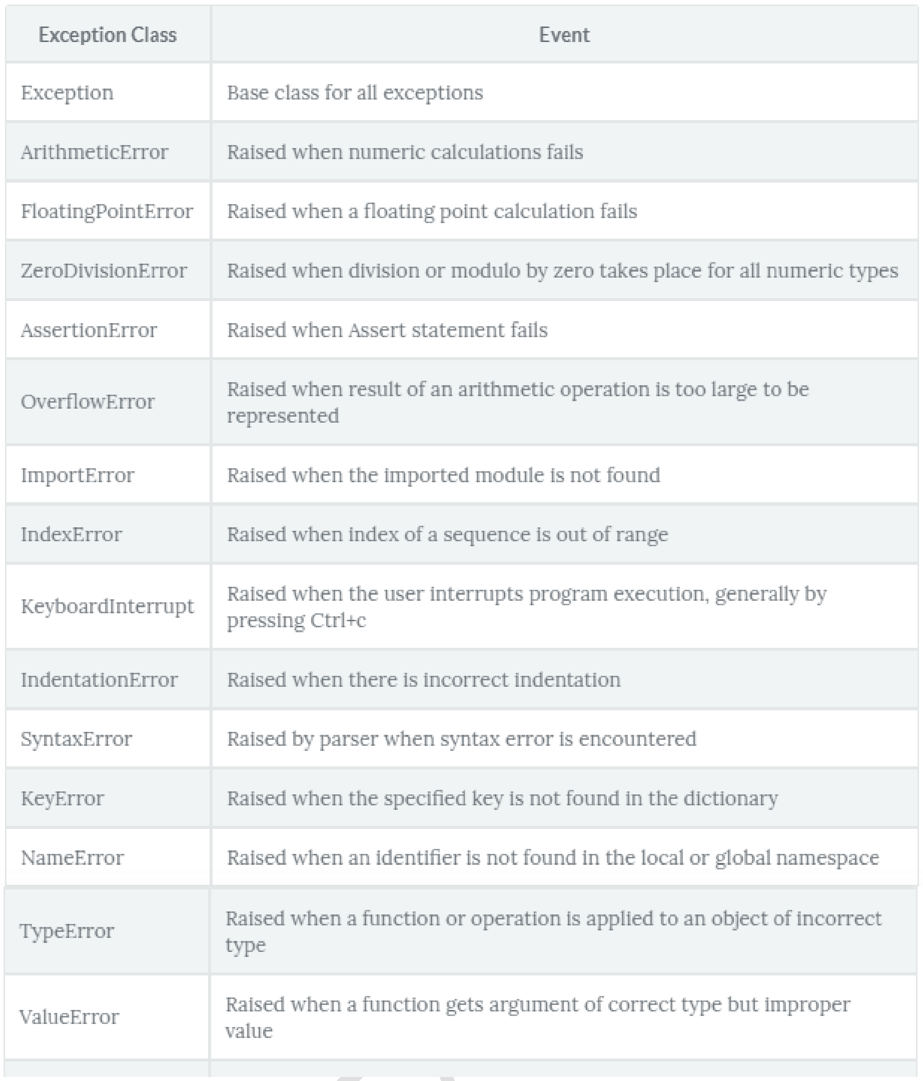
```
set1={"Washington D.C.", "London"}

set1.add("Delhi")

set1.remove("London")
```

If we need to update a set with another collection we can update it.

```
set2={"Beijing", "Penang"}

set1.update(set2)

print(set1)
```

Exception Handling


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code



Exception Class	Event
Exception	Base class for all exceptions
ArithmeticError	Raised when numeric calculations fails
FloatingPointError	Raised when a floating point calculation fails
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types
AssertionError	Raised when Assert statement fails
OverflowError	Raised when result of an arithmetic operation is too large to be represented
ImportError	Raised when the imported module is not found
IndexError	Raised when index of a sequence is out of range
KeyboardInterrupt	Raised when the user interrupts program execution, generally by pressing Ctrl+c
IndentationError	Raised when there is incorrect indentation
SyntaxError	Raised by parser when syntax error is encountered
KeyError	Raised when the specified key is not found in the dictionary
NameError	Raised when an identifier is not found in the local or global namespace
TypeError	Raised when a function or operation is applied to an object of incorrect type
ValueError	Raised when a function gets argument of correct type but improper value

What is Exception

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time. These events change the flow control of the program in execution.

Why Exception handler is needed

When an exception occurs, say an ArithmeticException as shown in the previous example the current operation is aborted, and the runtime system looks for an exception handler that can accept an ArithmeticException. Control resumes with the innermost such handler. If no such handler exists, the program terminates.

Exception handler blocks:

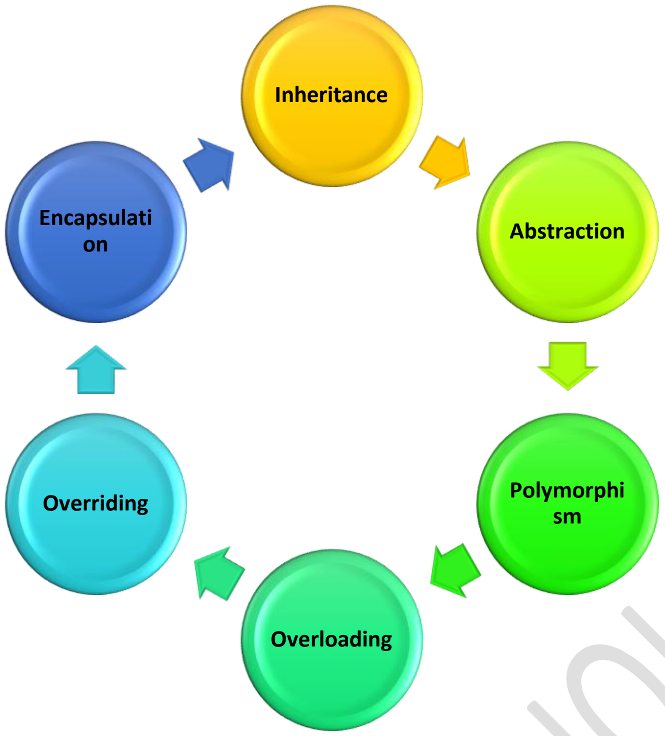
The *try* & Except Constructs
Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

The finally Construct :
If we want some part of our code to execute irrespective of how the expression terminates we can use a finally block. .

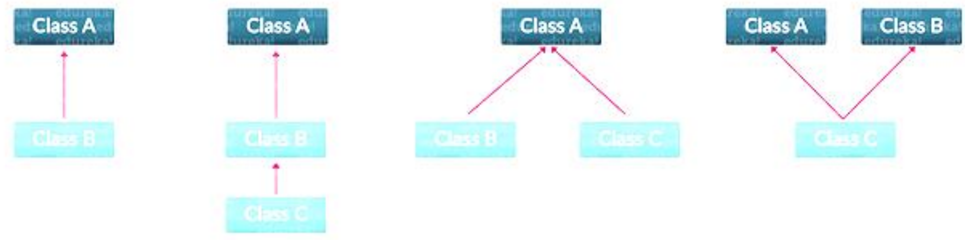
Eg:

```
try:
   print("Try Block1")
   num1 = 10
   "Strongly typed" + num1
except Exception as i_store_the_errormessage:
   # except block will be called when try block throws error
   print("General Exception Block, error message is -> {} ".format(i_store_the_errormessage))
else:
   print("I will be executing if no exception occurs in the try block - closing connections created in the try block")
finally:
   print("I will be executing at any cost, whether exception occured or not - closing all connections created in this code")
```

<u>OOPS Concepts</u>



Inheritance:

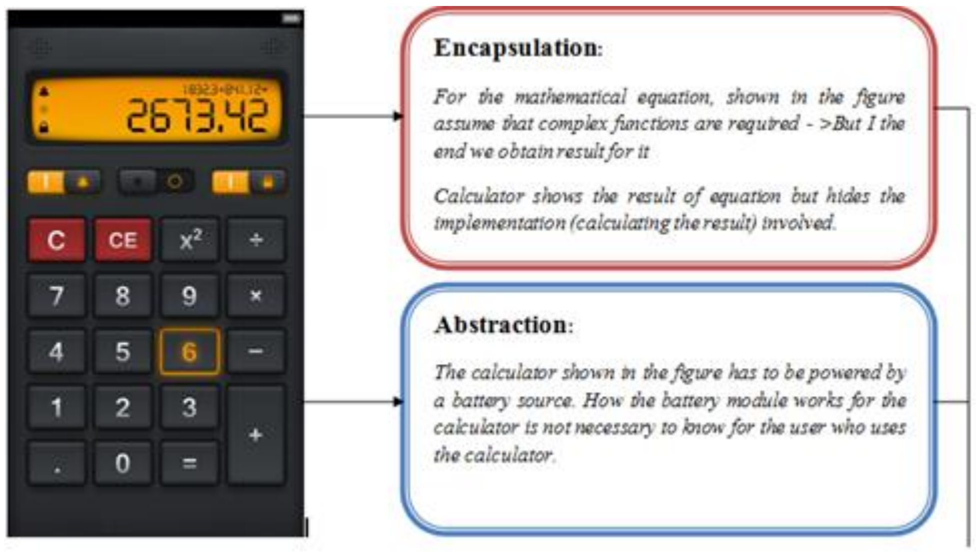


- ➢ Inheritance is the process of inheriting the features/members of the parent class
- ➢ Types: Single, Multilevel, Hierarchical, Multiple
- ➢ Multiple Inheritance: In Multiple inheritance , one class can have more than one superclass and inherit features from all parent classes

Key advantages/differences of OOP compared to POP (Procedural Oriented Programming)

- ➢ OOPs has a Bottom to Top approach compared to POP's Top to Bottom approach.
- ➢ Object Oriented Programming supports inheritance but POP does not support inheritance.
- ➢ Object Oriented Programming has more security compared to Procedural Oriented Programming.
- ➢ OOPs uses access modifiers such as 'Public', 'Private' and 'Protected' whereas POP does not support any of that.
- ➢ In Object Oriented Programming the program is divided into 'Objects'. And in Procedural Oriented Programing it is divided into 'Functions'.

Abstraction:



Encapsulation:

For the mathematical equation, shown in the figure assume that complex functions are required - >But I the end we obtain result for it

Calculator shows the result of equation but hides the implementation (calculating the result) involved.

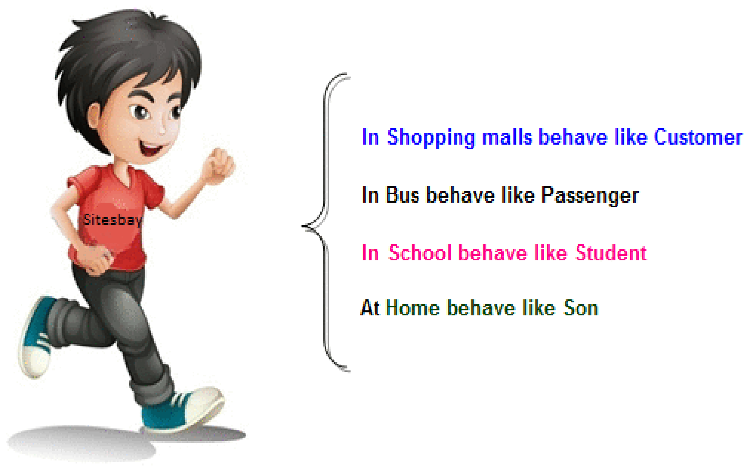
Abstraction:

The calculator shown in the figure has to be powered by a battery source. How the battery module works for the calculator is not necessary to know for the user who uses the calculator.

- ➢ Abstraction - Abstraction is the process to hide the internal details and showing only the functionality.
- ➢ Abstraction is achieved by using an abc class.
- ➢ In Python, we can achieve abstraction by incorporating abstract classes and methods.
- ➢ Any class that contains abstract method(s) is called an abstract class. Abstract methods do not include any implementations – they are always defined and implemented as part of the methods of the sub-classes inherited from the abstract class.

```
from abc import ABC,abstractmethod

class type_shape(ABC):

 @abstractmethod

 def area(self):

  #abstract method

  pass


class Rectangle(type_shape):

 length = 6

 breadth = 4

 def area(self):

  return self.length * self.breadth

r = Rectangle() # object created for the class 'Rectangle'

print("Area of a rectangle:", r.area()) # call to 'area' method defined inside the class.
```

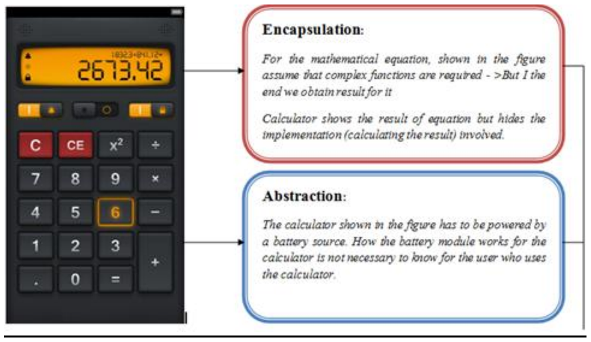
Polymorphism/Overloading:


Sitesbay
In Shopping malls behave like Customer
In Bus behave like Passenger
In School behave like Student
At Home behave like Son

- Python implements polymorphism through virtual functions, overloaded functions and overloaded operators. The word Polymorphism itself indicates the meaning as Poly means many and Morphism means types
- Polymorphism means that a function type comes "in many forms". The type can have instances of many types.
- Example - Method with different number/type of arguments

```
print(len["Inceptez","Technologies"])

print(len(["Inceptez","Technologies"]))


pip3 install multipledispatch

from multipledispatch import dispatch

@dispatch(int, int)

def f1(a,b):

 print (a+b)


@dispatch(int, str,str)

def f1(a,b,c):

 print( a+int(b)+int(c))


print(f1(10,2))

print(f1(10,'2','2'))
```

Encapsulation:



Encapsulation:

For the mathematical equation, shown in the figure assume that complex functions are required - >But I the end we obtain result for it

Calculator shows the result of equation but hides the implementation (calculating the result) involved.

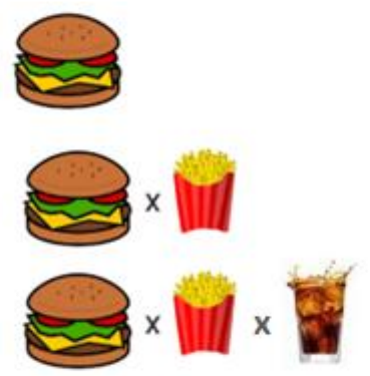
Abstraction:

The calculator shown in the figure has to be powered by a battery source. How the battery module works for the calculator is not necessary to know for the user who uses the calculator.

- Encapsulation is the method of hiding/restricting the access for certain members defined in a class.
- Specify access specified/modifier for providing access control to the objects or values
- Example : private var a=100;

```
class Base:
  def __private(self): #Scope within this class
    print("private value in Base")
  def _protected(self):
#Protected variables are those data members of a class that can be accessed within the class and the classes derived from that class
    print("protected value in Base")
  def public(self):
    print("public value in Base")
    self.__private()
class Derived(Base):
  def _protected(self):
    print("protected value in Derived")
```

Overriding:



- In Python when you have two methods with the same name that each perform different tasks then it is overriding, the overrided method provides your own implementation of it.
- When a class inherits from another, it may want to modify the definition for a method of the superclass or provide a new version of it.
- We use the 'override' modifier to implement this.
- Example : Method or variables in python override with different implementations

```
print('Python Overriding Example')
class Mobile:
   def GSM(self):
      print('Base Mobile Class Method')
class Samsung(Mobile):
   def GSM(self):
      print('Child Samsung Class Method for GSM')
   def CDMA(self):
      print('Child Samsung Class Method for CDMA')
mob_obj = Mobile()
mob_obj.GSM()
sam_obj = Samsung()
sam_obj.GSM()
sam_obj.CDMA()
```