CSD 2101

Python Programming

MODULE IV CLASS AND OBJECTS 9

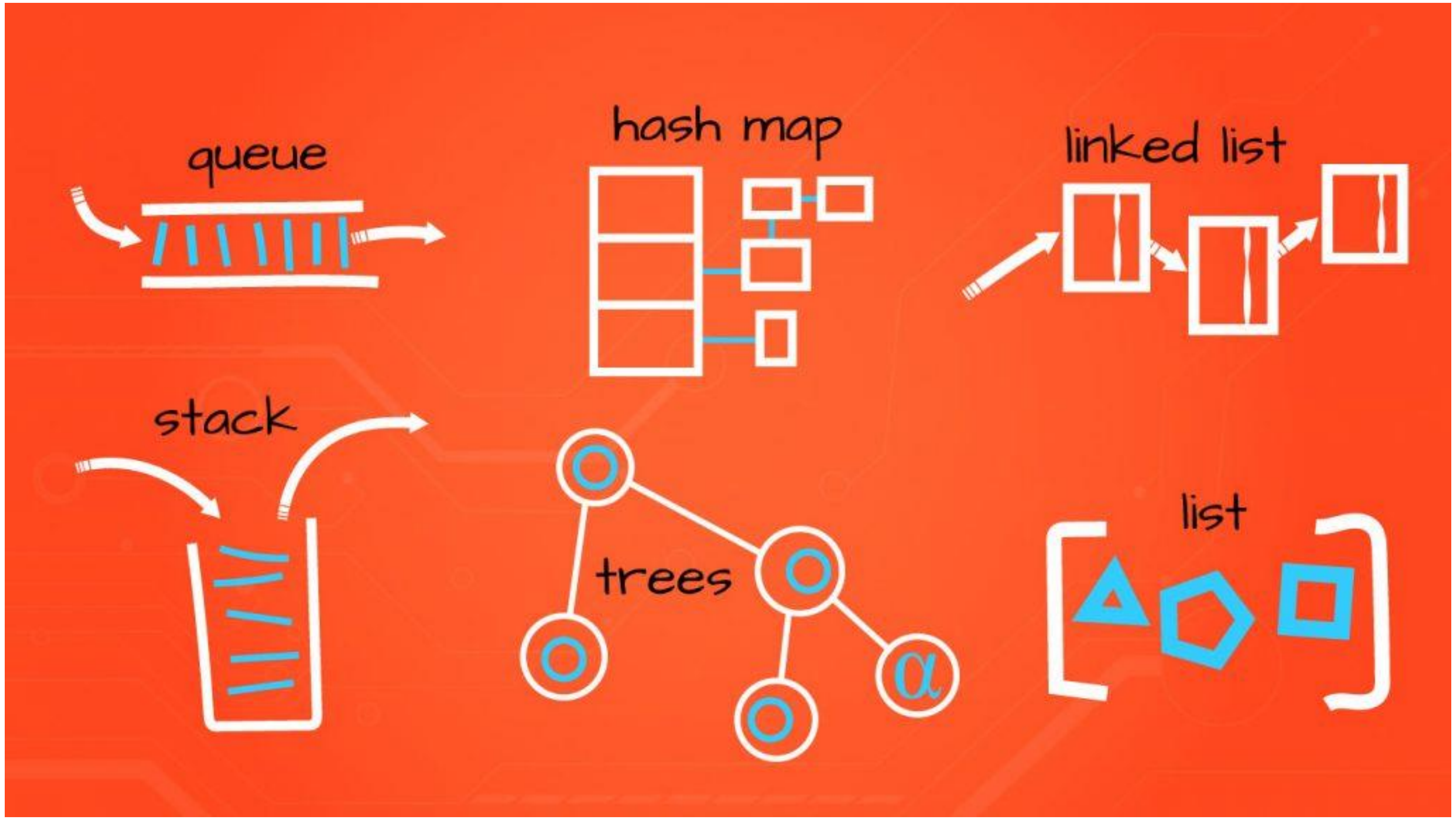
Abstract Data Types – Classes – Inheritance – Multiple level of Inheritance – Substitution Principles – Encapsulation and Information Hiding- Python Standard Libraries–Packages.

Abstract Data Types

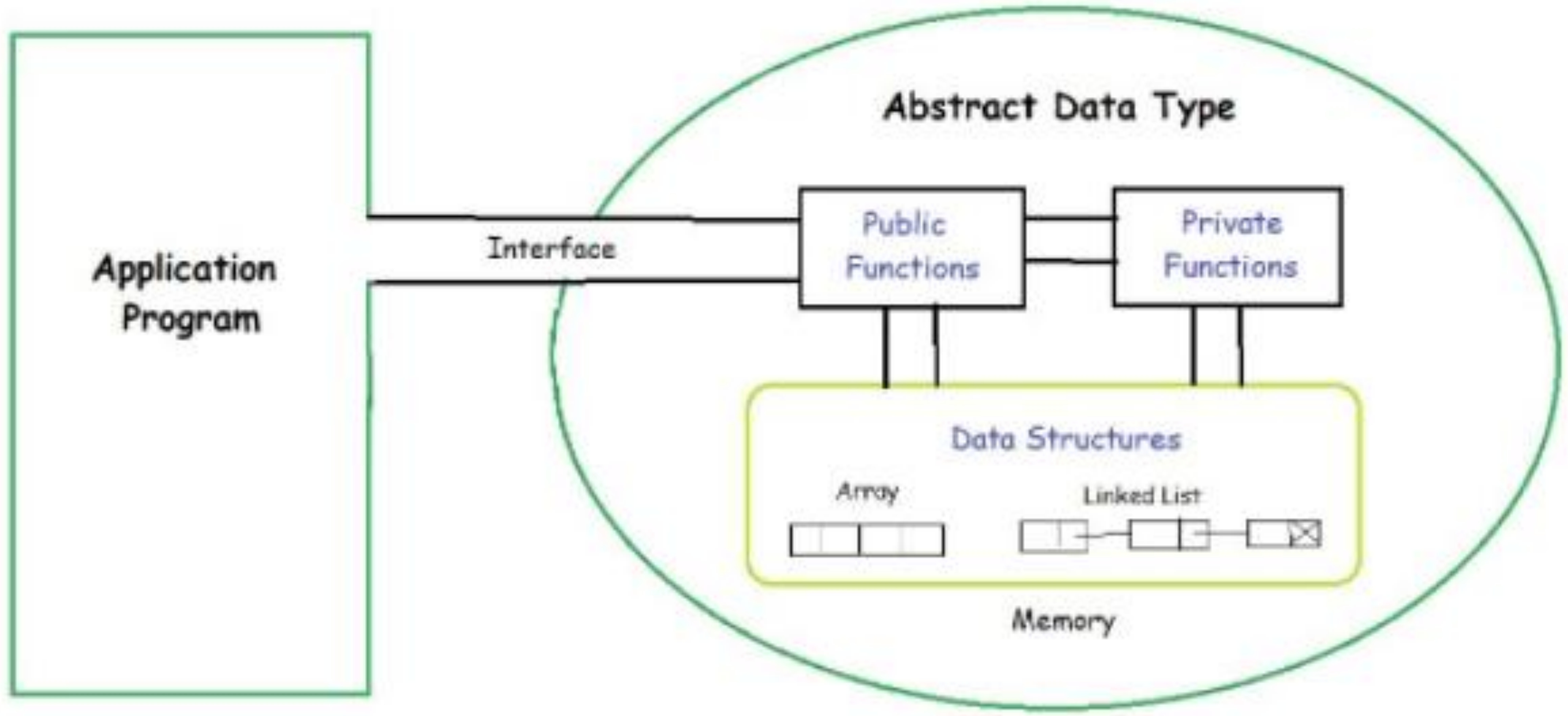
- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- An ADT involves both data and operations on that data.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- The process of providing only the essentials and hiding the details is known as abstraction.

Abstract Data Types

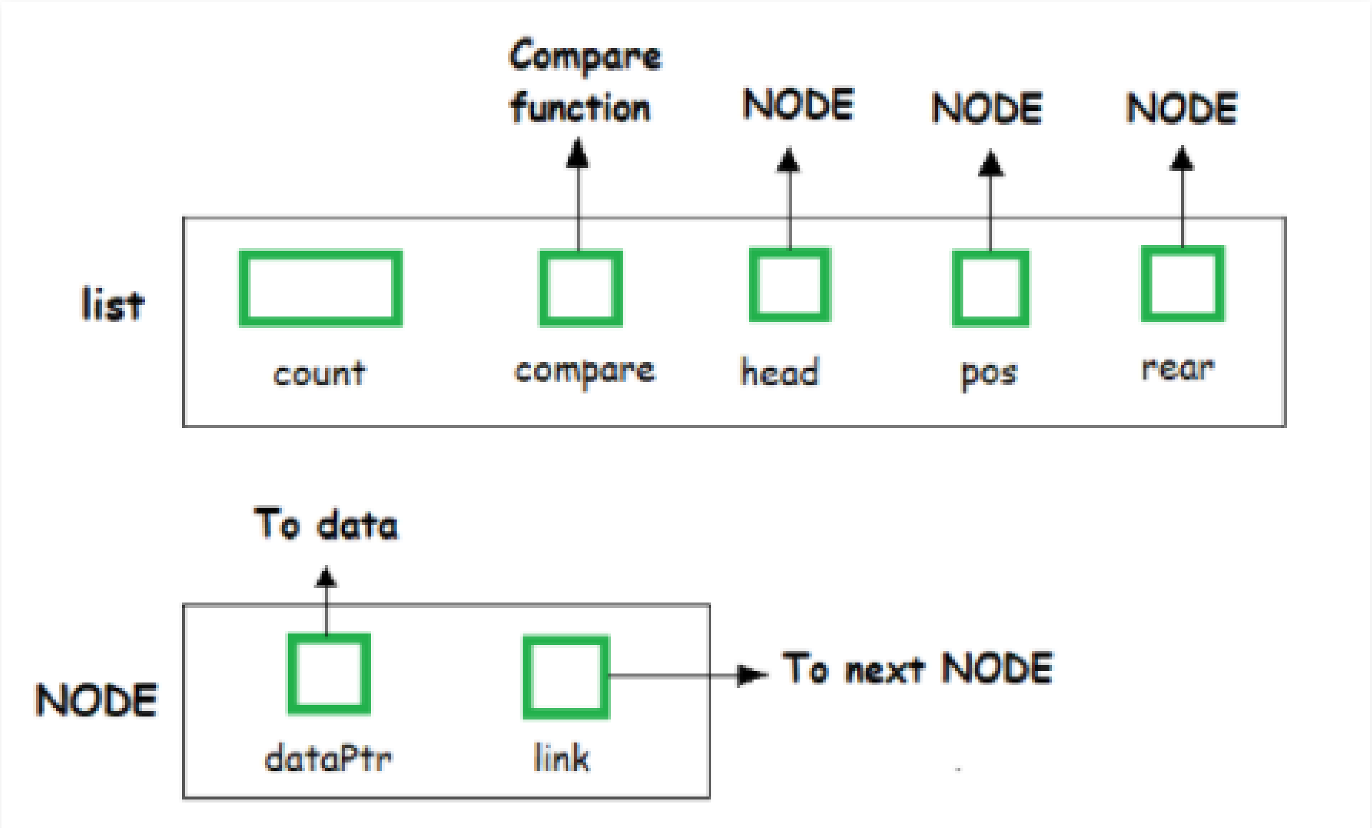
- **Abstract:** Conceptual rather than concrete
- **Data:** Information formatted for use by a computer
- **Type:** A category of objects with shared characteristics



- ADT as a black box which hides the inner structure and design of the data type. The three ADTs namely List ADT, Stack ADT, Queue ADT.



1. **List ADT**
   - The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.



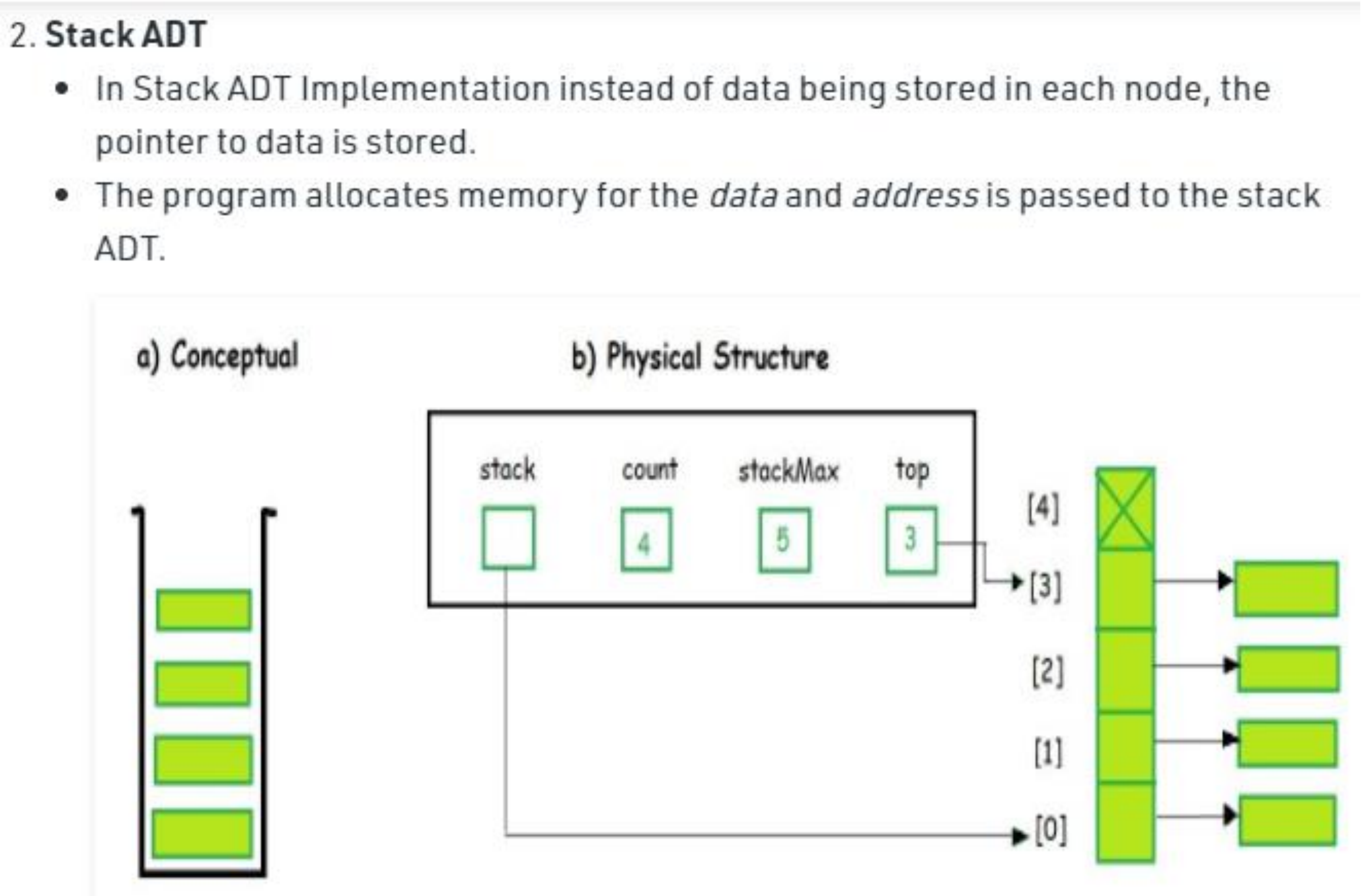
- size(), this function is used to get number of elements present into the list.
- insert(x), this function is used to insert one element into the list.
- remove(x), this function is used to remove given element from the list.
- get(i), this function is used to get element at position I
- replace(x, y), this function is used to replace x with y value

Example: Stack

A stack is a LIFO (last in, first out) list with the following operations: Push, Pop, Create(Init),Top, IsEmpty, Size. (Such a list is called a Signature or the Interface to the(ADT.)

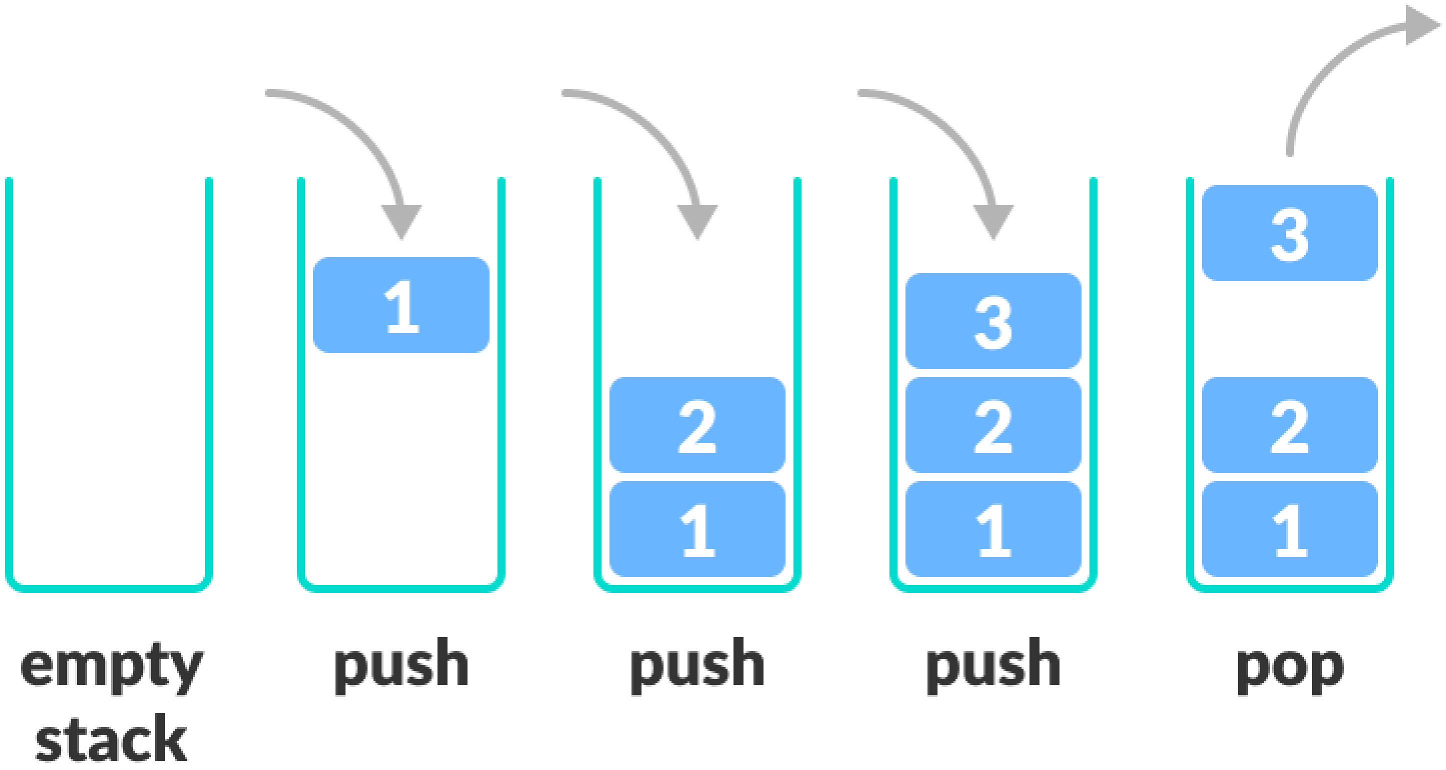
Example: Queue

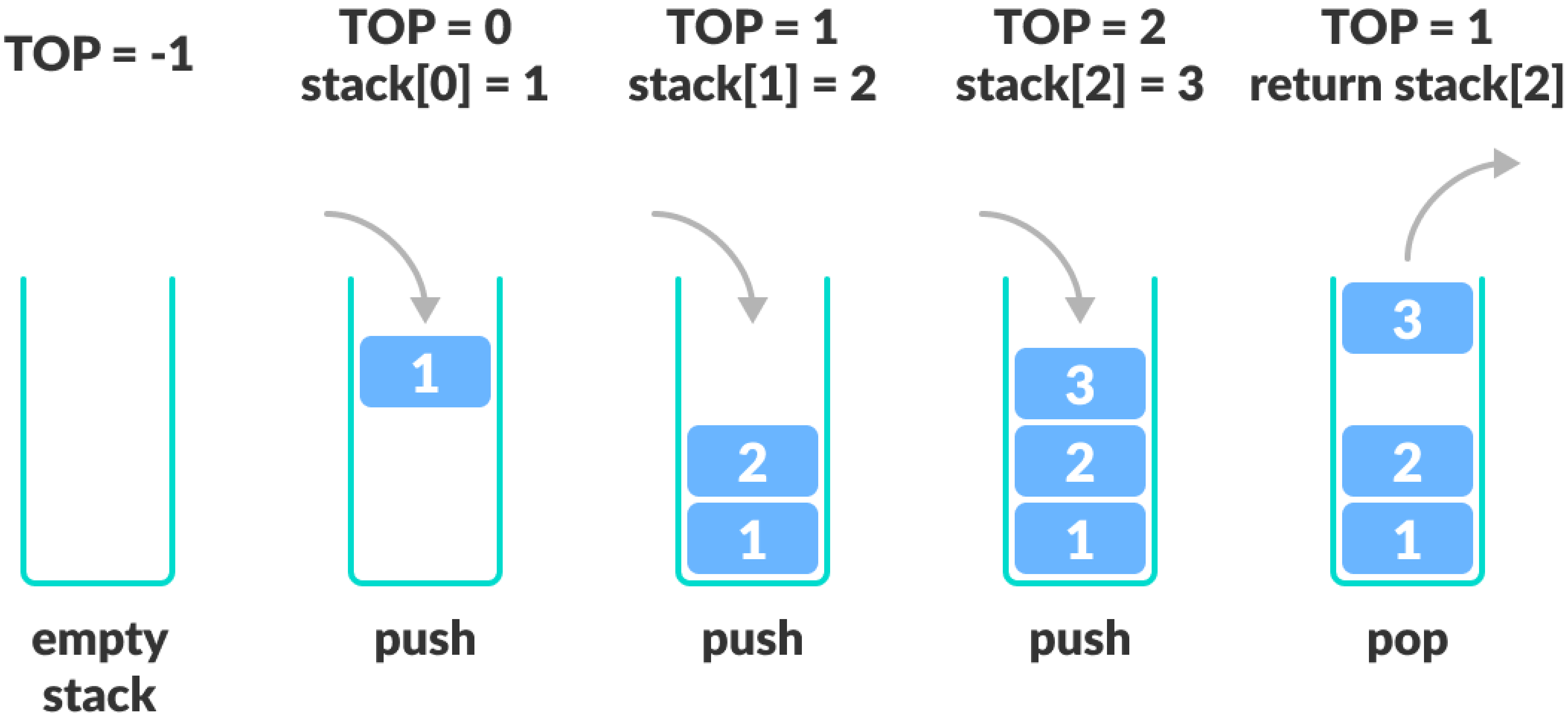
A Queue is a FIFO (first in, first out) list with the following operations: Enqueue, Dequeue, Size, Font.



2. **Stack ADT**

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.


1
2
1
3
2
1
3
2
1
empty stack
push
push
push
pop



TOP = -1

empty stack

TOP = 0
stack[0] = 1

push

TOP = 1
stack[1] = 2

push

TOP = 2
stack[2] = 3

push

TOP = 1
return stack[2]

pop

Stack ADT

Stack() - creates a new stack that is empty. It needs no parameters and returns an empty stack.

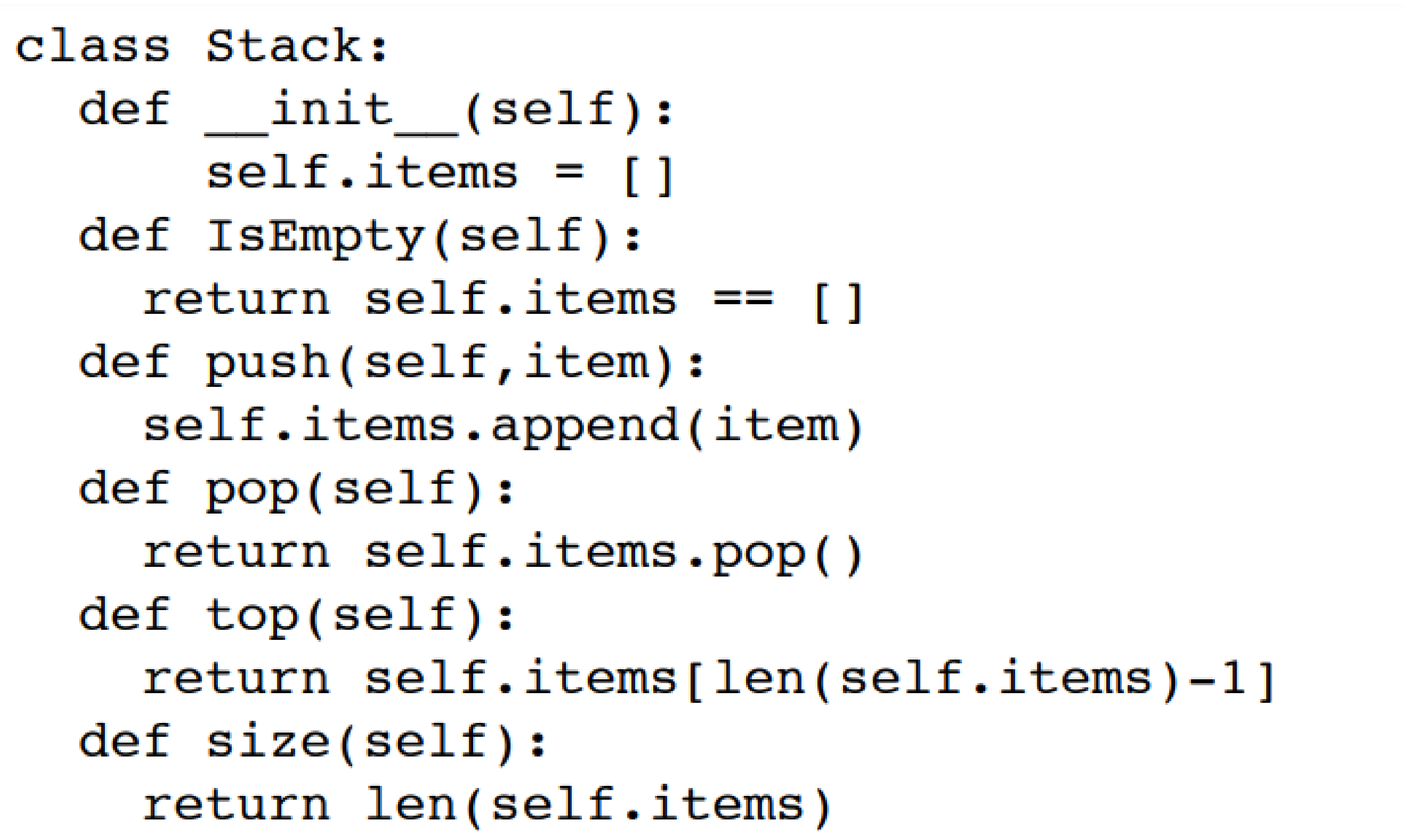
push(item) – adds a new item to the top of the stack. It needs the item and returns nothing.

pop() - removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

top() – returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

isEmpty() – tests to see whether the stack is empty. It needs no parameters and returns a Boolean value.

size() – returns the number of items on the stack. It needs no parameters and returns an integer.

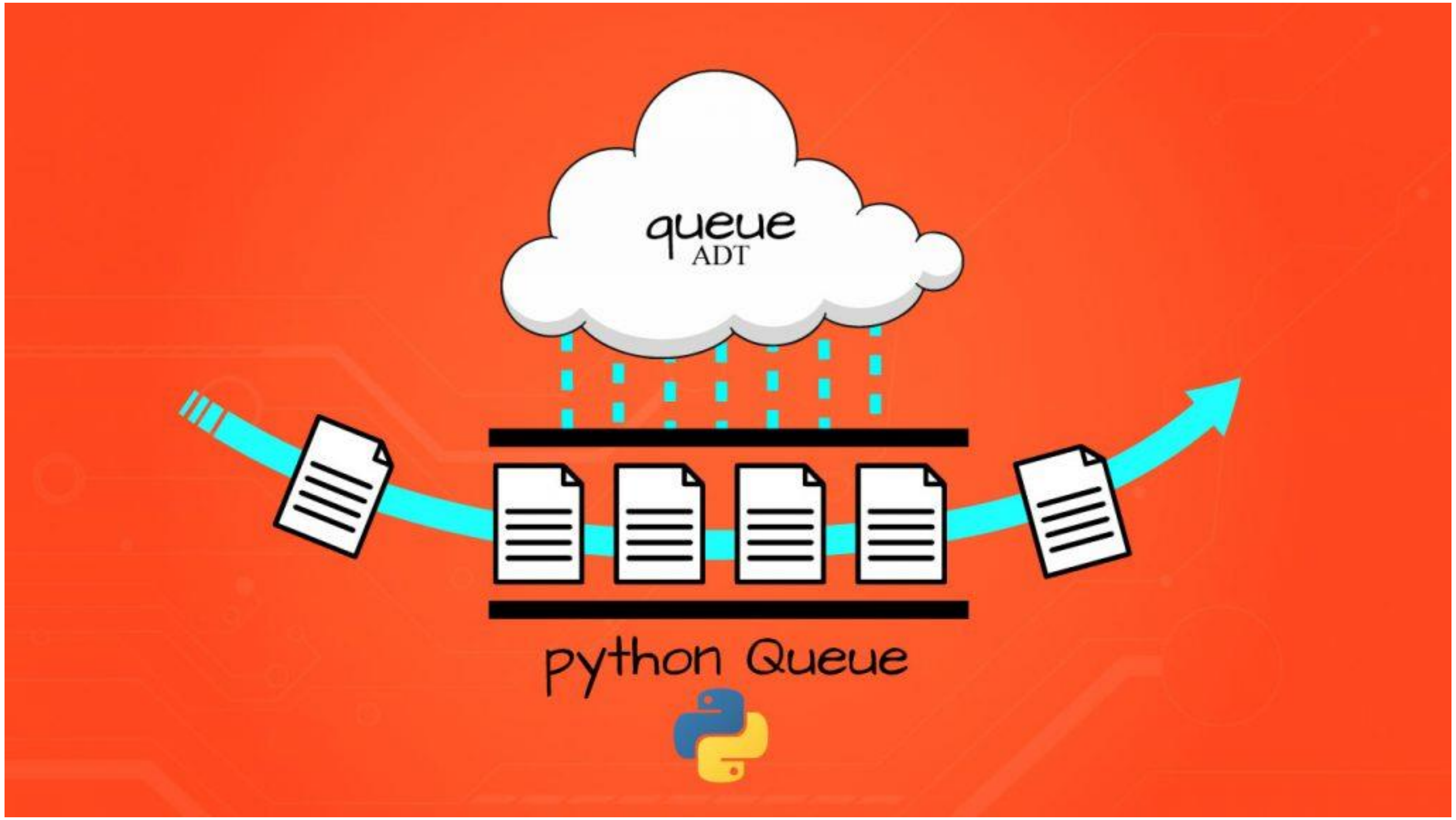


```
class Stack:
  def __init__(self):
      self.items = []
  def IsEmpty(self):
    return self.items == []
  def push(self,item):
    self.items.append(item)
  def pop(self):
    return self.items.pop()
  def top(self):
    return self.items[len(self.items)-1]
  def size(self):
    return len(self.items)
```

```python
def create_stack():
    stack = []
    return stack


# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0


# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)
```

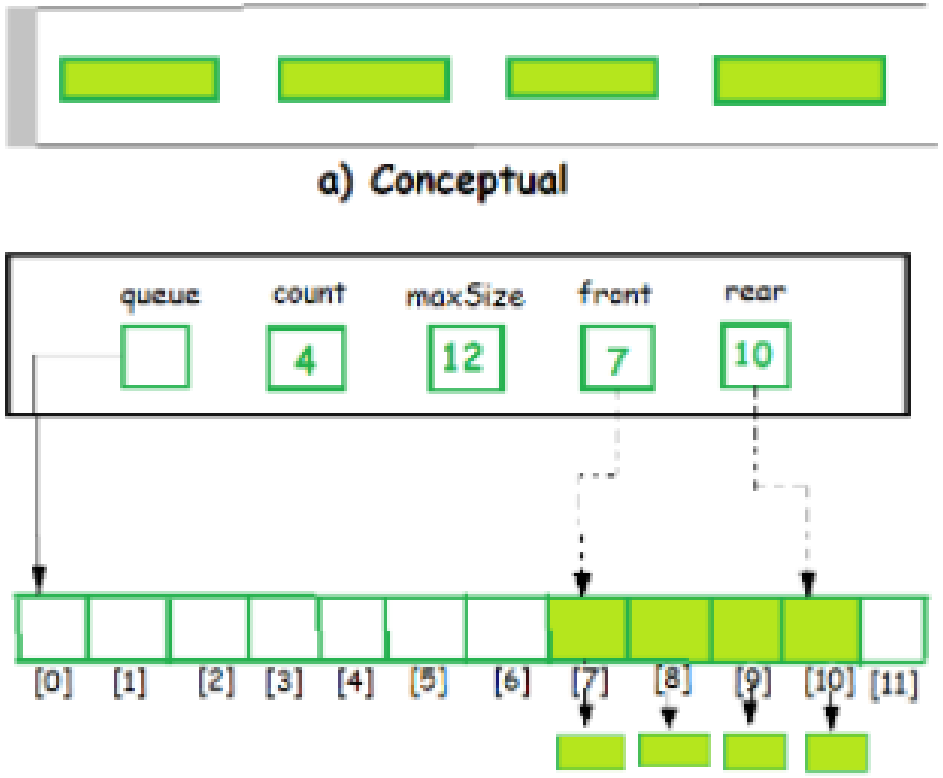
```python
# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"

    return stack.pop()


stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element: ['1', '2', '3']
```


queue
ADT
python Queue

3. **Queue ADT**

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.



a) Conceptual

b) Physical Structures

Queue ADT

Queue() – creates a new queue that is empty. It needs no parameters and returns an empty queue.

Enqueue(item) – adds a new item to the rear of the queue. It needs the item and returns nothing.

Dequeue() – removes the item from the front of the queue. It needs no parameters and returns the item. The queue is modified.

Front() – returns the front item from the queue but does not remove it. It needs no parameters. The queue is not modified.

isEmpty() – tests to see whether the queue is empty. It needs no parameters and returns a Boolean value.

size() – returns the number of items on the queue. It needs no parameters and returns an integer.

```
Class Queue:
  def __init__(self):
      self.items = []
  def IsEmpty(self):
    return self.items == []
  def enqueue(self,item):
    self.items.append(item)
  def dequeue(self):
    return self.items.pop(0)
  def front(self):
    return self.items[len(self.items)-1]
  def size(self):
    return len(self.items)
```

```python
class Queue:
    def __init__(self):
        self._items = []
    def add(self, item: Any) -> NoReturn:
        """Adds an item to the back of the queue"""
        self._items.append(item)
    def pop(self) -> Any:
        """Removes the first item of the queue"""
        return self._items.pop(0)
    def isEmpty(self) -> bool:
        """Checks if any items are in the queue"""
        return len(self._items) == 0
    def __str__(self):
        return str(self._items)
```

```python
if __name__ == '__main__':
    # Create a new Queue object
    q = Queue()
    # Have some items
    people = ['bob', 'alice', 'pat']
    # View starting queue
    print("Initial Queue:", q)
    # Add everyone to the queue
    for person in people:
        print("\tAdding:", person)
        q.add(person)
    # View full queue
    print("Filled Queue:", q)
    # Remove People, one-by-one
    while not q.isEmpty():
        next_person = q.pop()
        print("\tNext Person:", next_person)
    # Print resulting queue
    print("Final Queue:", q)
```

```
Initial Queue: []
     Adding: bob
     Adding: alice
     Adding: pat
Filled Queue: ['bob', 'alice', 'pat’]
     Next Person: bob
     Next Person: alice
     Next Person: pat
Final Queue: []
```

Classes

- Python is an object oriented programming language.

- A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together.

```
class sample:
    x=10
    print(x)
```

objects

- A unique instance of a data structure that's defined by its class.
- An object comprises both data members (class variables and instance variables) and methods.

```
class myclass:
    x=10
    y=20
    name="hello"
my=myclass()
print(my.x)
print(my.y)
print(my.name)
```

The __init__() Function

- All classes have a function called __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties

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

p1 = Person("John", 36)

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

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.

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

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

p1 = Person(“abdul", 18)
p1.myfunc()
```

The self Parameter

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- 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 function in the class:

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

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

p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

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

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

p1 = Person(“abdul", 18)

p1.age = 19

print(p1.age)
```

Delete Object Properties

- delete properties on objects by using the del keyword

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

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

p1 = Person(“abdul", 18)

del p1.age

print(p1.age)
```

Delete Objects

delete objects by using the del keyword

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

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

p1 = Person(“abdul", 18)

del p1

print(p1)
```

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.

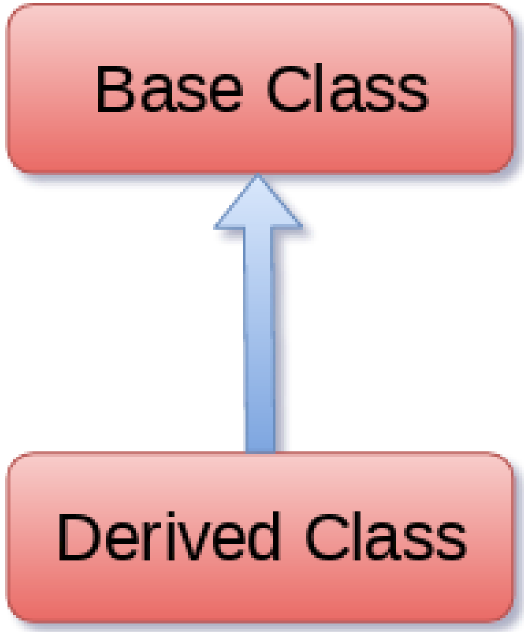
Create a Parent Class

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

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

x = Person(“abdur", “rahman")
x.printname()
```

Create a Child Class

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

```
class dept:
    def stud(self):
        print("hello welcome")
class cse(dept):
    def aids(self):
        print("thank you")
c=cse()
c.stud()
c.aids()
```



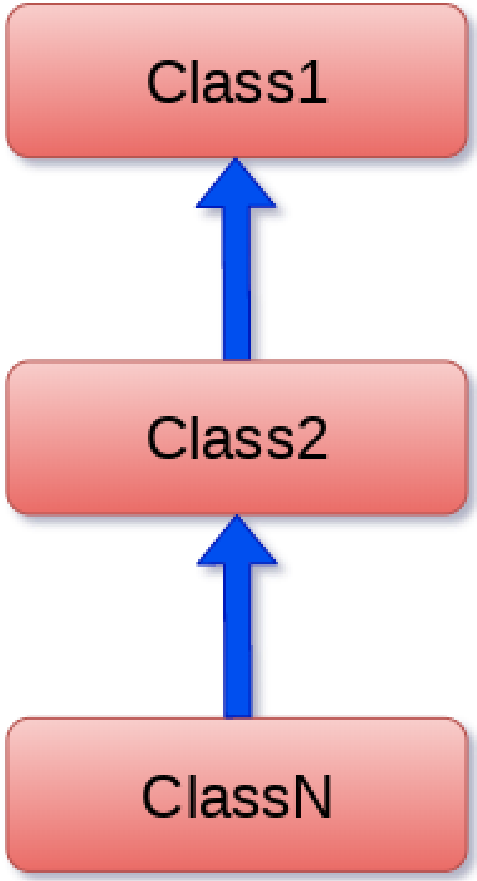
Base Class

Derived Class

```
class value:
    def first(self):
        self.a=int(input("Enter the a value"))
    def second(self):
        self.b=int(input("Enter the b value"))

class operation(value):
    def add(self):
        super().first()
        super().second()
        self.c=self.a+self.b
        self.c=self.a-self.b
        print("The sum is:",self.c)

c=operation()
c.add()
```

```
class value:
    def first(self):
        self.a=int(input("Enter the a value"))
        self.b=int(input("Enter the b value"))

class operation(value):
    def add(self):
        super().first()

        self.c=self.a+self.b
        self.z=self.a-self.b
        print("The sum is:",self.c)
        print("The subtraction is:",self.z)

c=operation()
c.add()
```

Multi-Level inheritance



- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

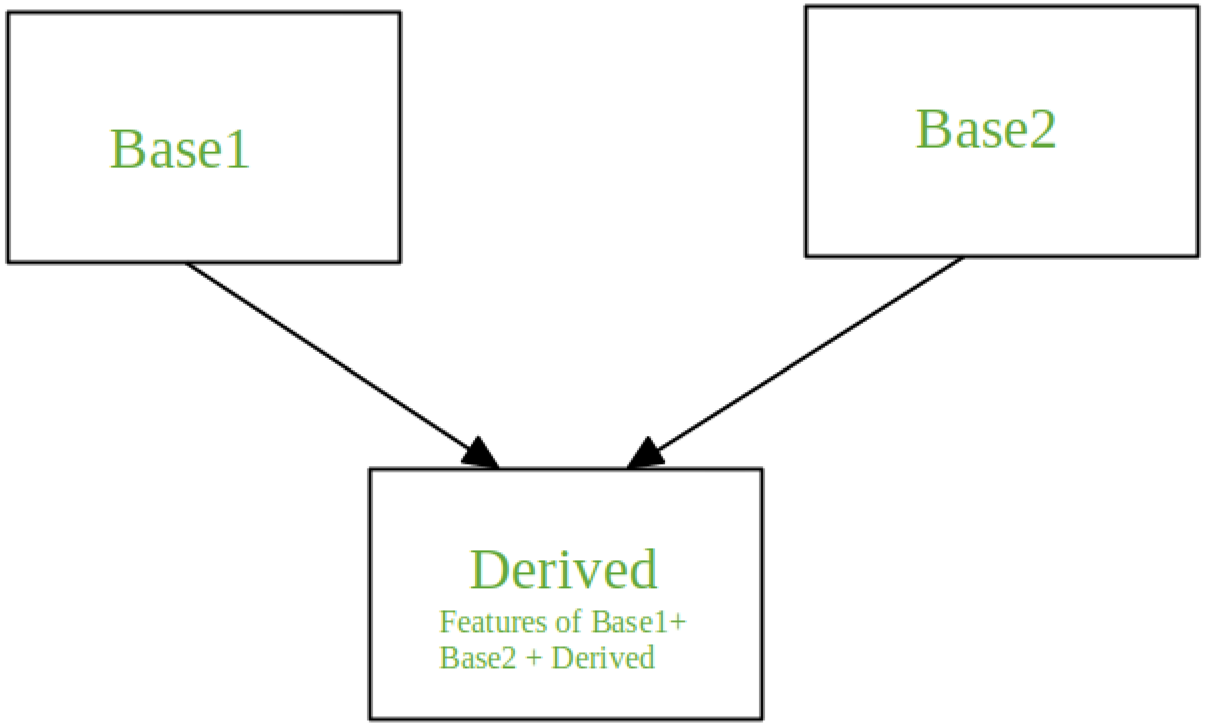
Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

```
class gp:
  def fgp(self):
    print("I am grand parent")
class p(gp):
  def fp(self):
    print("I am parent")
class c(p):
  def fc(self):
    print("I am child")
obj=c()
c.fc(None)
c.fp(None)
c.fgp(None)
```

Multiple Inheritance

When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.



```
class father:
  def __init__(self,a):
    self.a=int(input("enter the a value"))
  def ffun(self):
    print(self.a)

class mother:
  def __init__(self,b):
    self.b=int(input("enter the b value"))
  def mfun(self):
    print(self.b)
```

```
class child(father,mother):
  def __init__(self,a,b,c):
    father.__init__(self,a)
    mother.__init__(self,b)
    self.c=int(input("enter the c value"))
  def cfun(self):
    print(self.c)
    print("The result is", self.a+self.b+self.c)
obj=child(None,None,None)

obj.ffun()
obj.mfun()
obj.cfun()
```

```
enter the a value2
enter the b value3
enter the c value4
2 3 4
The result is 9
```

__init__() Function (Parent Class)

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

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

class Student(Person):
  pass

x = Student(“abdur", “rahman")
x.printname()
```

__init__() Function (child class)

__init__() function to the child class (instead of the pass keyword)

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

```
class person:
    def __init__(self, fname,lname):
        self.fname = fname
        self.lname = lname

    def printname(self):
        print(self.fname,self.lname)

class student(person):
    def __init__(self,fname,lname):
        person.__init__(self, fname, lname)

p=student("abdul","rahman")
p.printname()
```

the super() Function

super() function that will make the child class inherit all the methods and properties from its parent

```
class person:
    def __init__(self, fname,lname):
        self.fname = fname
        self.lname = lname

    def printname(self):
        print(self.fname,self.lname)

class student(person):
    def __init__(self,fname,lname):
        super().__init__( fname, lname)

p=student("abdul","rahman")
p.printname()
```

Add Properties

```
class person:
    def __init__(self, fname,lname):
        self.fname = fname
        self.lname = lname

    def printname(self):
        print(self.fname,self.lname)

class student(person):
    def __init__(self,fname,lname):
        super().__init__( fname, lname)
        self.dept="AIDS"

p=student("abdul","rahman")
p.printname()
print(p.dept)
```

Add Methods

```
class person:
    def __init__(self, fname,lname):
        self.fname = fname
        self.lname = lname

    def printname(self):
        print(self.fname,self.lname)

class student(person):
    def __init__(self,fname,lname,depts):
        super().__init__(fname, lname)
        self.dept=depts

    def year(self):
print("welcome",self.fname,self.lname,self.dept)

p=student("abdul","rahman","AIDS")
p.year()
```

Substitution Principles

- In software development, **Object-Oriented Design** plays a crucial role when it comes to writing flexible, scalable, maintainable, and reusable code.
- There are so many benefits of using OOD but every developer should also have the knowledge of the SOLID principle for good object-oriented design in programming.
- The SOLID principle was introduced by Robert C. Martin, also known as Uncle Bob and it is a coding standard in programming.

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle
3. Liskov’s Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

1. **Single Responsibility Principle:**

This principle states that “*a class should have only one reason to change*” which means every class should have a single responsibility or single job or single purpose.

2. **Open/Closed Principle:**

This principle states that “*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*” which means you should be able to extend a class behavior, without modifying it.

3. Liskov’s Substitution Principle:

The principle was introduced by Barbara Liskov in 1987 and according to this principle “*Derived or child classes must be substitutable for their base or parent classes*“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

4. Interface Segregation Principle:

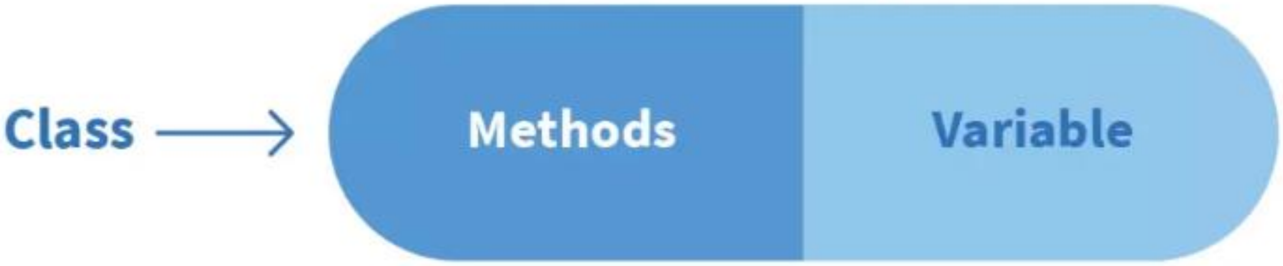
This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “do not force any client to implement an interface which is irrelevant to them“.

5. Dependency Inversion Principle:

- High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

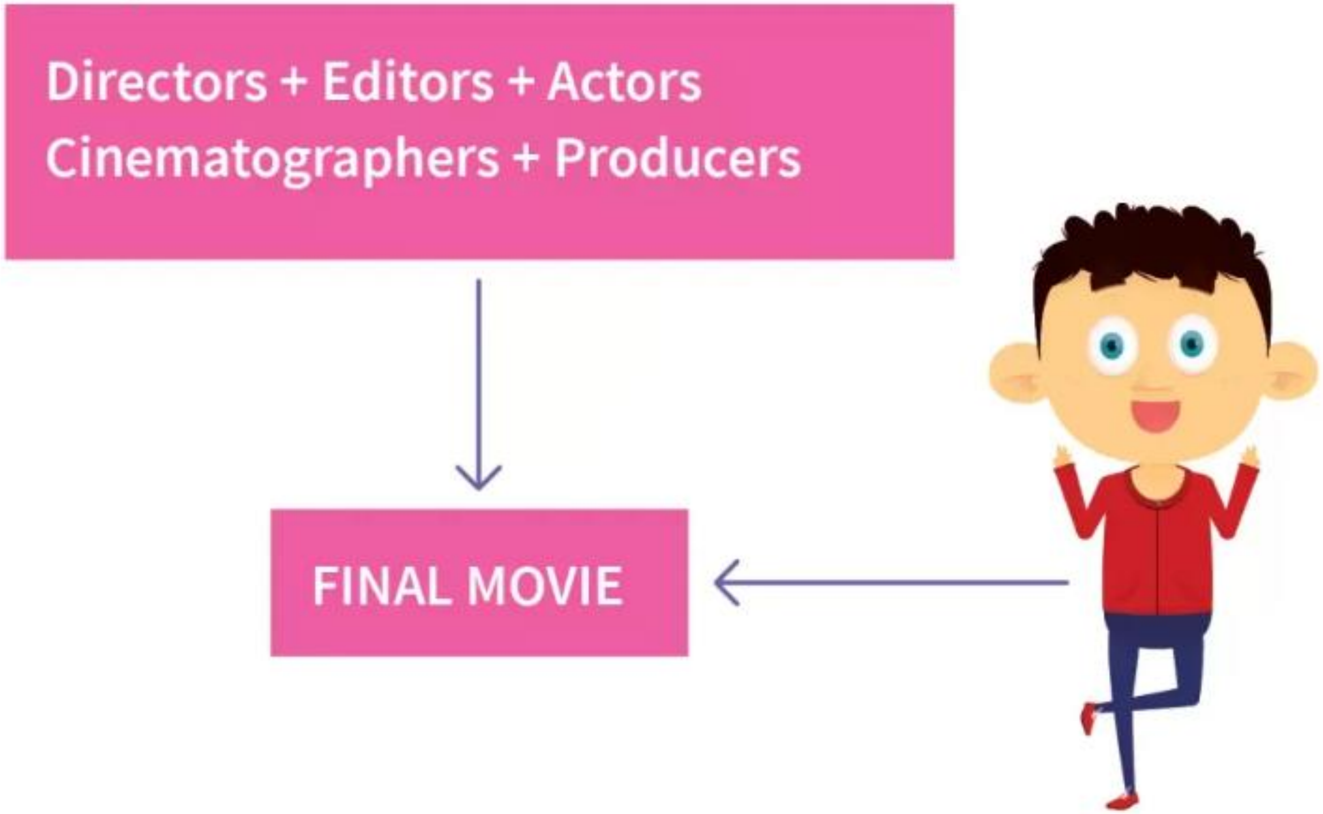
```
class calculator():

def calculate(self, a, b): # returns a number
return a * b

results = [

calculator().calculate(2, 2),
calculator().calculate(3, 4), ]

print(results)
```

ENCAPSULATION



Class → Methods | Variable

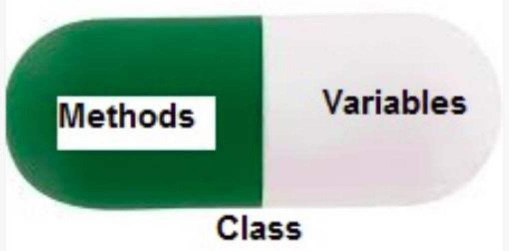
Encapsulation



Directors + Editors + Actors
Cinematographers + Producers

FINAL MOVIE

Encapsulation

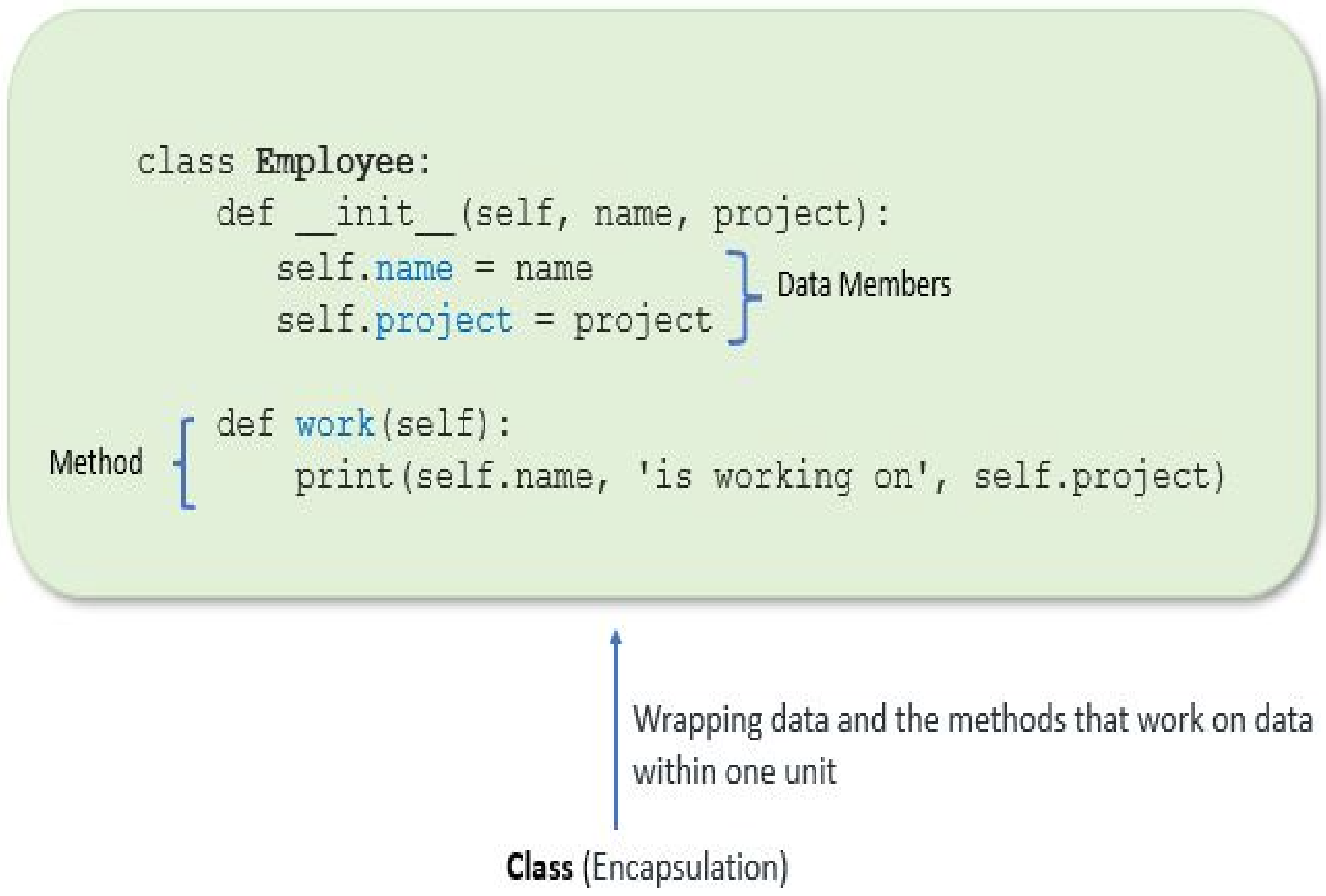


- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- It describes the idea of wrapping data and the methods that work on data within one unit.
- This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as **private variable**.

Encapsulation in Python describes the concept of **bundling data and methods within a single unit**.

So, for example, when you create a class, it means you are implementing encapsulation.

A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.



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

    def work(self):
        print(self.name, 'is working on', self.project)
```

Data Members
Method
Wrapping data and the methods that work on data within one unit
Class (Encapsulation)

Implement encapsulation using a class

The advantages of Encapsulation in Python

1. Encapsulation provides well-defined, readable code

2. Prevents Accidental Modification or Deletion

3. Encapsulation provides security

1.Public Members

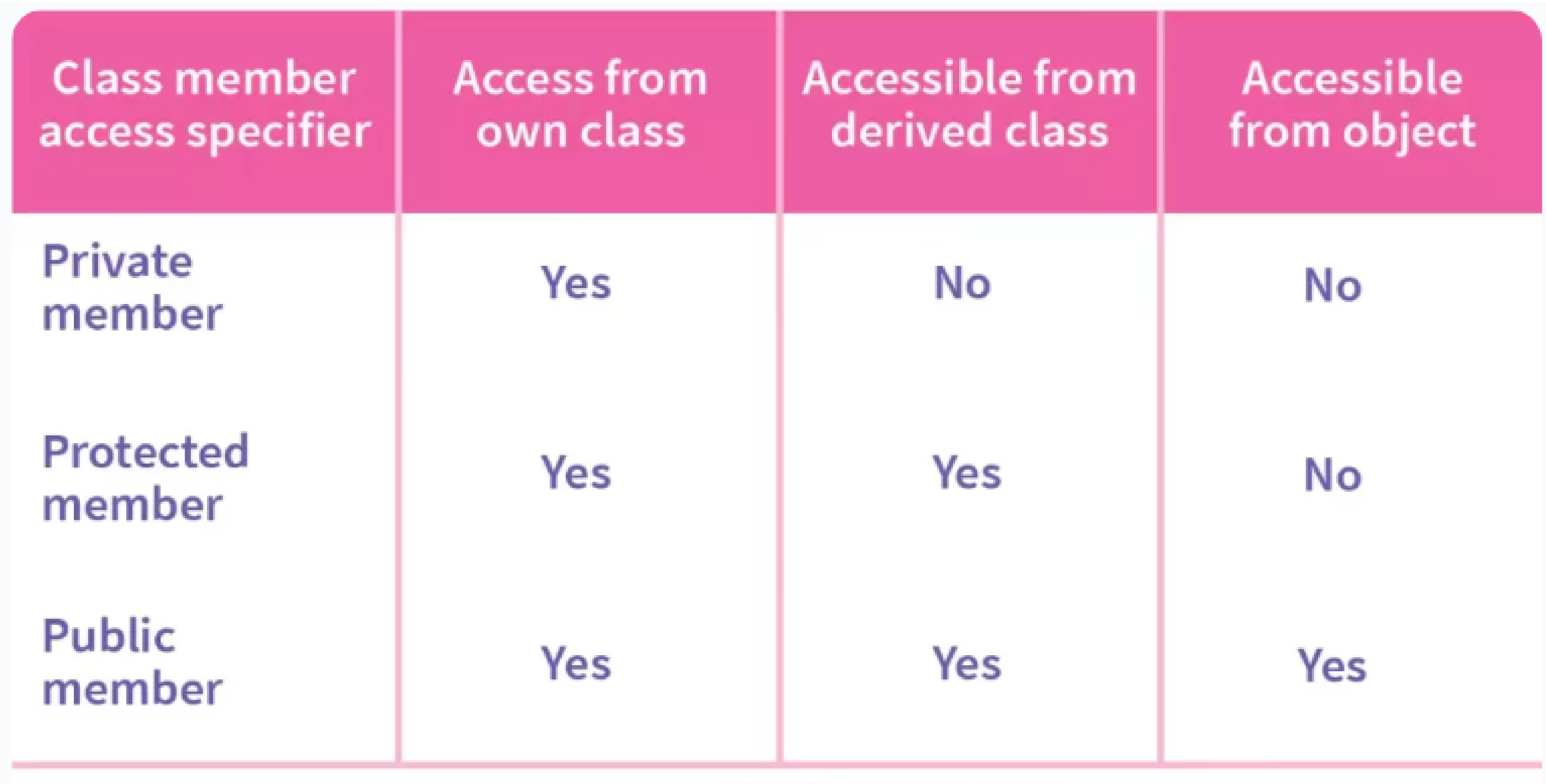
The member of class are define public which are easily accessible from any part of the program but within a program. In class all the member and member function by default is public.

2. Protected Members

- The members of a class declared as protected are not accessible by those outside of the class, however, it is accessible only by inherited class or child class.
- To create protected data inside the class, we use a single underscore (_) before the data member.

3. Private Members

- The members of the class declared as private are accessible inside the class only. It is not accessible outside of the class even in any child class. To create private data inside the class,
- To create private data members inside the class, we use a double underscore (__) before the data member.



Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

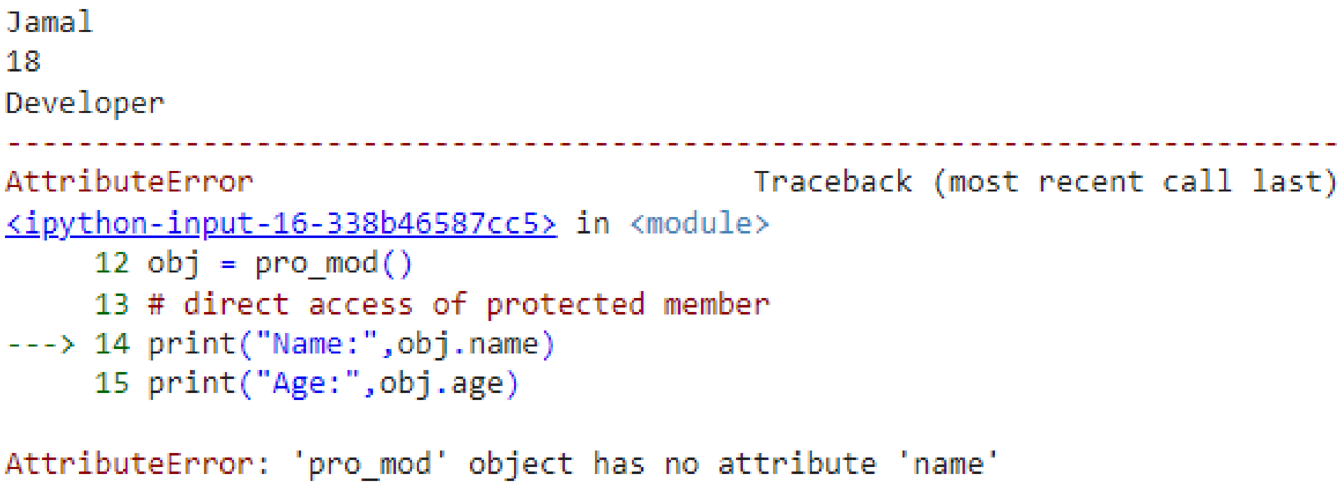
Illustrating Public members & Public access modifier

```python
class pub_mod:
    # constructor
    def __init__(self, name, age):
        self.name = name;
        self.age = age;

    def Age(self):
        # accessing public data member
        print("Age: ", self.age)
# creating object
obj = pub_mod("Jamal", 18);
# accessing public data member
print("Name: ", obj.name)
# calling public member function of the class
obj.Age()
```

```
Name:  Jamal
Age:  18
```

Illustrating Protected members & Protected access modifier

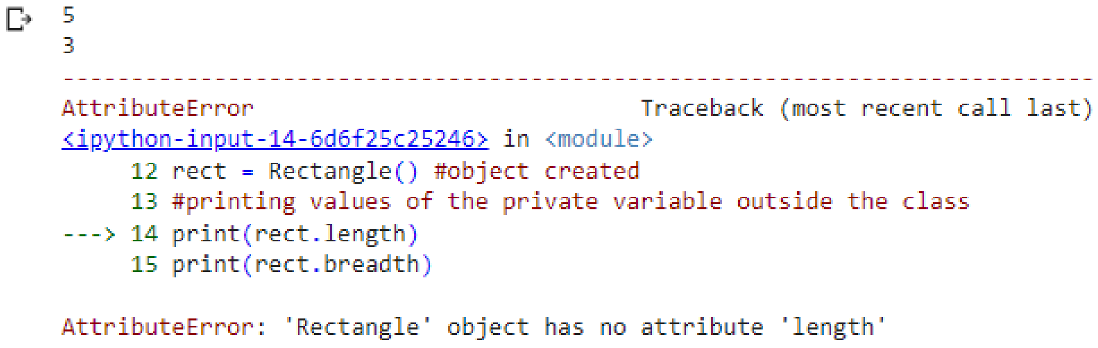
```python
class details:
    _name="Jamal"
    _age=18
    _job="Developer"
class pro_mod(details):
    def __init__(self):
        print(self._name)
        print(self._age)
        print(self._job)

# creating object of the class
obj = pro_mod()
# direct access of protected member
print("Name:",obj.name)
print("Age:",obj.age)
```



```
Jamal
18
Developer
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-338b46587cc5> in <module>
     12 obj = pro_mod()
     13 # direct access of protected member
---> 14 print("Name:",obj.name)
     15 print("Age:",obj.age)

AttributeError: 'pro_mod' object has no attribute 'name'
```

Illustrating Private members & Private access modifier

```python
class Rectangle:
  __length = 0 #private variable
  __breadth = 0#private variable
  def __init__(self):
    #constructor
    self.__length = 5
    self.__breadth = 3
    #printing values of the private variable within the class
    print(self.__length)
    print(self.__breadth)

rect = Rectangle() #object created
#printing values of the private variable outside the class
print(rect.length)
print(rect.breadth)
```



```
5
3
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-6d6f25c25246> in <module>
     12 rect = Rectangle() #object created
     13 #printing values of the private variable outside the class
---> 14 print(rect.length)
     15 print(rect.breadth)

AttributeError: 'Rectangle' object has no attribute 'length'
```

Data hiding or Information Hiding

- Data hiding is also known as **data encapsulation** and it is the process of hiding the implementation of specific parts of the application from the user.
- An object's attributes may or may not be visible outside the class definition.
- Data hiding combines members of the class thereby restricting direct access to the members of the class.
- we need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

Advantages of Data Hiding

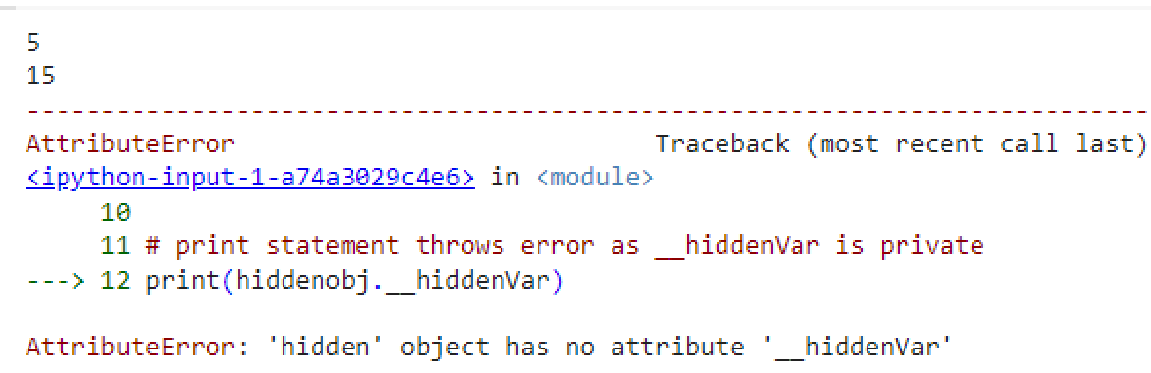
- The class objects are disconnected from the irrelevant data.
- It enhances the security against hackers that are unable to access important data.
- It isolates objects as the basic concept of OOP.
- It helps programmers from incorrect linking to corrupt data.
- We can isolate the object from the basic concept of OOP.
- It provides high security which stops damage to violating data by hiding it from the public.

Disadvantages of Data Hiding

- Sometimes programmers need to write the extra lien of the code.
- The data hiding prevents linkage that acts as a link between visible and invisible data makes the object faster.
- It forces programmers to write extra code to hide important data from common users.

Data hiding is performed by declaring the variable in the class as private

```
class hidden:
    # declaring private member of class
    __hiddenVar = 0
    def sum(self, counter):
        self.__hiddenVar += counter
        print (self.__hiddenVar)
hiddenobj = hidden()
hiddenobj.sum(5)
hiddenobj.sum(10)

# print statement throws error as __hiddenVar is private
print(hiddenobj.__hiddenVar)
```



```
5
15
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-a74a3029c4e6> in <module>
     10
     11 # print statement throws error as __hiddenVar is private
---> 12 print(hiddenobj.__hiddenVar)

AttributeError: 'hidden' object has no attribute '__hiddenVar'
```

The hidden data can be accessed directly outside the class

```
class hidden:
# declaring hiddenVar private by using __
   __hiddenVar = 0
   def sum(self, counter):
      self.__hiddenVar += counter
      print (self.__hiddenVar)
hiddenobj = hidden()
hiddenobj.sum(5)
hiddenobj.sum(10)

# adding class name before variable to access the variable
outside the class
print(hiddenobj._hidden__hiddenVar)
```

```
5
15
15
```

Data hiding by using both private and protected members of the class

```
class Employee:
   # Hidden members of the class
   __password = 'private12345' # Private member
   _id = '12345' # Protected member
   def Details(self):
      print("ID: ",(self._id))
      print("Password: ",(self.__password)+"")
hidden = Employee()
hidden.Details()
print(hidden._Employee__password)
```

```
ID: 12345
Password: private12345
private12345
```

Python Standard Libraries

Python Built-in Functions

abs()

returns absolute value of a number

any()

Checks if any Element of an Iterable is True

all()

returns true when all elements in iterable is true

ascii()

Returns String Containing Printable Representation

bin()

converts integer to binary string

bool()

Converts a Value to Boolean

bytearray()

returns array of given byte size

Etc…..

Python Dictionary Methods

clear()

Removes all Items

copy()

Returns the Shallow Copy of a Dictionary

fromkeys()

creates dictionary from given sequence

get()

Returns Value of The Key

Etc…

Python List Methods

append()

Add a single element to the end of the list

extend()

adds iterable elements to the end of the list

insert()

insert an element to the list

Etc…

Python Set Methods

remove()

removes the specified element from the set

add()

adds element to a set

Etc…

Python String Methods

capitalize()

Converts first character to Capital Letter

center()

Pads string with specified character

Etc…

Python Tuple Methods

count()

returns count of the element in the tuple

index()

returns the index of the element in the tuple

Packages

- The package is **a simple directory having collections of modules**.
- A package is **a directory of Python modules that contains an additional __init__.py file**, which distinguishes a package from a directory that is supposed to contain multiple Python scripts.
    - NumPy.
    - pandas.
    - Matplotlib.
    - Seaborn.
    - scikit-learn.
    - Requests.
    - NLTK.

Built-in Modules

- os module
- random module
- math module
- time module
- sys module
- collections module
- statistics module

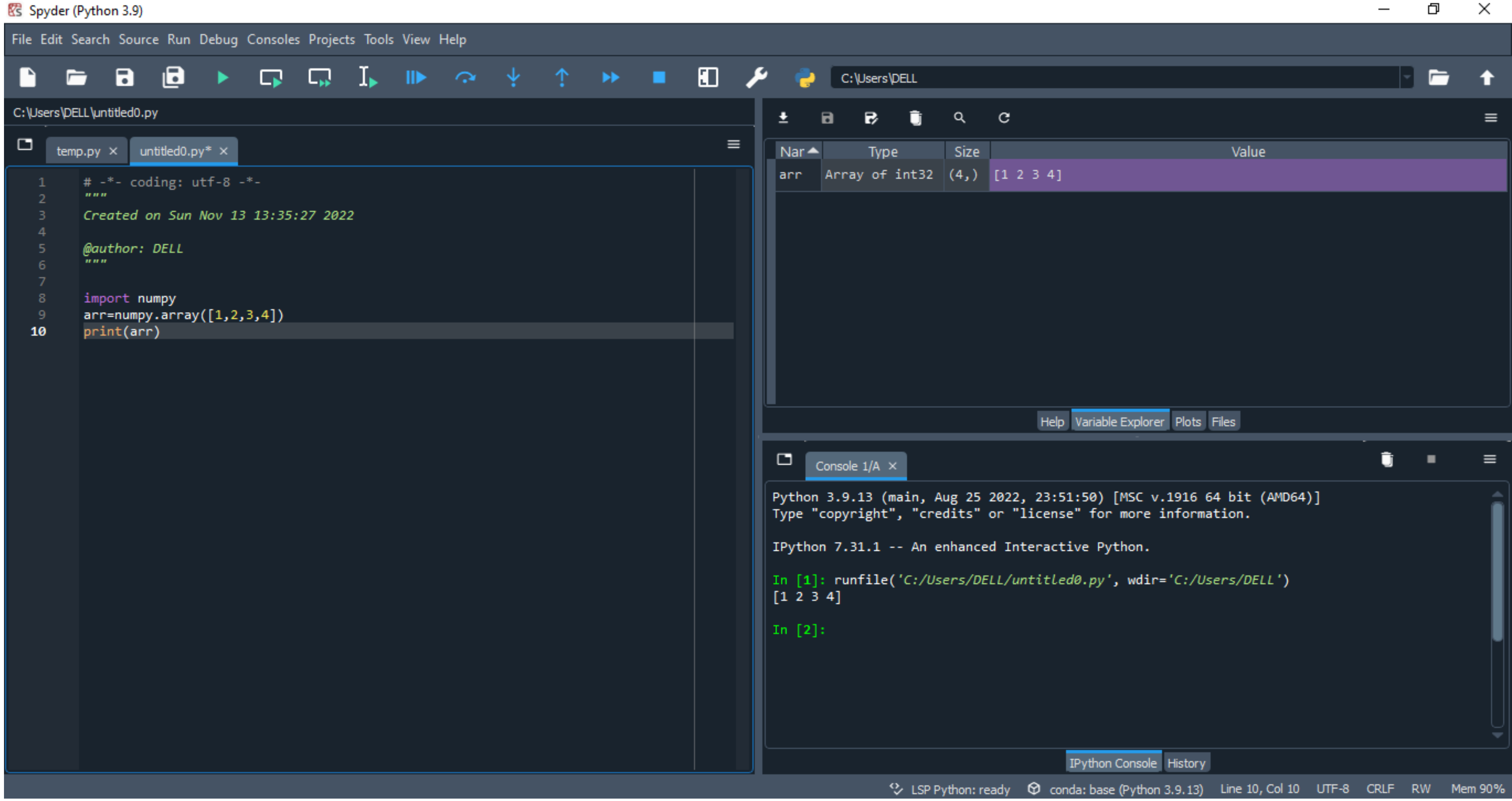
https://www.knowledgehut.com/tutorials/python-tutorial/python-built-in-modules

User-defined Modules

```
import math
math.pi
3.141592653589793

math.e
2.718281828459045

math.radians(30)
0.5235987755982988

math.degrees(math.pi/6)
29.999999999999996

math.sin(60)
-0.3048106211022167

math.pow(2,4)
16.0
```

```
import numpy

arr=numpy.array([1,2,3])

print(arr)
```

```
[1 2 3]
```



```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 13 13:35:27 2022

@author: DELL
"""

import numpy
arr=numpy.array([1,2,3,4])
print(arr)
```

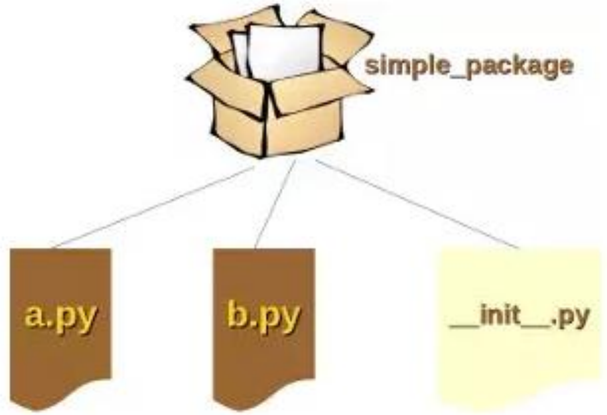
Nar	Type	Size	Value
arr	Array of int32	(4,)	[1 2 3 4]

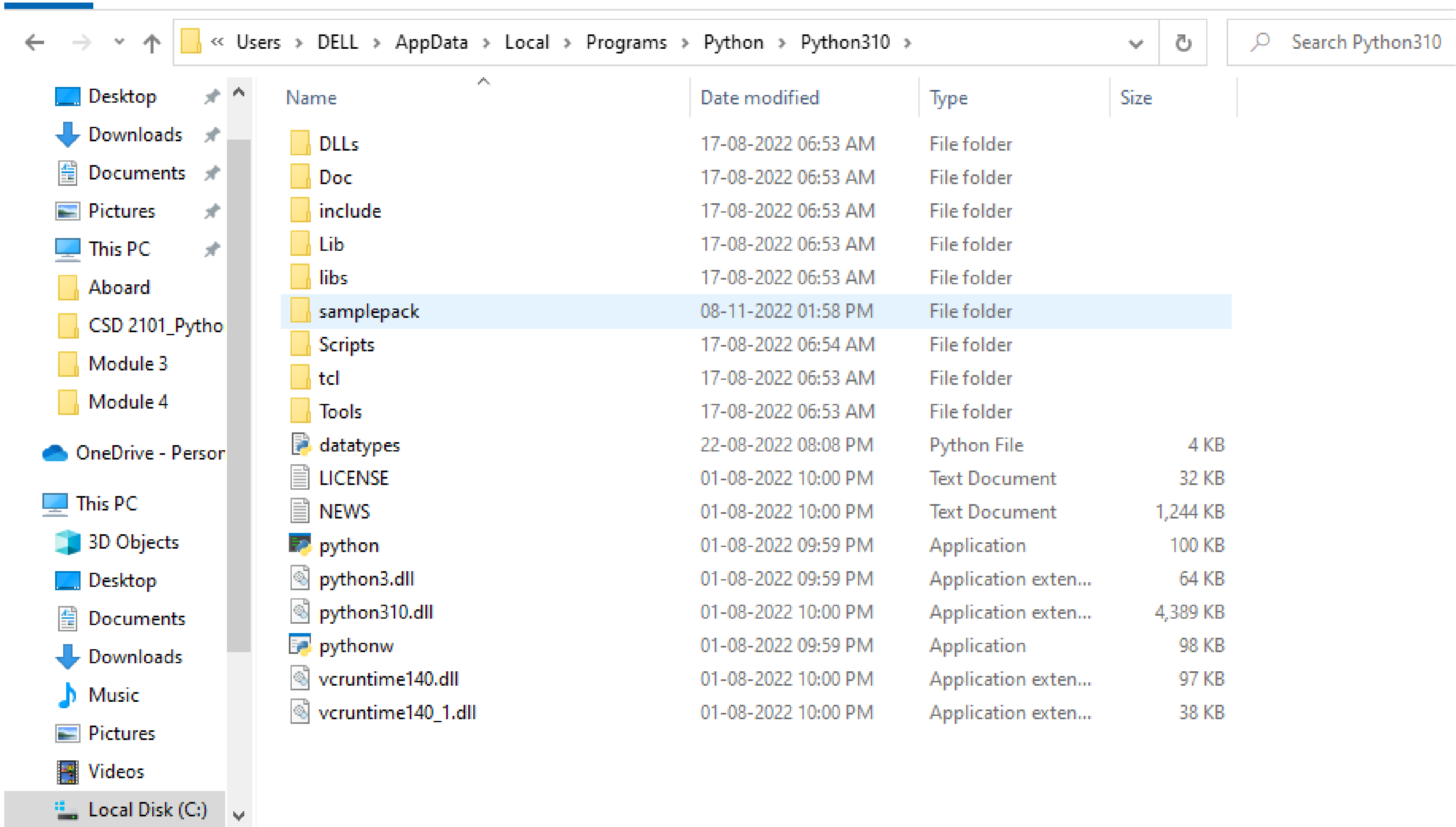
```
Python 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.31.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/DELL/untitled0.py', wdir='C:/Users/DELL')
[1 2 3 4]

In [2]:
```

Packages

A **python package** is a collection of modules. Modules that are related to each other are mainly put in the same package. When a module from an external package is required in a program, that package can be imported and its modules can be put to use.

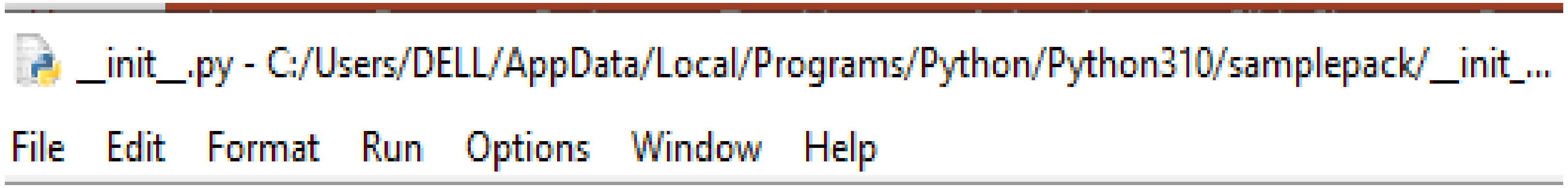
A package is a directory of Python modules that contains an additional __init__.py file, which distinguishes a package from a directory that is supposed to contain multiple Python scripts. Packages can be nested to multiple depths if each corresponding directory contains its own __init__.py file.



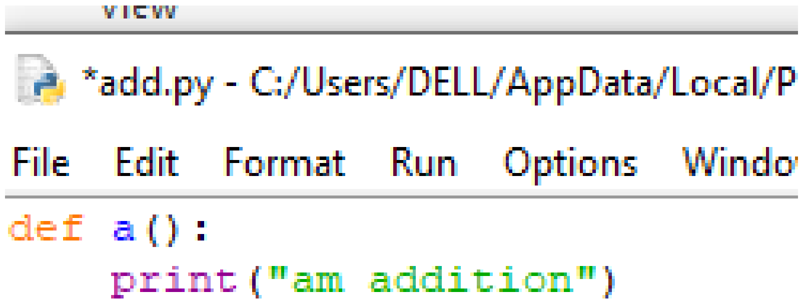


Name	Date modified	Type	Size
DLLs	17-08-2022 06:53 AM	File folder	
Doc	17-08-2022 06:53 AM	File folder	
include	17-08-2022 06:53 AM	File folder	
Lib	17-08-2022 06:53 AM	File folder	
libs	17-08-2022 06:53 AM	File folder	
samplepack	08-11-2022 01:58 PM	File folder	
Scripts	17-08-2022 06:54 AM	File folder	
tcl	17-08-2022 06:53 AM	File folder	
Tools	17-08-2022 06:53 AM	File folder	
datatypes	22-08-2022 08:08 PM	Python File	4 KB
LICENSE	01-08-2022 10:00 PM	Text Document	32 KB
NEWS	01-08-2022 10:00 PM	Text Document	1,244 KB
python	01-08-2022 09:59 PM	Application	100 KB
python3.dll	01-08-2022 09:59 PM	Application exten...	64 KB
python310.dll	01-08-2022 10:00 PM	Application exten...	4,389 KB
pythonw	01-08-2022 09:59 PM	Application	98 KB
vcruntime140.dll	01-08-2022 10:00 PM	Application exten...	97 KB
vcruntime140_1.dll	01-08-2022 10:00 PM	Application exten...	38 KB

Create the folder

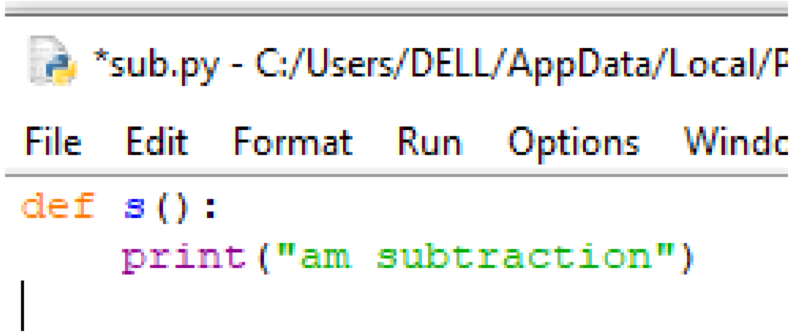

init.py - C:/Users/DELL/AppData/Local/Programs/Python/Python310/samplepack/_init_...
File Edit Format Run Options Window Help

create **__init__ .py** file


*add.py - C:/Users/DELL/AppData/Local/P
File Edit Format Run Options Windo

```python
def a():
    print("am addition")
```

create **add .py** file

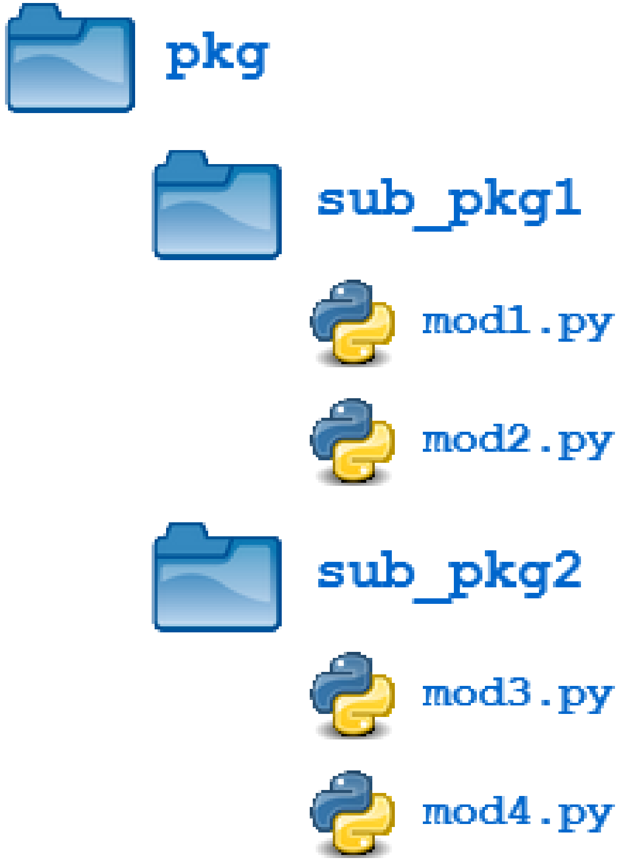

*sub.py - C:/Users/DELL/AppData/Local/P
File Edit Format Run Options Windc

```python
def s():
    print("am subtraction")
```

create **sub .py** file

```
>>>from samplepack import add,sub
>>>add.a()
am addition
>>>sub.s()
am subtraction

>>>import samplepack
>>>samplepack.add.a()
am addition
>>>samplepack.sub.s()
am subtraction
```

Subpackages



```
import subpack.Pack1.addition

subpack.Pack1.addition.adds()

enter the a value10

enter the b value10

20.0
```

Thank You