## 1 .1 Introduction to Object Oriented Programming:

**Object-oriented programming** (OOP) is a programming paradigm based on the concept of "**objects**". The object contains both data and code: Data in the form of properties (often known as attributes), and code, in the form of methods (actions object can perform).

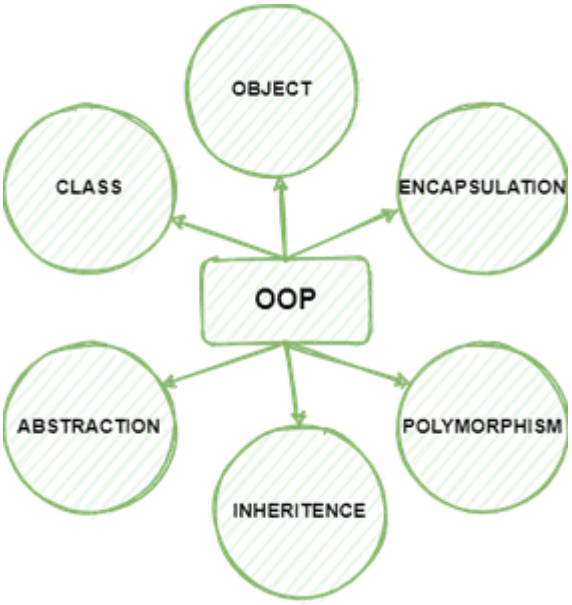
An object-oriented paradigm is to design the program using classes and objects. Python programming language supports different programming approaches like functional programming, modular programming. One of the popular approaches is object-oriented programming (OOP) to solve a programming problem is by creating objects.

**Difference between Object-Oriented and Procedural Oriented Programming:**

| Object-Oriented Programming (OOP) | Procedural-Oriented Programming (Pop) |
|---|---|
| It is a bottom-up approach | It is a top-down approach |
| Program is divided into objects | Program is divided into functions |
| Makes use of Access modifiers<br><br>'public', private', protected' | Doesn't use Access modifiers |
| It is more secure | It is less secure |
| Object can move freely within member functions | Data can move freely from function to function within programs |
| It supports inheritance | It does not support inheritance |

## 1.2 Features / Main Concepts of OOPS(Object Oriented Programming):

1. **Class & Objects**
2. **Inheritance**
3. **Data Encapsulation**
4. **Data Abstraction**
5. **Polymorphism**




**Python OOP concepts**

An object has the following two characteristics:

- Attribute
- Behavior

For example, A Car is an object, as it has the following properties:

- name, price, color as attributes
- breaking, acceleration as behavior

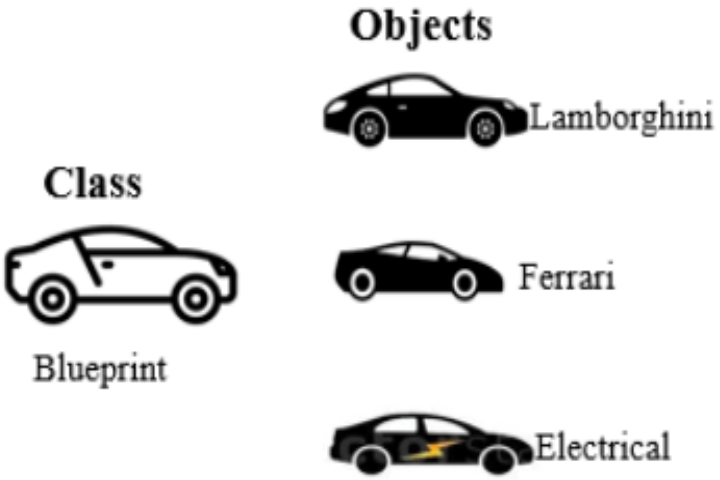
One important aspect of OOP in Python is to create **reusable code** using the concept of inheritance. This concept is also known as DRY (Don't Repeat Yourself).

### *1.3 Class and Objects:*

In Python, everything is an object. A **class is a blueprint for the object**. To create an object we require a model or plan or blueprint which is nothing but class.

For example, you are creating a vehicle according to the Vehicle blueprint (template). The plan contains all dimensions and structure. Based on these descriptions, we can construct a car, truck, bus, or any vehicle. Here, a car, truck, bus are objects of Vehicle class

A class contains the properties (attribute) and action (behavior) of the object. Properties represent variables, and the methods represent actions. Hence class includes both variables and methods.

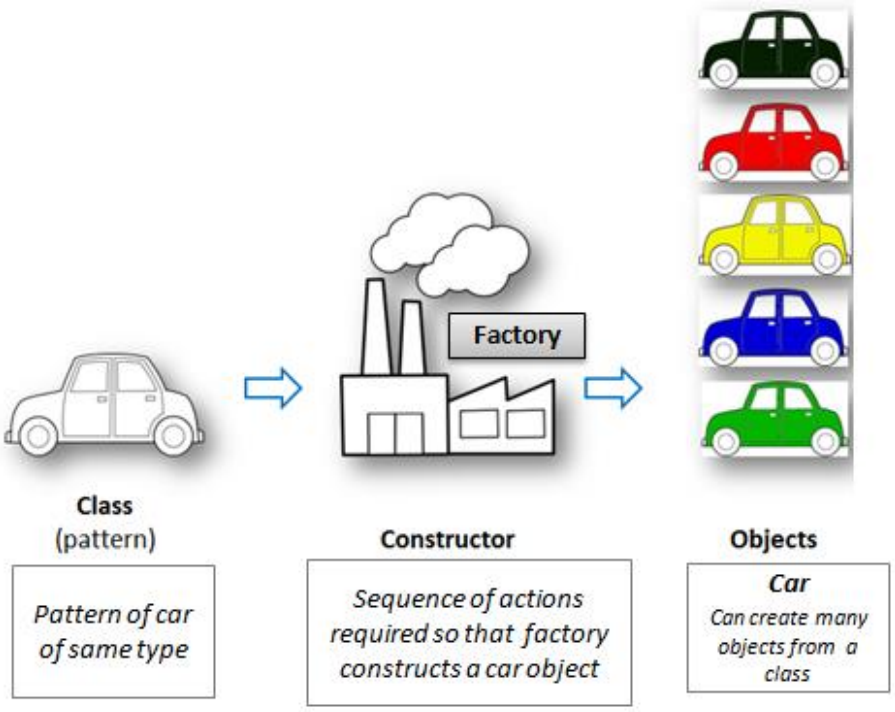


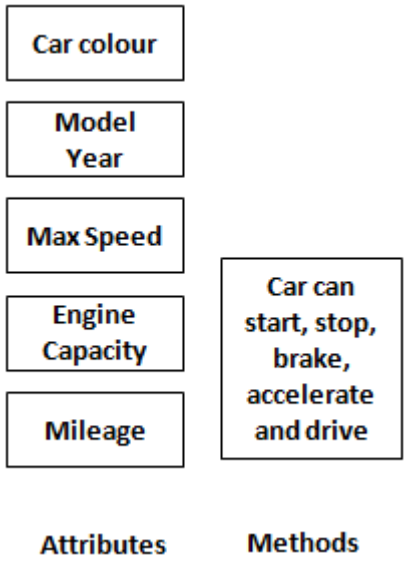

**Python Class and Objects**

**Object is an instance of a class**. The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behaviour. It may be any real-world object like the mouse, keyboard, laptop, etc.

**Let us see one more examples on class and object:**

Class is a architecture of the object. It is a proper description of the attributes and methods of a class. For example, design of a car of same type is a class. You can create many objects from a class. Like you can make many cars of the same type from a design of car.






**There are many real-world examples of classes as explained below -**

- **Recipe of Omelette is a class**. Omelette is an object.
- **Bank Account Holder is a class**. Attributes are First Name, Last Name, Date of Birth, Profession, Address etc. Methods can be "Change of address", "Change of Profession", " Change of last name" etc. "Change of last name" is generally applicable to women when they change their last name after marriage
- **Dog is a class**. Attributes are Breed, Number of legs, Size, Age, Color etc. Methods can be Eat, Sleep, Sit, Bark, Run etc.

In python, we can create a class using the keyword **class**. Method of class can be defined by keyword def. It is similar to a normal function but it is defined within a class and is a function of class. The first parameter in the definition of a method is always **self** and method is called without the parameter **self.**

### *1.4. Inheritance:*

Acquiring the properties of one class to another class is called Inheritance. In layman's terms, the attributes that you inherit from your parents are a simple illustration of inheritance. Parent classes, in other words, extend properties and behaviors to child classes.

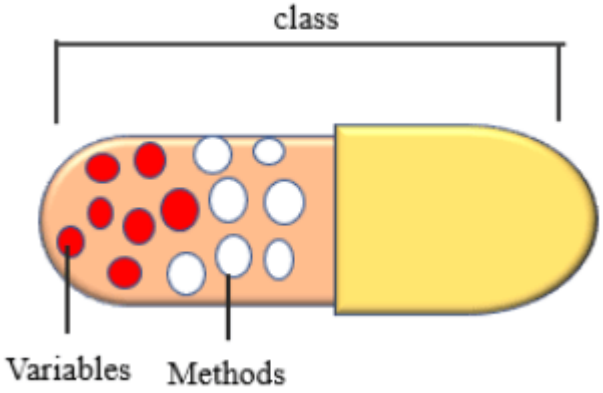
Inheritance is the procedure in which one class inherits the attributes and methods of another class. The interesting thing is, along with the inherited properties and methods, a child class can have its own properties and methods.

**Feature of Inheritance are**

1.Code Reusabiity
2. Easy to add new features
3. Overriding

### *1.5 Encapsulation:*

In Python, encapsulation is a method of wrapping data and functions into a single entity. For example, A class encapsulates all the data ( methods and variables). Encapsulation means the internal representation of an object is generally hidden from outside of the object's definition.




**Python Encapsulation**

**Need of Encapsulation**

Encapsulation acts as a protective layer. We can restrict access to methods and variables from outside, and It can prevent the data from being modified by accidental or unauthorized modification. Encapsulation provides security by hiding the data from the outside world.

It can be viewed as a wrapper that restricts or prevents properties and methods from outside access. With the help of access modifiers such as private, protected and public keywords, one can control the access of data members and member functions.

**Encapsulation Demonstration in Real-Time :**One of the most practical examples of encapsulation is a school bag. Our books, pencils, and other items may be kept in our school bag.The following are some of the advantages of encapsulation:

- **Data Hiding**
- **Increased Flexibility**
- It also promotes **reusability** .

### *1.6 Data Abstraction:*

Abstraction refers to hiding the background details and provides only essential details. One of the most common examples regarding this feature is television, in television we control it with help of a remote and, know its external or feature to be used most whereas we don't know the internal working of television.

**Another Real-Time Example :** Abstraction reveals just the most significant facts to the user while hiding the underlying intricacies. For example, when we ride a bike, we only know how to ride it but not how it works. We also have no idea how a bike works on the inside.

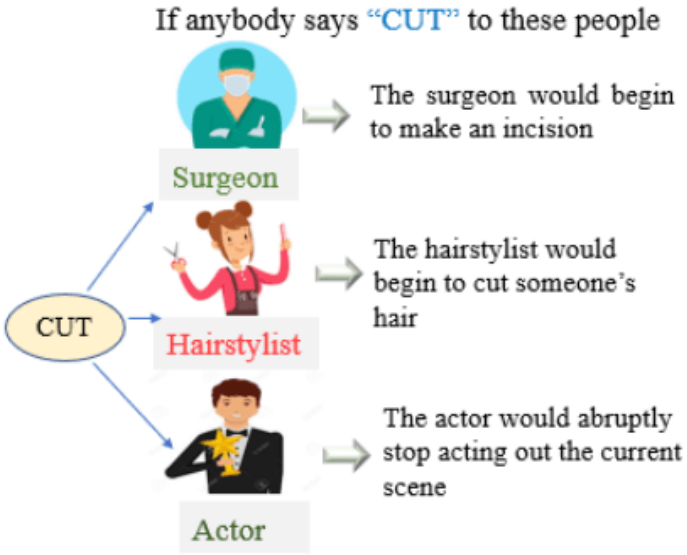
**Advantages of Abstraction**

- It simplifies the process of seeing things in their entirety.
- Code duplication is avoided, and reusability is increased.
- Because just the most necessary information is shown to the user, it helps to enhance the security of an application or software.

***1.7. Polymorphism:***

Polymorphism in OOP is the ability of an object to take many forms. In simple words, it allows us to perform the same action in many different ways. Polymorphism is taken from the Greek words Poly (many) and morphism (forms). Polymorphism defines the ability to take different forms.

For example, The student can act as a student in college, act as a player on the ground, and as a daughter/brother in the home. Another example in the programming language, the + operator, acts as a concatenation and arithmetic addition.




Python Polymorphism

Polymorphism refers to the creation of items that have similar behavior. For example, objects may override common parent behaviors with particular child behaviors through inheritance. Method overriding and method overloading are two ways that polymorphism enables the same method to perform various actions.

**1.8. Merits & demerits of OOPS Programming:**

**Advantages Of Oop:**

1. **Productivity of software development increased:** Object-arranged writing computer programs is measured, as it gives detachment of obligations in object-based program advancement. It is additionally extensible, as articles can be stretched out to incorporate new qualities and practices. Items can likewise be reused inside and across applications. Due to these three variables – particularity, extensibility, and

reusability – object-situated programming gives further developed programming advancement usefulness over conventional strategy based programming methods.

2. **Software maintenance improved:** For the reasons referenced above, object-oriented programming is likewise simpler to keep up with. Since the plan is secluded, a piece of the framework can be refreshed if there should arise an occurrence of issues without a need to roll out huge scope improvements.
3. **Quicker improvement:** Reuse empowers quicker advancement. Object-situated programming dialects accompany rich libraries of articles, and code created during projects is additionally reusable in later ventures.
4. **Cost of development lowered:** The reuse of programming likewise brings down the expense of advancement. Normally, more exertion is placed into the article situated examination and plan, which brings down the general expense of improvement.
5. **Good quality software:** Faster improvement of programming and lower cost of advancement permits additional time and assets to be utilized in the confirmation of the product. Albeit quality is reliant upon the experience of the groups, object-situated programming will in general bring about greater programming.

**Disadvantages Of Oop:**

1. **Steep expectation to learn and adapt:** The perspective engaged with object-situated programming may not be normal for certain individuals, and it can invest in some opportunity to become accustomed to it. It is complex to make programs in view of the cooperation of articles. A portion of the key programming procedures, like inheritance and polymorphism, can be tested to appreciate at first.
2. **Bigger program size:** Object-arranged programs commonly include more lines of code than procedural projects.
3. **More slow projects:** Object-arranged programs are normally slower than procedure-based programs, as they ordinarily require more guidelines to be executed.
4. **Not appropriate for a wide range of issues:** There are issues that loan themselves well to useful programming style, rationale programming style, or strategy based programming style, and applying object-arranged programming in those circumstances will not bring about effective projects.

**1.9 Comparison of Advantages And Disadvantages Of Oop:**

| **Advantages** | **Disadvantages** |
|---|---|
| We can reuse the code multiple times using class | Size is larger than other programs |
| Inherit the class to subclass for data redundancy | It required a lot of effort to create |
| It is easy to maintain and modify | It is slower than other programs |
| It maintains the security of data | It is not suitable for some sorts of problems |

| | |
|---|---|
| Low-cost development | It takes time to get used to it. |

## 1.10. Applications of Object-Oriented Programming:

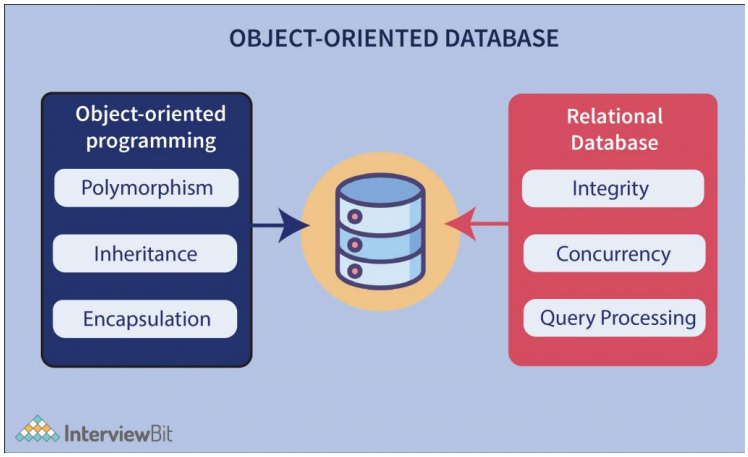
**1. Client-Server Systems:** Object-oriented client-server systems provide the IT infrastructure, creating Object-Oriented Client-Server Internet (OCSI) applications. Here, infrastructure refers to operating systems, networks, and hardware. OSCI consist of three major technologies:

1. The Client Server
2. Object-Oriented Programming
3. The Internet


**2. Object-Oriented Databases:** They are also called Object Database Management Systems (ODBMS). These databases store objects instead of data, such as real numbers and integers. Objects consist of the following:

**Attributes:** Attributes are data that define the traits of an object. This data can be as simple as integers and real numbers. It can also be a reference to a complex object.
**Methods:** They define the behavior and are also called functions or procedures.

These databases try to maintain a direct correspondence between the real-world and database objects in order to let the object retain its identity and integrity. They can then be identified and operated upon.



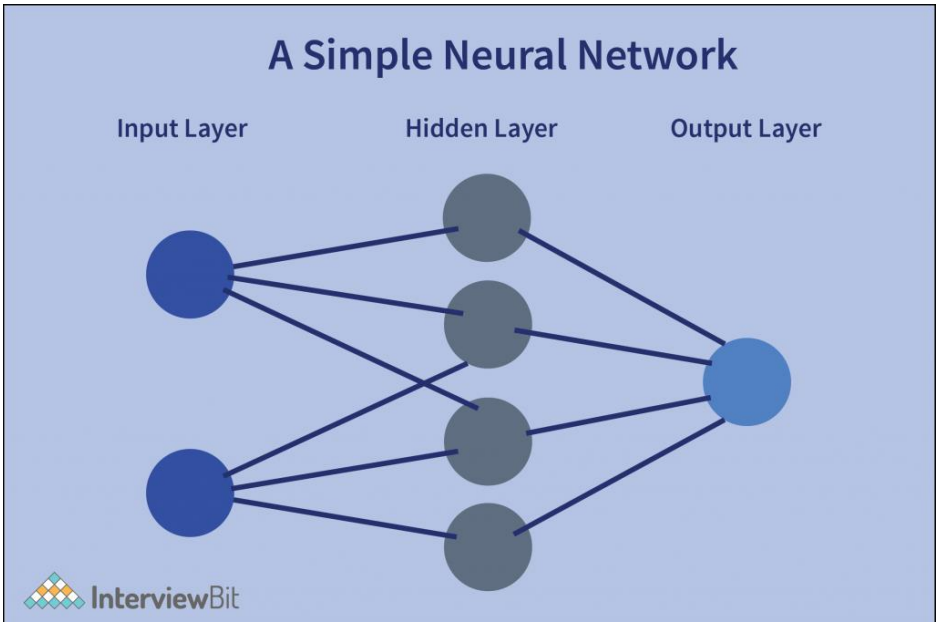

**3. Real-Time System Design:** Real-time systems inherent complexities that make it difficult to build them. Object-oriented techniques make it easier to handle those complexities. These techniques present ways of dealing with these complexities by providing an integrated framework, which includes schedulability analysis and behavioral specifications.

**4. Simulation and Modelling System:** It's difficult to model complex systems due to the varying specification of variables. These are prevalent in medicine and in other areas of natural science, such as ecology, zoology, and agronomic systems. Simulating complex systems requires modelling and understanding interactions explicitly. Object-oriented programming provides an alternative approach for simplifying these complex modelling systems.

**5. Hypertext and Hypermedia:** OOP also helps in laying out a framework for hypertext. Basically, hypertext is similar to regular text, as it can be stored, searched, and edited easily. The only difference is that hypertext is text with pointers to other text as well.

Hypermedia, on the other hand, is a superset of hypertext. Documents having hypermedia not only contain links to other pieces of text and information but also to numerous other forms of media, ranging from images to sound.

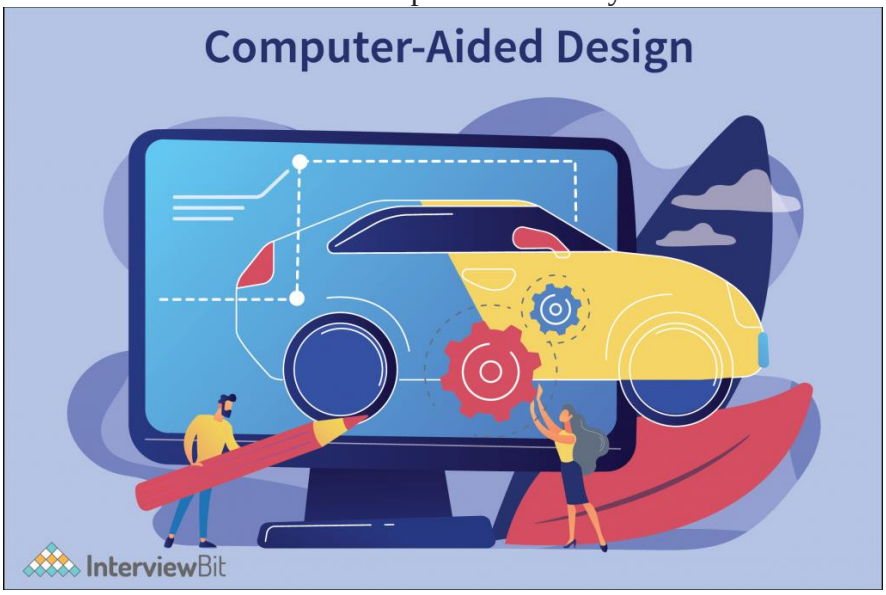
**6. Neural Networking and Parallel Programming:** It addresses the problem of prediction and approximation of complex time-varying systems. Firstly, the entire time-varying process is split into several time intervals or slots. Then, neural networks are developed in a particular time interval to disperse the load of various networks. OOP simplifies the entire process by simplifying the approximation and prediction ability of networks.




**7. Office Automation Systems:** These include formal as well as informal electronic systems primarily concerned with information sharing and communication to and from people inside and outside the organization. Some examples are:

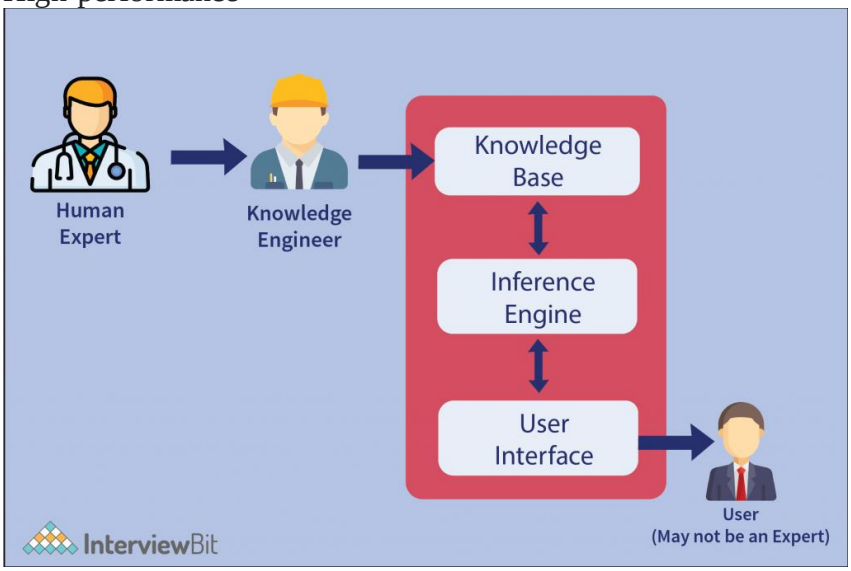
- Email
- Word processing
- Web calendars
- Desktop publishing

**8. CIM/CAD/CAM Systems:** OOP can also be used in manufacturing and design applications, as it allows people to reduce the effort involved. For instance, it can be used while designing blueprints and flowcharts. OOP makes it possible for the designers and engineers to produce these flowcharts and blueprints accurately.




**9. AI Expert Systems:** These are computer applications that are developed to solve complex problems pertaining to a specific domain, which is at a level far beyond the reach of a human brain. It has the following characteristics:

- Reliable
- Highly responsive
- Understandable
- High-performance




## 1.11. Implementation of classes and objects in Python:

A class is a collection of objects. Unlike the primitive data structures, classes are data structures that the user defines. They make the code more manageable.

### 1.11.1 Creating Class and Objects:

In Python, Use the keyword class to define a Class. In the class definition, the first string is docstring which, is a brief description of the class.
The docstring is not mandatory but recommended to use. We can get docstring using __doc__ attribute. Use the following syntax to create a class.

**Syntax**

```
class classname:
   "'documentation string'"
   class_suite
```

- **Documentation string**: represent a description of the class. It is optional.
- **class_suite**: class suite contains class attributes and methods

We can create any number of objects of a class. use the following syntax to create an object of a class.

```
reference_variable = classname()
```

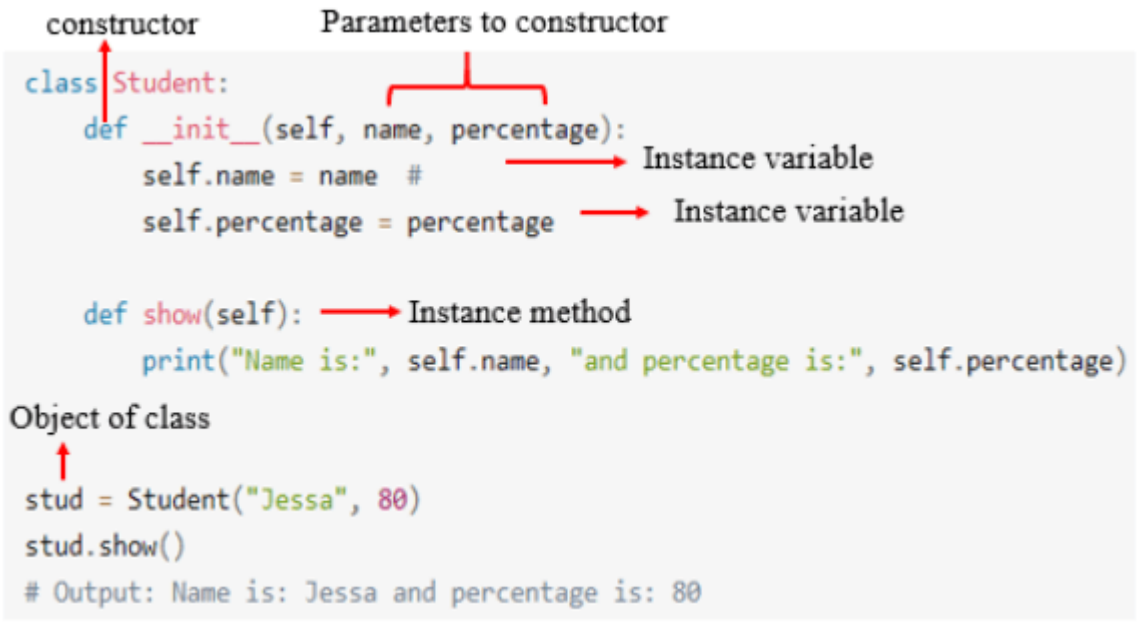
**OOP Example: Creating Class and Object in Python**

```
class Employee:
    # class variables
    company_name = Jewel Software'

    # constructor to initialize the object
    def __init__(self, name, salary):
        # instance variables
        self.name = name
        self.salary = salary

    # instance method
    def show(self):
        print('Employee:', self.name, self.salary, self.company_name)

# create first object
emp1 = Employee("Ravi Raju", 60000)
emp1.show()

# create second object
emp2 = Employee(" Joseph", 90000)
emp2.show()
```

**Output**:

```
Employee: Ravi Raju 60000 Jewel Software

Employee: Joseph 90000 Jewel Software
```

- In the above example, we created a Class with the name Employee.
- Next, we defined two attributes name and salary.
- Next, in the __init__() method, we initialized the value of attributes. This method is called as soon as the object is created. The init method initializes the object.
- Finally, from the Employee class, we created two objects, Ravi Raju and Joseph.
- Using the object, we can access and modify its attributes.




**instance variables and methods**

### *Constructors in Python*

In Python, a **constructor** is a special type of method used to initialize the object of a Class. The constructor will be executed automatically when the object is created. If we create three objects, the constructor is called three times and initialize each object.

The main purpose of the constructor is to declare and initialize instance variables. It can take at least one argument that is self. The __init()__ method is called the constructor in Python. In other words, the name of the constructor should be __init__(self).

A constructor is optional, and if we do not provide any constructor, then Python provides the default constructor. Every class in Python has a constructor, but it's not required to define it.

### 1.11.2 Class Attributes and Methods

When we design a class, we use instance variables and class variables.

In Class, attributes can be defined into two parts:

- **Instance variables**: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor ( the __init__() method of a class).
- **Class Variables**: A class variable is a variable that is declared inside of class, but outside of any instance method or __init()__ method.

Inside a Class, we can define the following three types of methods.

- **Instance method**: Used to access or modify the object attributes. If we use instance variables inside a method, such methods are called instance methods.

- **Class method**: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.
- **Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't have access to the class attributes.

### 1.11.3 Self Argument:

Whenever we create a class in Python, the programmer needs a way to access its *attributes* and *methods*. In most languages, there is a fixed syntax assigned to refer to attributes and methods; for example, C++ uses this for reference.

In Python, the word self is the first parameter of methods that represents the instance of the class. Therefore, in order to call attributes and methods of a class, the programmer needs to use self.

It is not mandatory to name the first parameter as a self. We can give any name whatever we like, but it has to be the first parameter of an instance method.

## Example

```
class Student:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # self points to the current object
    def show(self):
        # access instance variable using self
        print(self.name, self.age)

# creating first object
emma = Student('Ishvika', 02)
emma.show()

# creating Second object
kelly = Student('Ammulu', 15)
kelly.show()
```

## Output

```
Ishvika 02

Ammulu 15
```

### Example : 2

```
class food():

# init method or constructor

def __init__(self, fruit, color):
self.fruit = fruit
self.color = color

def show(self):
print("fruit is", self.fruit)
print("color is", self.color )

apple = food("apple", "red")
grapes = food("grapes", "green")

apple.show()
grapes.show()
```

**Output:**

```
Fruit is apple
color is red
Fruit is grapes
color is green
```

**Python Class self Constructor**

self is also used to refer to a variable field within the class. Let's take an example and see how it works:

```
class Person:


# name made in constructor
def __init__(self, John):
self.name = John

def get_person_name(self):
return self.name
```

In the above example, self refers to the name variable of the entire Person class. Here, if we have a variable within a method, self will not work. That variable is simply existent only while that method is running and hence, is local to that method. For defining global fields or the variables of the complete class, we need to define them outside the class methods.

### 1.11.4 The __init__Method():

In object-oriented programming, **A constructor is a special method used to create and initialize an object of a** class. This method is defined in the class.

- The constructor is executed automatically at the time of object creation.
- The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.

For example, when we execute obj = Sample(), Python gets to know that obj is an object of class Sample and calls the constructor of that class to create an object.

**Note**: In Python, internally, the __new__ is the method that creates the object, and __del__ method is called to destroy the object when the reference count for that object becomes zero.

In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**

1. Internally, the __new__ is the method that creates the object
2. And, using the __init__() method we can implement constructor to initialize the object.

**Syntax** of a constructor:

```
def __init__(self):
    # body of the constructor
```

Where,

- def: The keyword is used to define function.
- __init__() Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated.
- self: The first argument self refers to the current object. It binds the instance to the __init__() method. It's usually named self to follow the naming convention.

**Note**: The __init__() method arguments are optional. We can define a constructor with any number of arguments.

**Example: Create a Constructor in Python**

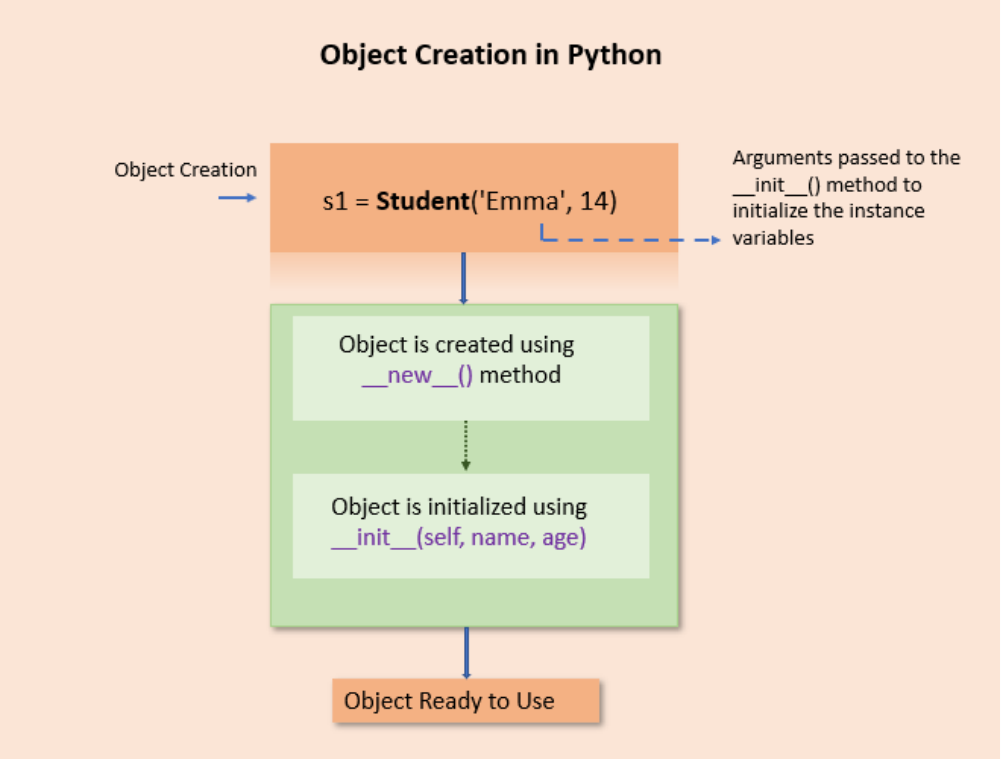
In this example, we'll create a Class **Student** with an instance variable student name. we'll see how to use a constructor to initialize the student name at the time of object creation.

```
class Student:

    # constructor
    # initialize instance variable
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')

    # instance Method
    def show(self):
        print('Hello, my name is', self.name)


# create object using constructor
s1 = Student('Ravi')
s1.show()
```

**Output:**

```
Inside Constructor

All variables initialized



Hello, my name is Ravi
```

- In the above example, an object s1 is created using the constructor
- While creating a Student object name is passed as an argument to the __init__() method to initialize the object.
- Similarly, various objects of the Student class can be created by passing different names as arguments.




**Create an object in Python using a constructor**

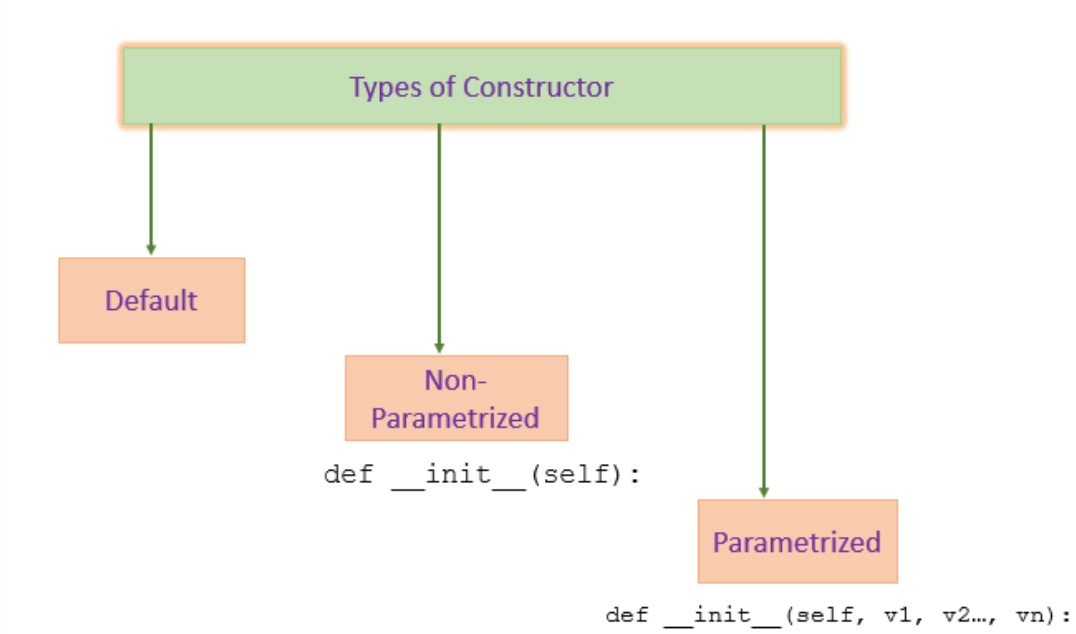
**Note**:

- For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.
- In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional.
- Python will provide a default constructor if no constructor is defined.

### Types of Constructors:

In Python, we have the following three types of constructors.

- Default Constructor
- Non-parametrized constructor
- Parameterized constructor




### 1.11.4.1 Default Constructor:

Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body.

If you do not implement any constructor in your class or forget to declare it, the Python inserts a default constructor into your code on your behalf. This constructor is known as the default constructor.

It does not perform any task but initializes the objects. It is an empty constructor without a body.

**Note**:

- The default constructor is not present in the source py file. It is inserted into the code during compilation if not exists. See the below image.
- If you implement your constructor, then the default constructor will not be added.

**Example**:

```
class Employee:

    def display(self):
        print('Inside Display')

emp = Employee()
emp.display()
```

**Output:**

```
Inside Display
```

As you can see in the example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation.

### 1.11.4.2 Non-Parametrized Constructor:

A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.

This constructor doesn't accept the arguments during object creation. Instead, it initializes every object with the same set of values.

```
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "Jewel Software"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

**Output**

```
Name: Jewel Software Address: ABC Street
```

As you can see in the example, we do not send any argument to a constructor while creating an object.

### 1.11. 4.3. Parameterized Constructor

A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.

The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments.

For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor.

**Example**:

```
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    # display object
    def show(self):
        print(self.name, self.age, self.salary)

# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()

kelly = Employee('Kelly', 25, 8500)
kelly.show()
```

**Output**

```
Emma 23 7500

Kelly 25 8500
```

In the above example, we define a parameterized constructor which takes three parameters.

### 1.12. The_del_Method()/ Destructors:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically. The **del**() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

**Syntax of destructor declaration**

```
def __del__(self):

  # body of destructor
```

**Example :**

```
 Python program to illustrate destructor

class Employee:

    # Initializing

    def __init__(self):

        print('Employee created.')



    # Deleting (Calling destructor)

    def __del__(self):

        print('Destructor called, Employee deleted.')



obj = Employee()

del obj
```

**output :**

```
Employee created.

Destructor called, Employee deleted.
```

## 1.13. INTRODUCTION:

**Python** is a high-level, interpreted scripting language developed in the late 1980s by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands

**Python** is a general purpose, dynamic, high level, and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by Guido Van Rossum at CWI in Netherland.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
- ABC language.
- Modula-3

A Few list of python versions with its released date is given below.

| Python Version | Released Date |
|---|---|
| Python 1.0 | January 1994 |
| Python 1.5 | December 31, 1997 |
| Python 1.6 | September 5, 2000 |
| Python 2.0 | October 16, 2000 |
| Python 2.5 | September 19, 2006 |
| Python 2.6 | October 1, 2008 |
| Python 2.7 | July 3, 2010 |
| Python 3.0 | December 3, 2008 |
| Python 3.3 | September 29, 2012 |
| Python 3.4 | March 16, 2014 |
| Python 3.5 | September 13, 2015 |
| Python 3.6 | December 23, 2016 |

| | |
|---|---|
| Python 3.7 | June 27, 2018 |
| Python 3.8 | October 14, 2019 |

### 1.14. FEATURES :

Python provides lots of features that are listed below.


1) Easy to Learn and Use: Python is easy to learn and use. It is developer-friendly and high level programming language.
2) Expressive Language: Python language is more expressive means that it is more understandable and readable.
3) Free and Open Source: Python language is freely available at offical web address.The source-code is also available. Therefore it is open source.
4) High level: Being a high-level language, Python code is quite like English. Looking at it, you can tell what the code is supposed to do. Also, since it is dynamically-typed, it mandates indentation. This aids readability.
5) Portable: Python language is also a portable language. for example, if we have python code for windows and if we want to run this code on other platform such as Linux, Unix and Mac then we do not need to change it, we can run this code on any platform.
6) Object-Oriented Language: Python supports object oriented language and concepts of classes and objects come into existence.
7) Extensible: It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.
8) Embeddable or Integrate : It can be easily integrated with languages like C, C++, JAVA etc.
9) Interpreted Language: Python is an Interpreted Language. because python code is executed line by line at a time. like other language c, c++, java etc there is no need to compile python code this makes it easier to debug our code. The source code of python is converted into an immediate form called byte code.

10) Large Standard Library: Python has a large and broad library and provides rich set of module and functions for rapid application development.
11) GUI Programming Support: Graphical user interfaces can be developed using Python
12) Dynamically Typed: Python is dynamically-typed language. That means the type (for example- int, double, long etc) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

## 1.15. PYTHON APPLICATIONS:

Python is known for its general purpose nature that makes it applicable in almost each domain of software development. Python as a whole can be used in any sphere of development. Here, we are specifying applications areas where python can be applied.

1) Web Applications:

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser etc. It also provides Frameworks such as Django, Pyramid, Flask etc to design and delelop web based applications. Some important developments are: PythonWikiEngines, Pocoo, PythonBlogSoftware etc.

2) Desktop GUI Applications:

Python provides Tk GUI library to develop user interface in python based application. Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms. The Kivy is popular for writing multitouch applications.

3) Software Development:

Python is helpful for software development process. It works as a support language and can be used for build control and management, testing etc.

4) Scientific and Numeric:

Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

5) Business Applications:

Python is used to build Bussiness applications like ERP and e-commerce systems. Tryton is a high level application platform.

6) Console Based Application:

We can use Python to develop console based applications. For example: IPython.

7) Audio or Video based Applications:

Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are: TimPlayer, cplay etc.

8) 3D CAD Applications:

To create CAD application Fandango is a real application which provides full features of CAD.

9) Enterprise Applications:

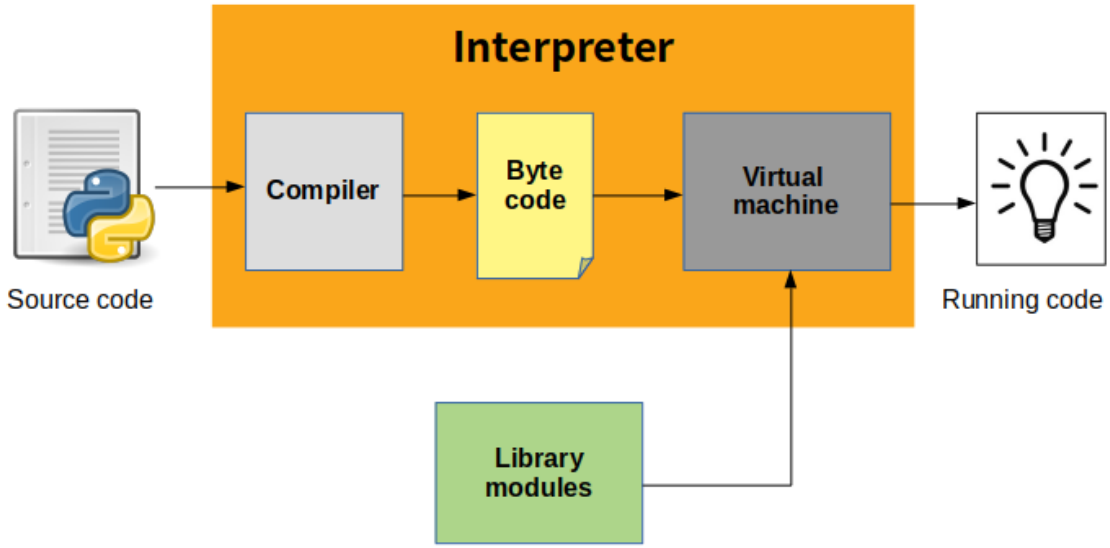
Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

10) Applications for Images:

Using Python several application can be developed for image. Applications developed are: VPython, Gogh, imgSeek etc.

## 1.16 CREATING & EXECUTING PYTHON :

There are four steps that python takes when you hit return: lexing, parsing, compiling, and interpreting. Lexing is breaking the line of code you just typed into tokens. The parser takes those tokens and generates a structure that shows their relationship to each other (in this case, an Abstract Syntax Tree). The compiler then takes the AST and turns it into one (or more) code objects. Finally, the interpreter takes each code object executes the code it represents.




The Python interpreter performs following tasks to execute a Python program :

- Compiler compiles your **source code (the statements in your file)** into a format known as **byte code**. Compilation is simply a **translation step**!
- Byte code is a lower level, platform independent, efficient and intermediate representation of source code. Each of your source statements is translated into a group of byte code instructions. Compiled code is usually stored in.pyc files , and is regenerated when the source is updated.
- **The Python Virtual Machine(PVM) is the runtime engine of Python**; it's always present as part of the Python system, and is the component that truly runs your scripts. The bytecode (.pyc file) is loaded into the Python runtime and interpreted by a Python Virtual Machine, which is a piece of code that reads each instruction in the bytecode and executes.

## 1.17 DATA TYPES :

Variables can hold values of different data types. Python is a dynamically typed language hence we need not define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

```
A=10
b="Hi Python"
c = 10.5
print(type(a));
print(type(b));
print(type(c));
```

Output:
<type 'int'>
<type 'str'>
<type 'float'>

### *Standard data types*

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. Numbers : Int, float & Complex
2. Sequences
    1. String
    2. List
    3. Tuple
    4. sets
3. Dictionary
4. Boolean

## 1.18. Numbers:

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. For example;

```
a = 3 , b = 5  #a and b are number objects
```

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
4. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)

Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.

A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts respectively).

### 1.19. Sequences:

In Python, **sequence** is the generic term for an ordered set. There are several types of sequences in Python, the following three are the most important.

Strings are a special type of sequence that can only store characters, and they have a special notation. However, all of the sequence operations described below can also be used on strings.

Lists are the most versatile sequence type. The elements of a list can be any object, and lists are **mutable** - they can be changed. Elements can be reassigned or removed, and new elements can be inserted.

Tuples are like lists, but they are **immutable** - they can't be changed.

### 1.19.1 String:

The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator * is known as repetition operator as the operation "Python " *2 returns "Python Python ".

The following example illustrates the string handling in python.

```
str1 = 'hello Ravi' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

**Output:**

```
he
o
hello Ravihello Ravi
hello Ravi how are you
```

### 1.19.2 List:

Lists are similar to arrays in C. However; the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
l1  = [1, "hi", "python", 2]
print (l1[3:]);
print (l1[0:2]);
print (l1);
print (l1+ l1);
print (l1 * 3);
```

**Output:**

```
[2]
[1, 'hi']
[1, 'hi', 'python', 2]
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
```

### 1.19.3 Tuple:

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```
t  = ("hi", "python", 2)
print (t[1:]);
print (t[0:1]);
print (t);
print (t + t);
```

```
print (t * 3);
print (type(t))
t[2] = "hi";  # tuple values are immutable
```

**Output:**

```
('python', 2)
('hi',)
('hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)
<type 'tuple'>
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

### 1.19.4 Sets:

A set is a collection of data types in Python, same as the list and tuple. However, it is not an ordered collection of objects. The set is a Python implementation of the set in Mathematics. A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.

A set object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in curly brackets {}.

Syntax:
Set={value1,value2,value3,....valueN}
The following defines a set object.

```
>>> S1={1, "Bill", 75.50}
```

A set doesn't store duplicate objects. Even if an object is added more than once inside the curly brackets, only one copy is held in the set object. Hence, indexing and slicing operations cannot be done on a set object.

```
>>> S1={1, 2, 2, 3, 4, 4, 5, 5}
>>> S1
{1, 2, 3, 4, 5}
```

## 1.20. Dictionary:

Dictionary is an ordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma and enclosed in the curly braces {}.

Consider the following example.

```
d = {1:'Ravi', 2:'Raju', 3:'Sudheer',4:'Sekhar'};
print("1st name is "+d[1]);
print("2nd name is "+ d[4]);
print (d);
print (d.keys());
print (d.values());
```

**Output:**

```
1st name is Ravi
2nd name is Sekhar
{1: 'Ravi', 2: 'Raju', 3: 'Sudheer', 4: 'Sekhar'}
[1, 2, 3, 4]
['Ravi', 'Raju', 'Sudheer', 'Sekhar']
```

## 1.21 .Boolean

True and False are the two default values for the Boolean type. These qualities are utilized to decide the given assertion valid or misleading. The class book indicates this. False can be represented by the 0 or the letter "F," while true can be represented by any value that is not zero.

Look at the following example.

```
# Python program to check the boolean type
print(type(True))
print(type(False))
print(false)
```

**Output:**

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

### 1.22 Operators:

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

1. Arithmetic operators
2. Comparison or Relational operators
3. Logical operators
4. Bitwise operators
5. Assignment operators

6. Special operators
    a. Identity operators
    b. Membership 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 |
| // | Division (floor): divides the first operand by the second | x // y |
| ** | Exponential: multiply first operand by second operand times | x**y |
| % | Modulus: returns the remainder when first operand is divided by the second | x % y |

```
# Examples of Arithmetic Operator
a = 9
b = 4
add = a + b       # Addition of numbers
sub = a – b       # Subtraction of numbers
mul = a * b       # Multiplication of number
div1 = a / b      # Division(float) of number
div2 = a // b     # Division(floor) of number
exp = a ** b      # Exponentaial of number
mod = a % b       # Modulo of both number

# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(exp)
print(mod)
```

**Output:**

```
13
5
36
2.25
6561
2
```

**2.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 |

**# Examples of Relational Operators**

```
a = 13
b = 33
print(a > b)        # a > b is False
print(a < b)        # a < b is True
print(a == b)       # a == b is False
print(a != b)       # a != b is True
print(a >= b)       # a >= b is False
print(a <= b)       # a <= b is True
```

**Output:**

**False**
**True**
**False**
**True**
**False**
True

**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 |

**# Examples of Logical Operator**
**a = True**
**b = False**
**print(a and b)** **# Print a and b is False**
**print(a or b)** **# Print a or b is True**
**print(not a)** **# Print not a is False**

**Output:**
**False**
**True**
**False**

**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<< |

**# Examples of Bitwise operators**
**a = 10**
**b = 4**

**print(a & b)** **# Print bitwise AND operation**
**print(a | b)** **# Print bitwise OR operation**
**print(~a)** **# Print bitwise NOT operation**
**print(a ^ b)** **# print bitwise XOR operation**
**print(a >> 2)** **# print bitwise right shift operation**
**print(a << 2)** **# print bitwise left shift operation**

**Output:**
**0**

**14**
**-11**
**14**
**2**
**40**

**5.Assignment operators:** Assignment operators are used in Python to assign values to variables.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.
There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Examples of Assignment operators in Python

| Operator | Example | Equivatent to |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| *= | x *= 5 | x = x * 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| **= | x **= 5 | x = x ** 5 |
| &= | x &= 5 | x = x & 5 |
| \|= | x \|= 5 | x = x \| 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

**6. Special Operators:**
There are some special type of operators like-

1. **Identity operators-**
   **is** and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

**is** True if the operands are identical
**is not** True if the operands are not identical

```
# Examples of Identity operators
a1 = 3
b1 = 3
a2 = 'AU -Information Technology'
b2 = ' AU -Information Technology'
a3 = [1,2,3]
b3 = [1,2,3]

print(a1 is not b1)
print(a2 is b2)
print(a3 is b3)       # Output is False, since lists are mutable.
```

**Output:**

**False**
**True**
**False**

**2 Membership operators-**

**in** and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

**in** True if value is found in the sequence
**not in** True if value is not found in the sequence

```
# Examples of Membership operator

x = ' AU -Information Technology'
y = {3:'a',4:'b'}
print('U' in x)
print('it-a' not in x)
print('AU' not in x)
print(3 in y)
print('b' in y)
```

**Output:**

**True**
**True**
**False**
**True**
**False**

### 1.23. Expressions:

Expressions are representations of value. They are different from statement in the fact that statements do something while expressions are representation of value. For example any string is also an expressions since it represents the value of the string as well.

Python has some advanced constructs through which you can represent values and hence these constructs are also called expressions.

Python expressions only contain identifiers, literals, and operators. So, what are these?

**Identifiers**: Any name that is used to define a class, function, variable module, or object is an identifier.

**Literals**: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

**Operators**: In Python you can implement the following operations using the corresponding tokens.

| Operator | Token |
|---|---|
| Add | + |
| Subtract | - |
| Multiply | * |
| Power | ** |
| Integer Division | / |
| Remainder | % |
| Decorator | @ |

| Operator | Token |
|---|---|
| Binary left shift | << |
| Binary right shift | >> |
| And | & |
| Or | \ |
| Binary Xor | ^ |
| Binary ones complement | ~ |
| Less than | < |
| Greater than | > |
| Less than or equal to | <= |
| Greater than or equal to | >= |
| Check equality | == |
| Check not equal | != |

Following are a few types of python expressions:

**List comprehension**

The syntax for list comprehension is shown below:

```
[ compute(var) for var in iterable ]
```

For example, the following code will get all the number within 10 and put them in a list.

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Dictionary comprehension**

This is the same as list comprehension but will use curly braces:

```
{ k, v for k in iterable }
```

For example, the following code will get all the numbers within 5 as the keys and will keep the corresponding squares of those numbers as the values.

```
>>> {x:x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

**Generator expression**

The syntax for generator expression is shown below:

```
( compute(var) for var in iterable )
```

For example, the following code will initialize a generator object that returns the values within 10 when the object is called.

```
>>> (x for x in range(10))
<generator object <genexpr> at 0x7fec47aee870>
>>> list(x for x in range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Conditional Expressions**

You can use the following construct for one-liner conditions:

```
true_value if Condition else false_value
```

Example:

```
>>> x = "1" if True else "2"
>>> x
'1'
```

### 1.24 Operator Precedence & Associativity:

The operators with the highest precedence at the top and lowest at the bottom.

| Operators | Usage |
|---|---|
| { } | Parentheses (grouping) |
| f(args…) | Function call |
| x[index:index] | Slicing |
| x[index] | Subscription |
| x.attribute | Attribute reference |
| ** | Exponent |
| ~x | Bitwise not |
| +x, -x | Positive, negative |
| *, /, % | Product, division, remainder |
| +, – | Addition, subtraction |
| <<, >> | Shifts left/right |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| \| | Bitwise OR |
| in, not in, is, is not, <, <=, >, >=, <>, !=, == | Comparisons, membership, identity |
| not x | Boolean NOT |
| and | Boolean AND |
| or | Boolean OR |
| lambda | Lambda expression |

The associativity is the order in which Python evaluates an expression containing multiple operators of the same precedence. Almost all operators except the exponent (**) support the left-to-right associativity.

### 1.25 Loop / Iterative / Repetative Statements:

The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times.

Before going to discuss about loops, we need to learn range() function concept in Python.

#### 1.25.1 RANGE() FUNCTION:

Range function is used to iterate the elements in sequence manner
range function is used in for loops. Range function makes programmer to index free while iterating the loops

**RANGE() FUNCTION ARGUMENTS:**

Range() function has two set of arguments. They are,

1. SINGLE ARGUMENT RANGE FUNCTION:
SYNTAX:
range (stop)
STOP:
To generate the numbers in sequence up to “stop-1” numbers. It will start from zero.
Ex: range(5)=[0,1,2,3,4]

2. TWO ARGUMENT RANGE FUNCTION:
SYNTAX:
**range (start, stop)**
START:
Generating the numbers starting from “start”
STOP:
To generate the numbers in sequence up to “stop-1” numbers .
Ex: range(0,5)=[0,1,2,3,4]

3. THREE ARGUMENT RANGE FUNCTION
SYNTAX:
**range([start], stop[, step])**
START:
Generating the numbers starting from “start”
STOP:
To generate a numbers in sequence up to “stop-1” numbers .
STEP:
difference between each number in the sequence.
Ex: range(1,5,1)=[1,2,3,4]

### 1.25.2 For:

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

```
for iterating_var in sequence:
   statement(s)
```

It can be used to iterate over iterators and a range. We can use for in loop for user defined iterators. See this for example.

```
print("List Iteration")                # Iterating over a list
l = [" Anurag", Information ", "Technology"]
for i in l:
        print(i)

print("\nTuple Iteration")            # Iterating over a tuple (immutable)
t = (" RaviRaju", "Sekhar", "Prudhvi")
for i in t:
```

```
        print(i)

print("\nString Iteration")          # Iterating over a String
s = "KING"
for i in s :
        print(i)

print("\nDictionary Iteration")      ## Iterating over dictionary
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
        print("%s %d" %(i, d[i]))
```

**Output:**

```
List Iteration
Anurag
Information
Technology

Tuple Iteration
RaviRaju
Sekhar
Prudhvi


String Iteration
K
I
N
G


Dictionary Iteration
xyz  123
abc  345
```

**Example:2**

```
i=1
n=int(input("Enter the number up to which you want to print the natural numbers?"))
for i in range(0,10):
   print(i,end = ' ')
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

**Example:3**

```
for num in range(1,10,2):
 print num
```

**output:**

```
1
3
5
7
9
```

**Nested for loop in python:**

Python allows us to nest any number of for loops inside a for loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax of the nested for loop in python is given below.

```
for iterating_var1 in sequence:
    for iterating_var2 in sequence:
        #block of statements
    #Other statements
```

Example:

```
n = int(input("Enter the number of rows you want to print?"))
i,j=0,0
for i in range(0,n):
    print()
    for j in range(0,i+1):
        print("*",end="")
```

Output:

```
Enter the number of rows you want to print?5
*
**
***
****
*****
```

### 1.25.3 While Loop:

Syntax :

```
while expression:
    statement(s)
```

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.

```
# prints Hello Anurag  3 Times
count = 0
while (count < 3):
```

```
count = count+1
print("Hello Anurag")
```

Output:

Hello Anurag
Hello Anurag
Hello Anurag

### 1.26 Decision Making in Python (if , if..else, Nested if, if-elif):

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

- if statement
- if..else statements
- nested if statements
- if-elif ladder

#### 1.26.1 if statement:

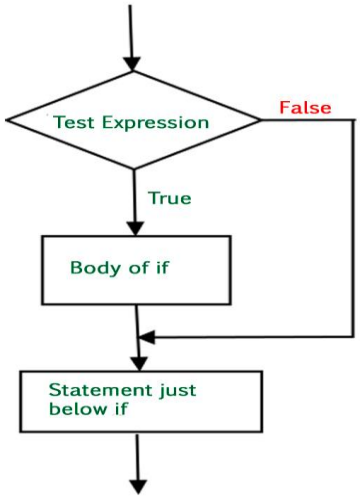
if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not. Python uses indentation to identify a block.

**Syntax**:

```
if condition:
        statement1
statement2
```

# Here if the condition is true, if block will consider only statement1 to be inside its block.

**Flowchart:**



```
# python program to illustrate If statement

i = 10
if (i > 15):
print ("10 is less than 15")
print ("I am Not in if")
```

**output:**

```
I am Not in if
```

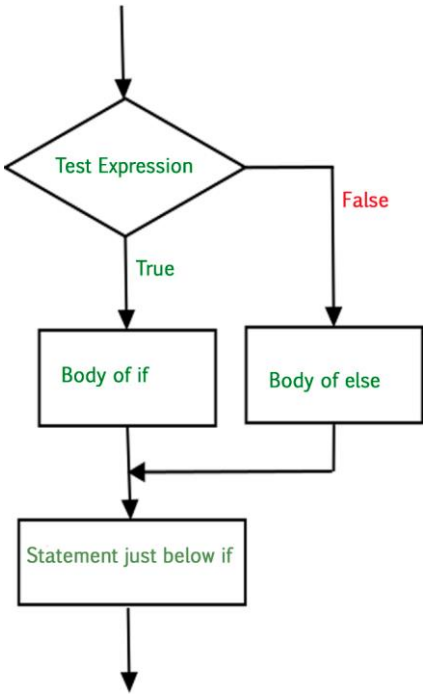
### 1.26.2 if-else statement:

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

**Syntax**:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

**Flow chart:**




**Example:**

```
# python program to illustrate If else statement
i = 20;
if (i < 15):
        print ("i is smaller than 15")
        print ("i'm in if Block")
else:
        print ("i is greater than 15")
        print ("i'm in else Block")
```

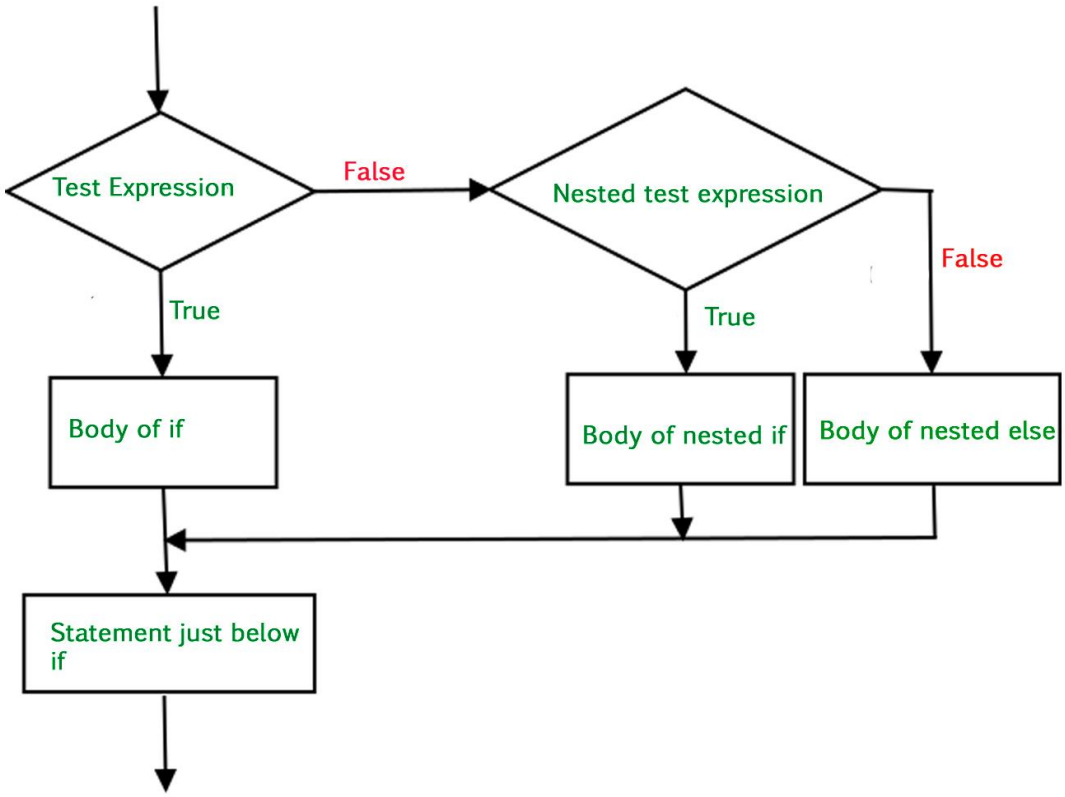
**print ("i'm not in if and not in else Block")**

**Output:**
i is greater than 15
i'm in else Block
i'm not in if and not in else Block

### 1.26.3 Nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

**Syntax**:
```
if (condition1):
  # Executes when condition1 is true
  if (condition2):
    # Executes when condition2 is true
  # if Block is end here
# if Block is end here
```
**Flow chart:-**




**Example:**
**# python program to illustrate nested If statement**

```
i = 10
if (i == 10):
        # First if statement
        if (i < 15):
```

```
        print ("i is smaller than 15")
# Nested - if statement
# Will only be executed if statement above
# it is true
if (i < 12):
        print ("i is smaller than 12 too")
else:
        print ("i is greater than 15")
```

**output:**

i is smaller than 15
i is smaller than 12 too

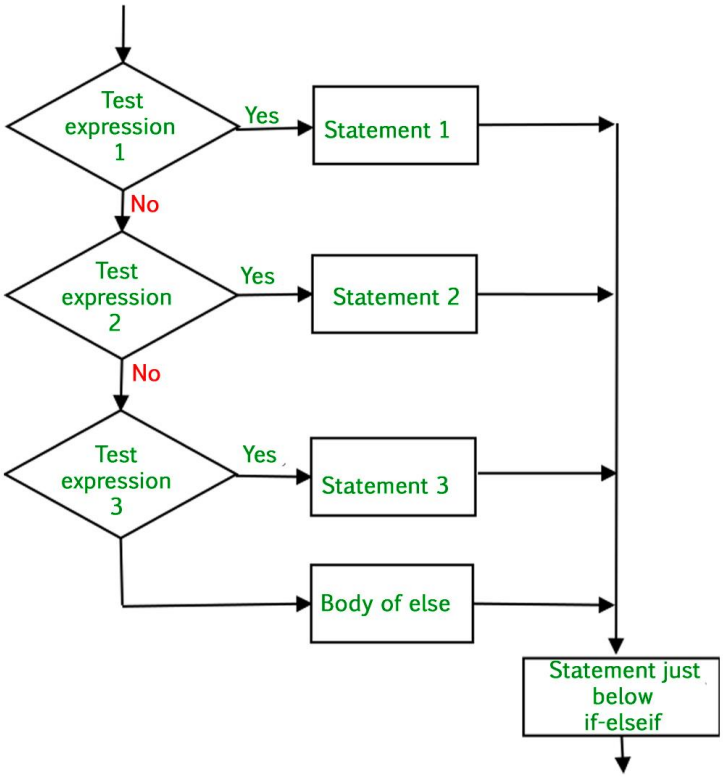
### 1.26.4 if-elif-else ladder:

A user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

**Syntax**:-

```
if (condition):
   statement
elif (condition):
   statement
.
.
else:
   statement
```

**Flow Chart:-**




**Example:**
**# Python program to illustrate if-elif-else ladder**

```
i = 20
if (i == 10):
        print ("i is 10")
elif (i == 15):
        print ("i is 15")
elif (i == 20):
        print ("i is 20")
else:
        print ("i is not present")
```

**output:**
i is 20

**# Python program to find the largest number among the three input numbers**

```
# take three numbers from user
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))
        num3 = float(input("Enter third number: "))

        if (num1 > num2) and (num1 > num3):
           largest = num1
        elif (num2 > num1) and (num2 > num3):
           largest = num2
        else:
```

```
        largest = num3

    print("The largest number is",largest)
```

**output:**

```
Enter first number: 10
Enter second number: 12
Enter third number: 14
The largest number is 14.0
```

## A program to calculate the electricity bill, we must understand electricity charges and rates.

/* 1 - 100 unit - 1.5/-, 101-200 unit - 2.5/-, 201-300 unit - 4/-, 300 - 350 unit - 5/-, above 300 - fixed charge 1500/- for example */

This program can explain as follows- Declare total **unit** consumed by the customer using the variable **unit.** If the **unit** consumed less or equal to 100 units, calculates the total amount of **consumed =units*1.5.** If **the unit** consumed between 100 to 200 units, calculates the total amount of **consumed=(100*1.5)+(unit-100)*2.5).** If **unit** consumed between 200 to 300 units ,calculates total amount of**consumed=(100*1.5)+(200-100)*2.5+(units-200)*4.** If **unit** consumed between 300-350 units ,calculates total amount of **consumed=(100*1.5)+(200-100)*2.5+(300-200)*4+(units-350)*5.** If **the unit** consumed above 350 units, **fixed charge – 1500/=**

**additional charges**

if units<=100 – 25.00, if 100< units and units<=200 – 50.00, if 200 < units and units<=300 – 75.00, if 300<units and units<=350 – 100.00, if units above 350 – No additional charges

**#program for calculating electricity bill in Python using and operator**

```
units=int(input("Number of unit consumed: "))
if(units>0 and units<=100):
        payAmount=units*1.5
        fixedcharge=25.00
elif(units>100 and units<=200):
        payAmount=(100*1.5)+(units-100)*2.5
        fixedcharge=50.00
elif(units>200 and units<=300):
        payAmount=(100*1.5)+(200-100)*2.5+(units-200)*4
        fixedcharge=50.00
elif(units>300):
        payAmount=2500;#fixed rate
        fixedcharge=75.00
else:
        payAmount=0;

Total= payAmount+fixedcharge
print("\nElecticity bill pay=%.2f: " %Total);
```

**output:**

Number of unit consumed: 234
Electricity bill pay=586.00:

## 1.27 Standard I/O operations:

### 1.27.1 Input():

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.

**input ( prompt )**
**raw_input ( prompt )**

**input ( ) :** This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –

```
# Python program showing    # a use of input()

val = input("Enter your value: ")
print(val)
```

**output:**
**Enter your value: 89**
**89**

**How the input function works in Python :**

- When input() function executes program flow will be stopped until the user has given an input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, input function convert it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using typecasting.

**Code:**

```
# Program to check input
# type in Python

num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)

# Printing type of input value
print ("Type of number", type(num))
print ("Type of name", type(name1))
```

**output:**
**Enter number: 1234**
**1234**
**Enter name: Raja**
**Raja**
**Type of number <class 'str'>**
**Type of name <class 'str'>**

**raw_input ( ) :** This function works in older version (like Python 2.x). This function takes exactly what is typed from the keyboard, convert it to string and then return it to the variable in which we want to store. For example –

```
# Python program showing
# a use of raw_input()

ch = raw_input("Enter your name : ")
print ch
```

**Output:**
**Enter your name: Ravi Raju Bandlamudi**
**Ravi Raju Bandlamudi**
Here, ***ch*** is a variable which will get the string value, typed by user during the execution of program. Typing of data for the `raw_input()` function is terminated by enter key. We can use `raw_input()` to enter numeric data also.

### 1.27.2 Output():

In order to print the output, python provides us with a built-in function called print().

**Syntax:**

**print (expression/constant/variable)**

Print evaluates the expression before printing it on the monitor. Print statement outputs an entire (complete) line and then goes to next line for subsequent output (s). To print more than one item on a single line, comma (,) may be used.

**Example:**

```
>>> print ("Hello")
Hello
>>> print (5.5)
5.5
>>> print (4+6)
10
```

### 1.27.3 Python | format() function:

**str.format()** is one of the *string formatting methods* in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

**Using a Single Formatter :**

Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces **{ }** into a string and calling the str.format(). The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.

***Syntax :*** *{ } .format(value)*
***Parameters :***
***(value) :*** *Can be an integer, floating point numeric constant, string, characters or even variables.*
***Returntype :*** *Returns a formatted string with the value passed as parameter in the placeholder position.*

```
# Python3 program to demonstarte   # the str.format() method

# using format option in a simple string
print ("{}, Best department in Anurag University." .format(" Information Technology"))

# using format option for a
# value stored in a variable
str = "This article is written in {}"
print (str.format("Python"))

# formatting a string using a numeric constant
print ("Hello, I am {} years old !".format(35))
```

**Output:**

```
Information Technology, Best department in Anurag University.
This article is written in Python
Hello, I am  35 years old!
```

**Using Multiple Formatters :**

Multiple pairs of curly braces can be used while formatting the string. Let's say if another variable substitution is needed in sentence, can be done by adding a second pair of curly braces and passing a second value into the method. Python will replace the placeholders by values in **order.**

***Syntax :***
*{ } { } .format(value1, value2)*
***Parameters :***
***(value1, value2) :*** *Can be integers, floating point numeric constants, strings, characters and even variables. Only difference is, the number of values passed as parameters in format() method must be equal to the number of placeholders created in the string.*

```
# Python program using multiple place  holders to demonstrate str.format() method
```

```
# Multiple placeholders in format() function

my_string = "{}, is a best {} in {}"
print (my_string.format("Information Technology ", "branch", "Anurag"))

# different datatypes can be used in formatting
print ("Hi ! My name is {} and I am {} years old" .format("RaviRaju", 35))

# The values passed as parameters
# are replaced in order of their entry
print ("This is {} {} {} {}".format("one", "two", "three", "four"))
```

**Output:**

```
Information Technology, is a best branch in Anurag
Hi! My name is RaviRaju and I am 35 years old
This is one two three four
```

## 2.1. Functions :

In Python, function is a group of related statements that perform 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.It avoids repetition and makes code reusable.

Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code.

There are three types of functions in Python:

1. Built-in functions, such as help() to ask for help, min() to get the minimum value, print()to print an object to the terminal, len(x) or type(y) or even random.choice([1, 2, 3]), these functions are called predefined functions.

2. User-Defined Functions (UDFs), which are functions that users create to help them out; And
3. Anonymous functions, which are also called lambda functions

**User defined Functions declaration:**

User-defined functions can be defined by using following format.

**Defining Functions**

A function is defined in Python by the following format:

```
def  functionname(arg1, arg2, ...):
   statement1
   statement2
   ...
>>> def functionname(arg1,arg2):
...     return arg1+arg2
...
>>> t = functionname(24,24) # Result: 48
```

If a function takes no arguments, it must still include the parentheses, but without anything in them:

```
def functionname():
   statement1
   statement2
   ...
```

The four steps to defining a function in Python are the following:

1. Use the keyword def to declare the function and follow this up with the function name.
2. Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.
3. Add statements that the functions should execute.
4. End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

The arguments in the function definition bind the arguments passed at function invocation (i.e. when the function is called), which are called actual parameters, to the names given when the function is defined, which are called formal parameters. The interior of the function has no knowledge of the names given to the actual parameters; the names of the actual parameters may not even be accessible (they could be inside another function).

A function can 'return' a value, for example:

```
def square(x):
   return x*x
```

A function can define variables within the function body, which are considered 'local' to the function. The locals together with the arguments comprise all the variables within the scope of the function. Any names within the function are unbound when the function returns or reaches the end of the function body.

You can **return multiple values** as follows:

```
def first2items(list1):
  return list1[0], list1[1]
a, b = first2items(["Hello", "world", "hi", "universe"])
print a + " " + b
```

### 2.1.1 Function Calls:

A *callable object* is an object that can accept some arguments (also called parameters) and possibly return an object (often a tuple containing multiple objects).

A function is the simplest callable object in Python, but there are others, such as classes or certain class instances.

**Example:**

```
# Define a function `plus()`
def plus(a,b):
    return a + b

print("sum is :" plus(10,50))
```

**output:**
**sum is: 60**

### 2.1.3 Arguments:

The arguments of a function are defined within the **def** statement. Like all other variables in Python, there is no explicit type associated with the function arguments. This fact is important to consider when making assumptions about the types of data that your function will receive.

Function arguments can optionally be defined with a default value. The default value will be assigned in the case that the argument is not present in the call to the function. All arguments without default values must be listed before arguments with default values in the function definition.

**Declaring Arguments**

When calling a function that takes some values for further processing, we need to send some values as Function **Arguments**. For example:

```
>>> def find_max(a,b):
  if(a > b):
    print str(a) + " is greater than " + str(b)
  elif(b > a):
    print str(b) + " is greater than " + str(a)
>>> find_max(30, 45)  #Here (30, 45) are the arguments passing for finding max between this two numbers
```

The output will be: 45 **is** greater than 30

### 2.2 Function arguments in Python:

In Python, user-defined functions can take four different types of arguments. The argument types and their meanings, however, are pre-defined and can't be changed. The following are the four types of arguments and their rules.

- Required arguments
- Default argumengts
- Key word arguments
- Variable length arguments or Arbitrary Arguments

### 2.2.1 Required Arguments:

We can provide the arguments at the time of function calling. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Consider the following example.

*Example 1*

```
#the argument name is the required argument to the function func
def func(name):
```

```
    message = "Hi "+name;
    return message;
name = input("Enter the name?")
print(func(name))
```

**Output:**

```
Enter the name?John
Hi John
```

***Example 2***

```
#the function simple_interest accepts three arguments and returns the simple interest accordin
gly
def simple_interest(p,t,r):
    return (p*t*r)/100
p = float(input("Enter the principle amount? "))
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

**Output:**

```
Enter the principle amount? 10000
Enter the rate of interest? 5
Enter the time in years? 2
Simple Interest:  1000.0
```

**Example 3**

```
#the function calculate returns the sum of two arguments a and b
def calculate(a,b):
    return a+b
calculate(10) # this causes an error as we are missing a required arguments b.
```

**Output:**

```
TypeError: calculate() missing 1 required positional argument: 'b'
```

**2.2.2 Default Arguments:**

Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value.

For some functions, you may want to make some parameters *optional* and use default values in case the user does not want to provide values for them. This is done with the help of default argument values. You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value.
Note that the default argument value should be a constant. More precisely, the default argument value should be immutable – this is explained in detail in later chapters. For now, just remember this.

**Example:**

```
#!/usr/bin/python
# Filename: func_default.py

def say(message, times = 1):
    print(message * times)

say('Hello')
say('World', 5)
```

Output:

```
$ python func_default.py
Hello
WorldWorldWorldWorldWorld
```

**How It Works:**

The function named say is used to print a string as many times as specified. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of 1 to the parameter times.

In the first usage of say, we supply only the string and it prints the string once. In the second usage of say, we supply both the string and an argument 5 stating that we want to *say* the string message 5 times.

Important -Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list.This is because the values are assigned to the parameters by position. For example, def func(a, b=5) is valid, but def func(a=5, b) is *not valid*.

### 2.2.3 Keyword Arguments:

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them – this is called *keyword arguments* – we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* – one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

**Example:**

```
#!/usr/bin/python
# Filename: func_key.py

def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
$ python func_key.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

**How It Works:**

The function named func has one parameter without a default argument value, followed by two parameters with default argument values.

In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 7 and c gets the default value of 10.

In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument. Then, the parameter cgets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5.

In the third usage func(c=50, a=100), we use keyword arguments for all specified values. Notice that we are specifying the value for parameter c before that for a even though a is defined before c in the function definition.

### 2.2.4 Arbitrary Arguments or Variable length Arguments:

Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.

***Create function with variable length arguments***

Following is the syntax to create a function that can take variable length arguments.

```
def func(*args):
  #
  # body of the function
  #
```

Where, func is the name of the function and *args holds variable length arguments.

***Passing multiple arguments***

In the following Python program we are recreating the sum function but this time we are modifying it to take multiple arguments and print them.

**Example:**

```
# func
def sum(*args):
    print(type(args))
    print(args)
sum(10, 20)
sum(10, 20, 30)
```

```
sum(10, 20, 30, 40)
```

**Output.**

```
<class 'tuple'>
(10, 20)
<class 'tuple'>
(10, 20, 30)
<class 'tuple'>
(10, 20, 30, 40
```

So, we can see that the args variable is of type tuple and we are also able to print all the values that were passed to the function as a tuple.

***Accessing multiple arguments***

Since the multiple arguments passed to the function are tuple so we can access them using for loop.

In the following Python program we are printing out the individual argument.

**Example-2:**

```
# func
def sum(*args):
    for arg in args:
        print(arg)
print('sum(10, 20)')
sum(10, 20)
print('sum(10, 20, 30)')
sum(10, 20, 30)
print('sum(10, 20, 30, 40)')
sum(10, 20, 30, 40)
```

**Output:**

```
sum(10, 20)
10
20
sum(10, 20, 30)
10
20
30
sum(10, 20, 30, 40)
10
20
30
40
```

So, now that we are able to access the individual argument passed to the function let's go ahead and modify the sum function that we are working on to return us the sum of the arguments.

***Multiple arguments sum function***

**Example -3:**

```
# func
def sum(*args):
    result = 0
    for arg in args:
        result = result + arg
    return result

print(sum(10, 20))            # 30
print(sum(10, 20, 30))        # 60
print(sum(10, 20, 30, 40))    # 100
```

Let's add some checks to the sum function so that we only add arguments that are numbers.

```
# func
def sum(*args):
    result = 0
    for arg in args:
        if type(arg) in (int, float):
            result = result + arg
    return result
print(sum(10, 20.10, "hello"))          # 30.1
```

### 2.3. Scope of variables:

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

***Example 1***

```
def print_message():
    message = "hello !! I am going to print a message." # the variable message is local to the fu
nction itself
    print(message)
print_message()
print(message) # this will cause an error since a local variable cannot be accessible here.
```

**Output:**

```
hello !! I am going to print a message.
  File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in
    print(message)
NameError: name 'message' is not defined
```

***Example 2***

```
def calculate(*args):
   sum=0
   for arg in args:
      sum = sum +arg
   print("The sum is",sum)
sum=0
calculate(10,20,30) #60 will be printed as the sum
print("Value of sum outside the function:",sum) # 0 will be printed
```

**Output:**

```
The sum is 60
Value of sum outside the function: 0
```

### 2.4. Lambda Functions:

In Python, we use the lambda keyword to declare an anonymous function, which is why we refer to them as "lambda functions". An anonymous function refers to a function declared with no name. Although syntactically they look different, lambda functions behave in the same way as regular functions that are declared using the def keyword.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The following are the characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

#### 2.4.1 Creating a Lambda Function

We use the following syntax to declare a lambda function:

**lambda argument(s): expression**

As stated above, we can have any number of arguments but only a single expression. The lambda operator cannot have any statements and it returns a function object that we can assign to any variable.

**For example:**

**remainder = lambda num: num % 2**

**print(remainder(5))**
Output:
**1**

In this code the lambda num: num % 2 is the lambda function. The num is the argument while num % 2 is the expression that is evaluated and the result of the expression is returned. The expression gets the modulus of the input parameter by 2. Providing 5 as the parameter, which is divided by 2, we get a remainder of 1.

You should notice that the lambda function in the above script has not been assigned any name. It simply returns a function object which is assigned to the identifier remainder. However, despite being anonymous, it was possible for us to call it in the same way that we call a normal function. The statement:

**lambda num: num % 2**

Is similar to the following:

```
def remainder(num):
  return num % 2
```

Here is another example of a lambda function:

```
product = lambda x, y : x * y

print(product(2, 3))
```

**Output**
**6**

### 2.4.2 Why use lambda functions?

The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function. In python, the lambda function can be used as an argument to the higher order functions as arguments. Lambda functions are also used in the scenario where we need a Consider the following example.

**Example 1:**

```
#the function table(n) prints the table of n
def table(n):
   return lambda a:a*n; # a will contain the iteration variable i and a multiple of n is returne
d at each function call
n = int(input("Enter the number?"))
b = table(n) #the entered number is passed into the function table. b will contain a lambda fun
ction which is called again and again with the iteration variable i
for i in range(1,11):
   print(n,"X",i,"=",b(i)); #the lambda function b is called with the iteration variable i,
```

**Output:**

```
Enter the number?10
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
```

```
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
```

**Example 2**

Use of lambda function with filter

```
#program to filter out the list which contains odd numbers

List = {1,2,3,4,10,123,22}
Oddlist = list(filter(lambda x:(x%3 == 0),List)) # the list contains all the items of the list for which the lambda function evaluates to true
print(Oddlist)
```

**Output:**

```
[3, 123]
```

**Example 3**

Use of lambda function with map

```
#program to triple each number of the list using map
List = {1,2,3,4,10,123,22}
new_list = list(map(lambda x:x*3,List)) # this will return the triple of each item of the list an d add it to new_list
print(new_list)
```

**Output:**

```
[3, 6, 9, 12, 30, 66, 369]
```

### 2.5 Recursive Functions:

A function that calls itself is a recursive function. This method is used when a certain problem is defined in terms of itself. Although this involves iteration, using an iterative approach to solve such a problem can be tedious. The recursive approach provides a very concise solution to a seemingly complex problem. It looks glamorous but can be difficult to comprehend!

Overview of how recursive function works

1. Recursive function is called by some external code.
2. If the base condition is met then the program do something meaningful and exits.
3. Otherwise, function does some required processing and then call itself to continue recursion.

The most popular example of recursion is the calculation of the factorial. Mathematically the factorial is defined as: n! = n * (n-1)!

We use the factorial itself to define the factorial. Hence, this is a suitable case to write a recursive function. Let us expand the above definition for the calculation of the factorial value of 5.

```
5! = 5 X 4!
    5 X4 X 3!
    5 X4 X 3 X 2!
    5 X4 X 3 X  2 X 1!
    5 X4 X 3 X  2 X 1
  = 120
```

While we can perform this calculation using a loop, its recursive function involves successively calling it by decrementing the number until it reaches 1. The following is a recursive function to calculate the factorial.

**Example: Recursive Function**

```
def factorial(n):
    if n == 1:
        print(n)
        return 1
    else:
        print (n,'*', end=' ')
        return n * factorial(n-1)
```

The above recursive function can be called as below.

```
>>> factorial(5)
5 * 4 * 3 * 2 * 1
120
```

When the factorial function is called with 5 as argument, successive calls to the same function are placed, while reducing the value of 5. Functions start returning to their earlier call after the argument reaches 1. The return value of the first call is a cumulative product of the return values of all calls.

**Example -2:**

**Problem Description**

The program takes two numbers and finds the GCD of two numbers using recursion.

**Program/Source Code**

Here is source code of the Python Program to find the GCD of two numbers using recursion. The program output is also shown below.

```
def gcd(a,b):
    if(b==0):
        return a
    else:
        return gcd(b,a%b)
```

```
a=int(input("Enter first number:"))
b=int(input("Enter second number:"))
GCD=gcd(a,b)
print("GCD is: ")
print(GCD)
```

**Program Explanation**

1. User must enter two numbers.
2. The two numbers are passed as arguments to a recursive function.
3. When the second number becomes 0, the first number is returned.
4. Else the function is recursively called with the arguments as the second number and the remainder when the first number is divided by the second number.
5. The first number is then returned which is the GCD of the two numbers.
6. The GCD is then printed.

**Runtime Test Cases**

```
Case 1:
Enter first number:5
Enter second number:15
GCD is:
5

Case 2:
Enter first number:30
Enter second number:12
GCD is:
6
```

**Example-3:**

**Problem Description**

The program takes the number of terms and determines the fibonacci series using recursion upto that term.

**Program/Source Code**

Here is source code of the Python Program to find the fibonacci series using recursion. The program output is also shown below.

```
def fibonacci(n):
    if(n <= 1):
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
n = int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i in range(n):
    print fibonacci(i),
```

**Program Explanation**

1. User must enter the number of terms and store it in a variable.
2. The number is passed as an argument to a recursive function.
3. The base condition is that the number has to be lesser than or equal to 1.
4. Otherwise the function is called recursively with the argument as the number minus 1 added to the function called recursively with the argument as the number minus 2.
5. The result is returned and a for statement is used to print the fibonacci series.

**Runtime Test Cases**

```
Case 1:
Enter number of terms:5
Fibonacci sequence:
0 1 1 2 3

Case 2:
Enter number of terms:7
Fibonacci sequence:
0 1 1 2 3 5 8
```

### 2.5.1 Advantages of Python Recursion

1. Reduces unnecessary calling of function, thus reduces length of program.
2. Very flexible in data structure like stacks, queues, linked list and quick sort.
3. Big and complex iterative solutions are easy and simple with Python recursion.
4. Algorithms can be defined recursively making it much easier to visualize and prove.

### 2.5.2 Disadvantages of Python Recursion

1. Slow.
2. Logical but difficult to trace and debug.
3. Requires extra storage space. For every recursive calls separate memory is allocated for the variables.
4. Recursive functions often throw a Stack Overflow E

## 2.6 Python Modules:

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

**Example**
In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.
Let's create the module named as **file.py.**

```
#displayMsg prints a message to the name being passed.
def displayMsg(name)
   print("Hi "+name);
```

Here, we need to include this module into our main module to call the method displayMsg() defined in the module named file.

### 2.6.1 Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

- The import statement
- The from-import statement

### 2.6.2 The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is given below.

```
import module1,module2,..... module n
```

Hence, if we need to call the function displayMsg() defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

*Example:*

```
import file;
name = input("Enter the name?")
file.displayMsg(name)
```

**Output:**

```
Enter the name?John
Hi John
```

### 2.6.3 The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

```
from < module-name> import <name 1>, <name 2>..,<name n>
```

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

**calculation.py:**

```
#place the code in the calculation.py
def summation(a,b):
    return a+b
def multiplication(a,b):
    return a*b;
def divide(a,b):
    return a/b;
```

**Main.py:**

```
        from calculation import summation
#it will import only the summation() from calculation.py
a = int(input("Enter the first number"))
b = int(input("Enter the second number"))
print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessin
g summation()
```

**Output:**

```
Enter the first number10
Enter the second number20
Sum =  30
```

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *.

Consider the following syntax.

```
        from <module> import *
```

### 2.6.4 Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

```
        import <module-name> as <specific-name>
```

***Example***

```
#the module calculation of previous example is imported in this example as cal.
import calculation as cal;
```

```
a = int(input("Enter a?"));
b = int(input("Enter b?"));
print("Sum = ",cal.summation(a,b))
```

**Output:**

```
Enter a?10
Enter b?20
Sum =  30
```

### 2.6.5 The reload() function

As we have already stated that, a module is loaded once regardless of the number of times it is imported into the python source file. However, if you want to reload the already imported module to re-execute the top-level code, python provides us the reload() function. The syntax to use the reload() function is given below.

**reload(<module-name>)**

for example, to reload the module calculation defined in the previous example, we must use the following line of code.

reload(calculation)

### 2.6.6 Python packages

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path /**home**.

2. Create a python source file with name ITEmployees.py on the path /**home**/**Employees**.

**ITEmployees.py**

```
def getITNames():
    List = ["John", "David", "Nick",    "Martin"]
    return List;
```

3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is __init__.py which contains the import statements of the modules defined in this directory.

**__init__.py**

```
from ITEmployees import getITNames
from BPOEmployees import getBPONames
```

5. Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create __init__.py inside a directory to convert this directory to a package.

6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

**Test.py**

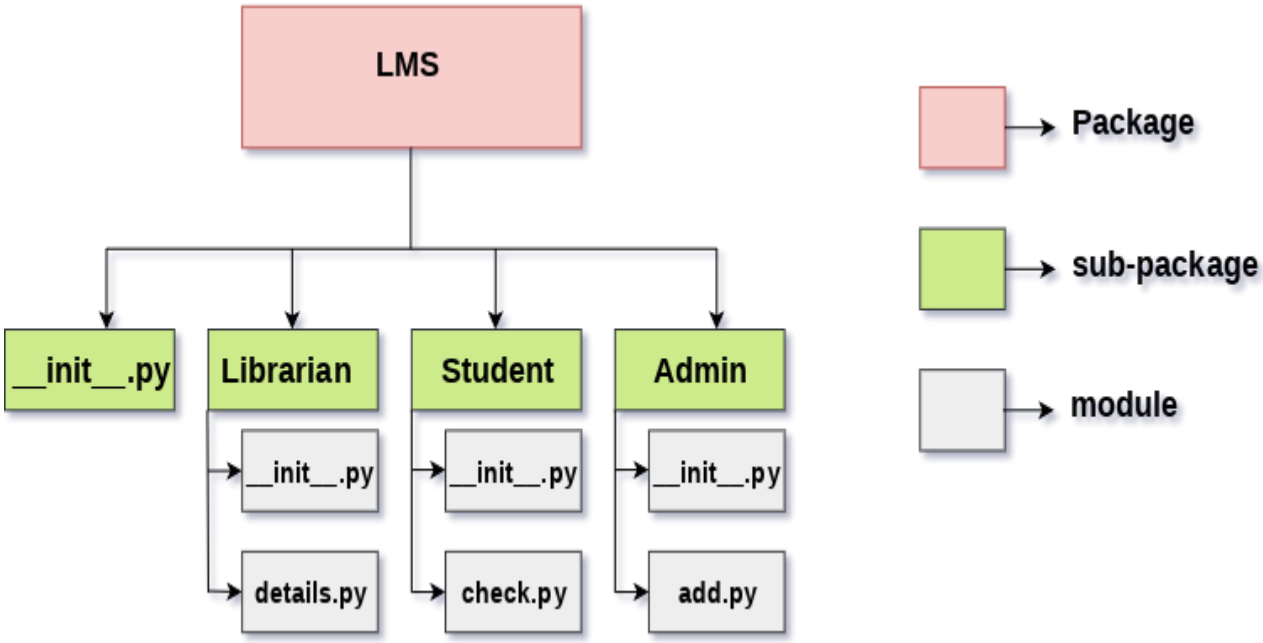
```
import Employees
print(Employees.getNames())
```

**Output:**

```
['John', 'David', 'Nick', 'Martin']
```

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.




## 2.7. DOCSTRING:

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

Python Docstring is the documentation string which is string literal, and it occurs in the class, module, function or method definition, and it is written as a first statement. Docstrings are accessible from the ***doc*** attribute for any of the Python object and also with the built-in help() function can come in handy.

Also, Docstrings are great for the understanding the functionality of the larger part of the code, i.e., the general purpose of any class, module or function whereas the comments are used for code, statement, and expressions which tend to be small. They are a descriptive text written by a programmer mainly for themselves to know what the line of code or expression does. It is an essential part that documenting your code is going to serve well enough for writing clean code and well-written programs. Though already mentioned there are no standard and rules for doing so.

There are two forms of writing a Docstring: one-line Docstrings and multi-line Docstrings. These are the documentation that is used by Data Scientists/programmers in their projects.

### 2.7.1 One-line Docstrings

The one-line Docstrings are the Docstrings which fits all in one line. You can use one of the quotes, i.e., triple single or triple double quotes and opening quotes and closing quotes need to be the same. In the one-line Docstrings, closing quotes are in the same line as with the opening quotes. Also, the standard convention is to use the triple-double quotes.

**Example:**

```
def square(a):
    '''Returns argument a is squared.'''
    return a**a

print (square.__doc__)

help(square)
Returns argument a is squared.
```

Help on function square in module __main__:

square(a)
Returns argument a is squared.

Here in the above code, you get the printed result:

Returns argument a is squared.

Help on function square in module __main__:

square(a)
Returns argument a is squared.

In the above Docstrings, you can observe that:

The line begins with a capital letter, i.e., R in our case and end with a period(".").

The closing quotes are on the same line as the opening quotes. This looks better for one-liners.

There's no blank line either before or after the Docstring. It is good practice.

The above line in quotes is more of command than a description which ends with a period sign at last.

### 2.7.2 Multi-line Docstrings

Multi-line Docstrings also contains the same string literals line as in One-line Docstrings, but it is followed by a single blank along with the descriptive text.

The general format for writing a Multi-line Docstring is as follows:

**Example:**

```
def some_function(argument1):
    """Summary or Description of the Function

    Parameters:
    argument1 (int): Description of arg1

    Returns:
    int:Returning value

    """

    return argument1

print(some_function.__doc__)
```

**Summary or Description of the Function**

```
    Parameters:
    argument1 (int): Description of arg1

    Returns:
    int:Returning value
```

**The above code outputs:**

```
Summary or Description of the Function

    Parameters:
    argument1 (int): Description of arg1

    Returns:
    int: Returning value
```

**Example:**

Let's look at the example which can show how the multi-line strings can be used in detail:

```
def string_reverse(str1):
    """ Returns the reversed String.

    Parameters:
       str1 (str):The string which is to be reversed.

    Returns:
       reverse(str1):The string which gets reversed.

    """
    reverse_str1 = ''
    i = len(str1)
    while i > 0:
       reverse_str1 += str1[ i - 1 ]
       i = i- 1
    return reverse_str1
print(string_reverse('projkal998580'))
```

**output:**

085899lakjorp

You can see above that the summary line is on one line and is also separated from other content by a single blank line. This convention needs to be followed which is useful for the automatic indexing tools.

### 2.7.3 Popular Docstring Formats

There are many Docstrings format available, but it is always better to use the formats which are easily recognized by the Docstring parser and also to fellow Data Scientist/ programmers. There is no any rules and regulations for selecting a Docstring format, but the consistency of choosing the same format over the project is necessary. Also, It is preferred for you to use the formatting type which is mostly supported by Sphinx.

The most common formats used are listed below.

| Formatting Type | Description |
|---|---|
| NumPy/SciPy docstrings | Combination of reStructured and GoogleDocstrings and supported by Sphinx |
| Pydoc | Standard documentation module for Python and supported by Sphinx |
| Epydoc | Render Epytext as series of HTML documents and a tool for generating API documentation for Python modules based on their Docstrings |
| Google Docstrings | Google's Style |

# 2.8 STRINGS & REGULAR EXPRESSIONS:

## 2.8.1 INTRODUCTION:

String is a collection of alphabets, words or other characters. It is one of the primitive data structures and are the building blocks for data manipulation. Python has a built-in string class named **str**.

Python strings are "immutable" which means they cannot be changed after they are created. For string manipulation, we create new strings as we go to represent computed values because of their immutable property.

## How to create a string and assign it to a variable

To create a string, put the sequence of characters inside either single quotes, double quotes, or triple quotes and then assign it to a variable.

```
# Python string examples - all assignments are identical.
        String_var = 'Python'
        String_var = "Python"
        String_var = """Python"""

# with Triple quotes Strings can extend to multiple lines
        String_var = """ This document will help you to
        explore all the concepts
        of Python Strings!!! """

# Replace "document" with "tutorial" and store in another variable
        substr_var = String_var.replace("document", "tutorial")
        print (substr_var)
```

Strings in python are immutable, means once string is created it can't be modified. Let's take an example to illustrate this point.

```
1  str1 = "welcome"
2  str2 = "welcome"
```

here str1 and str2 refers to the same string object "welcome" which is stored somewhere in memory. You can test whether str1 refers to same object as str2 using id() function.

**What is** id() : Every object in python is stored somewhere in memory. We can use id() to get that memory address.

```
1 >>> id(str1)
2 78965411
3 >>> id(str2)
4 78965411
```

### 2.8.2 Index And Slice Strings In Python & Access Individual Characters Of A String:

Like the most programming languages, Python allows to index from the zeroth position in Strings. But it also supports negative indexes. Index of '-1' represents the last character of the String. Similarly using '-2', we can access the penultimate element of the string and so on.

| P | Y | T | H | O | N | - | S | T | R | I | N | G |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

```
sample_str = 'Python String'

print (sample_str[0])       # return 1st character
# output: P

print (sample_str[-1])      # return last character
# output: g
print (sample_str[-2])      # return last second character
# output: n
```

### 2.8.3 Slice A String :

To retrieve a range of characters in a String, we use 'slicing operator,' the colon ':' sign. With the slicing operator, we define the range as [a:b]. It'll let us print all the characters of the String starting from index 'a' up to char at index 'b-1'. So the char at index 'b' is not a part of the output.

```
sample_str = 'Python String'
print (sample_str[3:5])           #return a range of character
# ho
print (sample_str[7:])      # return all characters from index 7
# String
print (sample_str[:6])      # return all characters before index 6
# Python
print (sample_str[7:-4])
# St
```

## 2.9 String Operators:

| Operator | Description |
|---|---|
| + | It is known as concatenation operator used to join the strings given either side of the operator. |
| * | It is known as repetition operator. It concatenates the multiple copies of the same string. |
| [] | It is known as slice operator. It is used to access the sub-strings of a particular string. |

| | |
|---|---|
| [:] | It is known as range slice operator. It is used to access the characters from the specified range. |
| In | It is known as membership operator. It returns if a particular sub-string is present in the specified string. |
| not in | It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string. |
| r/R | It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string. |
| % | It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python. |

### *Example*

Consider the following example to understand the real use of Python operators.

```
str = "Hello"
str1 = " world"
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
print(r'C://python37') # prints C://python37 as it is written
print("The string str : %s"%(str)) # prints The string str : Hello
```

**Output:**

```
HelloHelloHello
Hello world
o
ll
False
False
C://python37
The string str : Hello
```

### 2.9.1 Python Format Characters:

String '%' operator issued for formatting Strings. We often use this operator with the print() function.

Here's a simple example.

```
print ("Employee Name: %s,\nEmployee Age:%d" % ('RaviRaju',25))

# Employee Name: RaviRaju,
# Employee Age: 25
```

### List Of Format Symbols

Following is the table containing the complete list of symbols that you can use with the '%' operator.

| Symbol | Conversion |
|---|---|
| %c | character |
| %s | string conversion via str() before formatting |
| %i | signed decimal integer |
| %d | signed decimal integer |
| %u | unsigned decimal integer |
| %o | octal integer |
| %x | hexadecimal integer (lowercase letters) |
| %X | hexadecimal integer (UPPER-case letters) |
| %e | exponential notation (with lowercase 'e') |
| %E | exponential notation (with UPPER-case 'E') |
| %f | floating-point real number |
| %g | the shorter of %f and %e |
| %G | the shorter of %f and %E |

## 2.10 Built-In String Functions In Python:

## Conversion Functions:

**1. capitalize() –** Returns the string with the first character capitalized and rest of the characters in lower case.

```
var = 'PYTHON'
print (var.capitalize())
# Python
```

**2. lower() –** Converts all the characters of the String to lowercase

```
var = 'TechBeamers'
print (var.lower())
```

```
# techbeamers
```

**3. upper() –** Converts all the characters of the String to uppercase

```
var = 'TechBeamers'
print (var.upper())
# TECHBEAMERS
```

**4. swapcase() –** Swaps the case of every character in the String means that lowercase characters got converted to uppercase and vice-versa.

```
var = 'TechBeamers'
print (var.swapcase())
# tECHbEAMERS
```

**5. title() –** Returns the 'titlecased' version of String, which means that all words start with uppercase and the rest of the characters in words are in lowercase.

```
var = 'welcome to Python programming'
print (var.title())
# Welcome To Python Programming
```

**6. count( str[, beg [, end]]) –** Returns the number of times substring 'str' occurs in the range [beg, end] if beg and end index are given else the search continues in full String Search is case-sensitive.

```
var='TechBeamers'
str='e'
print (var.count(str))
# 3
var1='Eagle Eyes'
print (var1.count('e'))
# 2
var2='Eagle Eyes'
print (var2.count('E',0,5))
# 1
```

**Comparison Functions – Part1**

**1. islower() –** Returns 'True' if all the characters in the String are in lowercase. If any of the char is in uppercase, it will return False.

```
var='Python'
print (var.islower())
# False

var='python'
print (var.islower())
# True
```

**2. isupper() –** Returns 'True' if all the characters in the String are in uppercase. If any of the char is in lowercase, it will return False.

```
var='Python'
print (var.isupper())
# False

var='PYTHON'
```

```
print (var.isupper())
# True
```

**3. isdecimal() –** Returns 'True' if all the characters in String are decimal. If any character in the String is of other data-type, it will return False.
Decimal characters are those from the Unicode category Nd.

```
num=u'2016'
print (num.isdecimal())
# True
```

**4. isdigit() –** Returns 'True' for any char for which isdecimal() would return 'True and some characters in the 'No' category. If there are any characters other than these, it will return False'.
Precisely, digits are the characters for which Unicode property includes:
Numeric_Type=Digit or Numeric_Type=Decimal.

For example, superscripts are digits but fractions not.

```
print ('2'.isdigit())
# True

print ('²'.isdigit())
# True
```

**Comparison Functions – Part2**

**1. isnumeric() –** Returns 'True' if all the characters of the Unicode String lie in any one of the categories Nd', No, and NI.
If there are any characters other than these, it will return False.

Precisely, Numeric characters are those for which Unicode property includes:
Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

```
num=u'2016'
print (num.isnumeric())
# True

num=u'year2016'
print (num.isnumeric())
# False
```

**2. isalpha() –** Returns 'True' if String contains at least one character (non-empty String) and all the characters are alphabetic, 'False' otherwise.

```
print ('python'.isalpha())
# True

print ('python3'.isalpha())
# False
```

**3. isalnum() –** Returns 'True' if String contains at least one character (non-empty String) and all the characters are either alphabetic or decimal digits, 'False' otherwise.

```
print ('python'.isalnum())
# True
print ('python3'.isalnum())
# True
```

**Padding Functions**

**1. rjust(width[,fillchar]) –** Returns string filled with input char while pushing the original content on the right side.
By default, the padding uses a space. Otherwise 'fillchar' specifies the filler character.

```
var='Python'
print (var.rjust(10))
# Python

print (var.rjust(10,'-'))
# ----Python
```

**2. ljust(width[,fillchar]) –** Returns a padded version of String with the original String left-justified to a total of width columns
By default, the padding uses a space. Otherwise 'fillchar' specifies the filler character.

```
var='Python'
print (var.ljust(10))
# Python

print (var.ljust(10,'-'))
# Python----
```

**3. center(width[,fillchar]) –** Returns string filled with the input char while pushing the original content into the center.
By default, the padding uses a space. Otherwise 'fillchar' specifies the filler character.

```
var='Python'
print (var.center(20))
# Python

print (var.center(20,'*'))
# *******Python*******
```

**4. zfill(width) –** Returns string filled with the original content padded on the left with zeros so that total length of String becomes equal to the input size.
If there is a leading sign (+/-) present in the String, then with this function padding starts after the symbol, not before it.

```
var='Python'
print (var.zfill(10))
# 0000Python
```

```
var='+Python'
print (var.zfill(10))
# +000Python
```

**Search Functions**

**1. find(str [,i [,j]]) –** Searches for 'str' in complete String (if i and j not defined) or in a sub-string of String (if i and j are defined).This function returns the index if 'str' is found else returns '-1'.
Here, i=search starts from this index, j=search ends at this index.

```
var="Tech Beamers"
str="Beam"
print (var.find(str))
# 5

var="Tech Beamers"
str="Beam"
print (var.find(str,4))
# 5

var="Tech Beamers"
str="Beam"
print (var.find(str,7))
# -1
```

**2. index(str[,i [,j]]) –** This is same as 'find' method. The only difference is that it raises 'ValueError' exception if 'str' doesn't exist.

```
var='Tech Beamers'
str='Beam'
print (var.index(str))
# 5

var='Tech Beamers'
str='Beam'
print (var.index(str,4))
# 5

var='Tech Beamers'
str='Beam'
print (var.index(str,7))
# ValueError: substring not found
```

**3. rfind(str[,i [,j]]) –** This is same as find() just that this function returns the last index where 'str' is found. If 'str' is not found, it returns '-1'.

```
var='This is a good example'
str='is'
print (var.rfind(str,0,10))
```

```
# 5

print (var.rfind(str,10))
# -1
```

**4. count(str[,i [,j]]) –** Returns the number of occurrences of substring 'str' in the String. Searches for 'str' in the complete String (if i and j not defined) or in a sub-string of String (if i and j are defined).
Where: i=search starts from this index, j=search ends at this index.

```
var='This is a good example'
str='is'
print (var.count(str))
# 2

print (var.count(str,4,10))
# 1
```

**String Substitution Functions**

**1. replace(old,new[,count]) –** Replaces all the occurrences of substring 'old' with 'new' in the String.
If the count is available, then only 'count' number of occurrences of 'old' will be replaced with the 'new' var.

Where old =substring to replace, new =substring

```
var='This is a good example'
str='was'
print (var.replace('is',str))
# Thwas was a good example
print (var.replace('is',str,1))
# Thwas is a good example
```

**2. split([sep[,maxsplit]]) –** Returns a list of substring obtained after splitting the String with 'sep' as a delimiter.
Where, sep= delimiter, the default is space, maxsplit= number of splits to be done

```
var = "This is a good example"
print (var.split())
# ['This', 'is', 'a', 'good', 'example']
print (var.split(' ', 3))
# ['This', 'is', 'a', 'good example']
```

**3. splitlines(num) –** Splits the String at line breaks and returns the list after removing the line breaks.
Where, num = if this is a positive value. It indicates that line breaks will appear in the returned list.

```
var='Print new line\nNextline\n\nMove again to new line'
```

```
print (var.splitlines())
# ['Print new line', 'Nextline', '', 'Move again to new line']
print (var.splitlines(1))
# ['Print new line\n', 'Nextline\n', '\n', 'Move again to new line']
```

**4. join(seq) –** Returns a String obtained after concatenating the sequence 'seq' with a delimiter string.
Where: the seq= sequence of elements to join

```
seq=('ab','bc','cd')
str='='
print (str.join(seq))
# ab=bc=cd
```

**Misc String Functions**

**1. lstrip([chars]) –** Returns a string after removing the characters from the beginning of the String.
Where: Chars=this is the character to be trimmed from the String.

The default is whitespace character.

```
var=' This is a good example '
print (var.lstrip())
# This is a good example
var='*****This is a good example*****'
print (var.lstrip('*'))
# This is a good example**********
```

**2. rstrip() –** Returns a string after removing the characters from the End of the String.
Where: Chars=this is the character to be trimmed from the String. The default is whitespace character.

```
var=' This is a good example '
print (var.rstrip())
# This is a good example
var='*****This is a good example*****'
print (var.rstrip('*'))
# *****This is a good example
```

**3. rindex(str[,i [,j]]) –** Searches for 'str' in the complete String (if i and j not defined) or in a sub-string of String (if i and j are defined). This function returns the last index where 'str' is available.
If 'str' is not there, then it raises a ValueError exception.

Where: i=search starts from this index, j=search ends at this index.

```
var='This is a good example'
str='is'
```

```
print (var.rindex(str,0,10))
# 5
print (var.rindex(str,10))
# ValueError: substring not found
```

**4. len(string) –** Returns the length of given String

```
var='This is a good example'
print (len(var))
# 22
```

## 2.11 String constants:

| BUILT-IN FUNCTION | DESCRIPTION |
|---|---|
| string.ascii_letters | Concatenation of the ascii_lowercase and ascii_uppercase constants. |
| string.ascii_lowercase | Concatenation of lowercase letters |
| string.ascii_uppercase | Concatenation of uppercase letters |
| string.digits | Digit in strings |
| string.hexdigits | Hexadigit in strings |
| string.letters | concatenation of the strings lowercase and uppercase |
| string.lowercase | A string must contain lowercase letters. |
| string.octdigits | Octadigit in a string |
| string.punctuation | ASCII characters having punctuation characters. |
| string.printable | String of characters which are printable |
| String.endswith() | Returns True if a string ends with the given suffix otherwise returns False |

| | |
|---|---|
| String.startswith() | Returns True if a string starts with the given prefix otherwise returns False |
| String.isdigit() | Returns “True” if all characters in the string are digits, Otherwise, It returns “False”. |
| String.isalpha() | Returns “True” if all characters in the string are alphabets, Otherwise, It returns “False”. |
| string.isdecimal() | Returns true if all characters in a string are decimal. |
| str.format() | one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. |
| String.index | Returns the position of the first occurrence of substring in a string |
| string.uppercase | A string must contain uppercase letters. |
| string.whitespace | A string containing all characters that are considered whitespace. |
| string.swapcase() | Method converts all uppercase characters to lowercase and vice versa of the given string, and returns it |
| replace() | returns a copy of the string where all occurrences of a substring is replaced with another substring. |

## 2.12 String comparison:

You can use ( > , < , <= , <= , == , != ) to compare two strings. Python compares string lexicographically i.e using ASCII value of the characters.

Suppose you have str1 as "Mary" and str2 as "Mac" . The first two characters from str1 and str2 ( M and M ) are compared. As they are equal, the second two characters are compared. Because they are also equal, the third two characters ( r and c ) are compared. And because 'r' has greater ASCII value than 'c' , str1 is greater than str2 .

**Example:**

```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "right" >= "left"
True
>>> "teeth" < "tee"
False
>>> "yellow" <= "fellow"
False
>>> "abc" > ""
True
>>>
```

## 2.13 Regular Expressions:

A regular expression in a programming language is a special text string used for describing a search pattern. It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents.

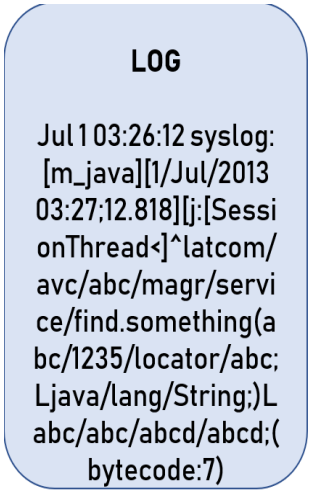
A Regular Expression is used for identifying a search pattern in a text string. It also helps in finding out the correctness of the data and even operations such as finding, replacing and formatting the data is possible using Regular Expressions.

***Why Use Regular Expression?***

we will look at the various problems faced by us which in turn is solved by using Regular Expressions.

Consider the following scenario:

You have a log file which contains a large sum of data. And from this log file, you wish to fetch only the date and time. As you can look at the image, readability of the log file is low upon first glance.



Regular Expressions can be used in this case to recognize the patterns and extract the required information easily.

Consider the next scenario – You are a salesperson and you have a lot of email addresses and a lot of those addresses are fake/invalid. Check out the image below:


What you can do is, you can make use of Regular Expressions you can verify the format of the email addresses and filter out the fake IDs from the genuine ones.

The next scenario is pretty similar to the one with the salesperson example. Consider the following image:

**Phone Number**
444-122-1234
123-122-78999
111-123-23
67-7890-2019

How do we verify the phone number and then classify it based on the country of origin?

Every correct number will have a particular pattern which can be traced and followed through by using Regular Expressions.

In Python, we have module "**re**" that helps with regular expressions. So you need to import library **re** before you can use regular expressions in Python.

Use this code --> Import re

### *2.14. Various methods of Regular Expressions:*

The 're' package provides multiple methods to perform queries on an input string. Here are the most commonly used methods, I will discuss:

1. re.match()
2. re.search()
3. re.findall()
4. re.split()
5. re.sub()
6. re.compile()

#### 2.14.1 re.match(*pattern, string*):

This method finds match if it occurs at start of the string.

For example,

```
import re
pattern = r"apple"
text = "apple I love apples"
match_object = re.match(pattern, text)
if match_object:
    print("Match found!")
else:
    print("No match found!")
```

output:
Match found!

#### 2.14.2 re.search(*pattern, string*):

It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only.

Example:

```
import re
```

```
pattern = r"apple"
text = "I love apples"
match_object = re.search(pattern, text)
if match_object:
    print("Match found!")
else:
    print("No match found!")
```

.

Output:
Match found!

### 2.14.3 re.findall (*pattern, string*):

It helps to get a list of all matching patterns. It has no constraints of searching from start or end. it can work like re.search() and re.match() both.

**Example:**

```
import re
pattern = r"apple"
text = "I love apple. apples are delicious."
matches = re.findall(pattern, text)
print(matches)
```

**Output:**

**['apple', 'apple']**

### 2.14.4 re.split(*pattern, string,* [*maxsplit=0*]):

This methods helps to split *string* by the occurrences of given *pattern*. It has default value of zero. In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string. Let's look at the example below:

**Example:**

```
import re
pattern = r"\s+"  # Matches one or more whitespace characters
text = "Hello    World!   How are you?"
substrings = re.split(pattern, text)
print(substrings)
```

Output:
['Hello', 'World!', 'How', 'are', 'you?']

### 2.14.5 re.sub(*pattern, repl, string*):

It helps to search a pattern and replace with a new sub string. If the pattern is not found, *string* is returned unchanged.

**Example:**

```
import re
pattern = r"apple"
replacement = "orange"
text = "I love apples. Apples are delicious."
modified_text, replacements = re.subn(pattern, replacement, text)
print(modified_text)
print(replacements)
```

**Output:**

**I love oranges. Apples are delicious.**

**1**

### 2.14.6 re.compile(*pattern, repl, string*):

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

**Example:**

```
import re
pattern = r"apple"
text = "I love apples"
regex = re.compile(pattern)
match_object = regex.search(text)
if match_object:
    print("Match found!")
else:
    print("No match found!
```

Output:

Match found!

### 2.14 .7 Basic patterns used in regular expression:

| SYMBOL | DESCRIPTION |
|---|---|
| . | dot matches any character except newline |
| \w | matches any word character i.e letters, alphanumeric, digits and underscore ( _ ) |

| SYMBOL | DESCRIPTION |
|---|---|
| \W | matches non word characters |
| \d | matches a single digit |
| \D | matches a single character that is not a digit |
| \s | matches any white-spaces character like \n, \t, spaces |
| \S | matches single non white space character |
| [abc] | matches single character in the set i.e either match a, b or c |
| [^abc] | match a single character other than a, b and c |
| [a-z] | match a single character in the range a to z. |
| [a-zA-Z] | match a single character in the range a-z or A-Z |
| [0-9] | match a single character in the range 0-9 |
| ^ | match start at beginning of the string |
| $ | match start at end of the string |
| + | matches one or more of the preceding character (greedy match). |
| * | matches zero or more of the preceding character (greedy match). |

**2.6.8 Program to verify phone number using regulr expressions**

**Phone Number Verification:**

Problem Statement – The need to easily verify phone numbers in any relevant scenario.

Consider the following Phone numbers:

- 444-122-1234
- 123-122-78999
- 111-123-23
- 67-7890-2019

The general format of a phone number is as follows:

- Starts with 3 digits and '-' sign
- 3 middle digits and '-' sign
- 4 digits in the end

We will be using w in the example below. Note that **w = [a-zA-Z0-9_]**

```
1    import re
```

```
2
3       phn = "412-555-1212"
4
5       if re.search("w{3}-w{3}-w{4}", phn):
6           print("Valid phone number")
```

**Output:**

Valid phone number

### 2.6.9 program to verify E-mail address:

**E-mail Verification:**

Problem statement – To verify the validity of an E-mail address in any scenario.

Consider the following examples of email addresses:

- raviraj@gmail.com
- ravi @ com
- AC .com
- 123 @.com

Manually, it just takes you one good glance to identify the valid mail IDs from the invalid ones. But how is the case when it comes to having our program do this for us? It is pretty simple considering the following guidelines are followed for this use-case.

Guidelines:

All E-mail addresses should include:

- 1 to 20 lowercase and/or uppercase letters, numbers, plus . _ % +
- An @ symbol
- 2 to 20 lowercase and uppercase letters, numbers and plus
- A period symbol
- 2 to 3 lowercase and uppercase letters

Code:

```
To Validate Email
import re
def validate_email(email):
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    if re.match(pattern, email):
        print (email,"- valid email address")
    else:
        print (email, "Invalid email address")
email1 = 'example@example.com'
email2 = 'invalid.email'
email3 = 'another_example@gmail.com'
```

```
validate_email(email1)
validate_email(email2)
validate_email(email3)

Output:
example@example.com - valid email address
invalid.email Invalid email address
another_example@gmail.com - valid email address
```

This basically proves how simple and efficient it is to work with Regular Expressions and make use of them practically.

### 2.15 Lists:

A list in Python, also known as a sequence, is an ordered collection of objects. It can hold values of any data types such as numbers, letters, strings, and even the nested lists as well.

Every element rests at some position (i.e., index) in the list. The index can be used later to locate a particular item. The first index begins at zero, next is one, and so forth.

Unlike **strings**, lists are mutable (or changeable) in Python. It implies that you can replace, add or remove elements.

#### 2.15.1 Create A List In Python:

There are multiple ways to form a list in Python.

Using [ ] To Create A List

The first is by placing all the items (elements) inside a **square bracket [ ]**, separated by commas.

```
L1 = [] # empty list
L2 = [expression, ...]
```

The list can take any number of elements, and each may belong to a different type (integer, float, string, etc.).

List Creation – Examples

```
# blank list
L1 = []

# list of integers
L2 = [10, 20, 30]

# List of heterogenous data types
L3 = [1, "Hello", 3.4]
```

**Python List – List() Method To Create A List**

Python includes a built-in **list()** method.

It accepts either a sequence or tuple as the argument and converts into a Python list.

Let's start with an example to create a list without any element.

```
>>> theList = list() #empty list
>>> len(theList)
0
```

**Note-** The **len() function** returns the size of the list.
You can supply a standard or nested sequence as the input argument to the **list() function**.

```
theList = list([n1, n2, ...] or [n1, n2, [x1, x2, ...]])
```

List Creation – Examples
Try out the below Python list examples.

```
>>> theList = list([1,2])
>>> theList
[1, 2]
>>> theList = list([1, 2, [1.1, 2.2]])
>>> theList
[1, 2, [1.1, 2.2]]
>>> len(theList)
3
```

### 2.15.2 Using List Comprehension – An Intuitive Way To Create Lists

Python supports a concept known as "**List Comprehension**." It helps in constructing lists in a completely natural and easy way.
A list comprehension has the following syntax:

```
#Syntax - How to use List Comprehension

theList = [expression(iter) for iter in oldList if filter(iter)]
```

It has square brackets grouping an expression followed by a for-in clause and zero or more if statements. The result will always be a list.

**List Comprehension – Examples**

```
> theList = [iter for iter in range(5)]
>>> print(theList)
[0, 1, 2, 3, 4]
```

Isn't that was easy to start with a simple list.

Here is a more complicated example of **List Comprehension** resulting in list creation.

```
>>> listofCountries = ["India","America","England","Germany","Brazil","Vietnam"]
>>> firstLetters = [ country[0] for country in listofCountries ]
>>> print(firstLetters)
['I', 'A', 'E', 'G', 'B', 'V']
```

List comprehension even allows an if statement, to only add members to the list which are fulfilling a specific condition:

```
>>> print ([x+y for x in 'get' for y in 'set'])
['gs', 'ge', 'gt', 'es', 'ee', 'et', 'ts', 'te', 'tt']
```

Let's now see how the if clause works with the list comprehension.

```
>>> print ([x+y for x in 'get' for y in 'set' if x != 't' and y != 'e' ])
['gs', 'gt', 'es', 'et']
```

Another complicated example is to create a list containing the odd months with List Comprehension syntax.

```
>>> months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov', 'dec']
>>> oddMonths = [iter for index, iter in enumerate(months) if (index%2 == 0)]
>>> oddMonths
['jan', 'mar', 'may', 'jul', 'sep', 'nov']
```

**2.15.3 Creating A Multi-Dimensional List**

You can create a sequence with a pre-defined size, by specifying an initial value for each element.

```
>>> init_list = [0]*3
>>> print(init_list)
[0, 0, 0]
```

With the above concept, you can build a two-dimensional list.

```
two_dim_list = [ [0]*3 ] *3
```

The above statement works but Python will only create the references as sublists instead of creating separate objects.

```
>>> two_dim_list = [ [0]*3 ] *3
>>> print(two_dim_list)
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> two_dim_list[0][2] = 1
>>> print(two_dim_list)
[[0, 0, 1], [0, 0, 1], [0, 0, 1]]
```

We changed the value of the third item in the first row, but the same column in other rows also got affected.

So, you must use list comprehensions to get around the above issue.

```
>>> two_dim_list = [[0]*3 for i in range(3)]

>>> print(two_dim_list)

[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

>>> two_dim_list[0][2] = 1
```

```
>>> print(two_dim_list)

[[0, 0, 1], [0, 0, 0], [0, 0, 0]]
```

### 2.15.4 Extend A List Using The Extend() And Append() Methods

Python allows lists to re-size in many ways. You can do that just by adding two or more of them.

```
>>> L1 = ['a', 'b']
>>> L2 = [1, 2]
>>> L3 = ['Learn', 'Python']
>>> L1 + L2 + L3
['a', 'b', 1, 2, 'Learn', 'Python']
```

**List Extend() Example**

Alternately, you can join lists using the **extend() method**.

```
>>> L1 = ['a', 'b']
>>> L2 = ['c', 'd']
>>> L1.extend(L2)
>>> print(L1)
['a', 'b', 'c', 'd']
```

**List Append() Example**

Next, you can append a value to a list by calling the **append() method**. See the below example.

```
>>> L1 = ['x', 'y']
>>> L1.append(['a', 'b'])
>>> L1
['x', 'y', ['a', 'b']]
```

### 2.15.5 Access A List In Python (Indexing The List)

You'll find many ways of accessing or indexing the elements of a Python list.

**Using The Index Operator**

The simplest one is to use the **index operator ([ ])** to access an element from the list. Since the list has zero as the first index, so a list of size ten will have indices from 0 to 9. Any attempt to access an item beyond this range would result in an **IndexError**. The index is always an integer. Using any other type of value will lead to **TypeError**.
Also, note that a Nested list will follow the nested indexing.

Access A List Using Index – Example

```
vowels = ['a','e','i','o','u']
consonants = ['b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'y', 'z']

#Accessing list elements using the index operator
print(vowels[0])
print(vowels[2])
print(vowels[4])
```

```
#Testing exception if the index is of float type
try:
 vowels[1.0]
except Exception as ex:
 print("Note:", ex)

#Accessing elements from the nested list
alphabets = [vowels, consonants]

print(alphabets[0][2])
print(alphabets[1][2])
```

**Output –**

```
a
i
u
Note: list indices must be integers or slices, not float
i
d
```

**Reverse Indexing**

Python enables reverse (or negative) indexing for the sequence data type. So, for Python list to index in the opposite order, you need to set the index using the minus (-) sign.

Indexing the list with "-1" will return the last element of the list, -2 the second last and so on.

```
vowels = ['a','e','i','o','u']

print(vowels[-1])

print(vowels[-3])
```

**Output –**

```
u
i
```

**2.15.6 Slicing A Python List:**

Python comes with a magical slice operator which returns the part of a sequence. It operates on objects of different **data types** such as strings, tuples, and works the same on a Python list.

It has a mystic syntax which is as follows.

**#The Python slicing operator syntax**

**[start(optional):stop(optional):step(optional)]**

**Say size => Total no. of elements in the list.**

**Start (x) -> It is the point (xth list index) where the slicing begins. (0 =< x < size, By default included in the slice output)**

**Stop (y) -> It is the point (y-1 list index) where the slicing ends. (0 < y <= size, The element at the yth index doesn't appear in the slice output)**

**Step (s) -> It is the counter by which the index gets incremented to return the next element. The default counter is 1.**

Let's consider the following list of integers.
>>> theList = [1, 2, 3, 4, 5, 6, 7, 8]
In the examples to follow, we'll test various slice operations on this list.
**Slicing Examples**
Return The Three Elements, I.E. [3, 4, 5] From The List

```
>>> theList[2:5]
[3, 4, 5]
```

Since Python list follows the zero-based index rule, so the first index starts at 0.

Hence, you can see that we passed '2' as starting index as it contains the value '3' which is included in the slice by default.

And passing '5' as the ending index meant to ensure the slice output can include elements up to the 4th index.

Print Slice As [3, 5], Don't Change The First Or Last Index

```
>>> theList[2:5:2]
[3, 5]
```

In this example, we incremented the step counter by '2' to exclude the median value, i.e., '4' from the slice output.

**Slice From The Third Index To The Second Last Element**
You can use a negative value for the stop argument. It means the traversing begins from the rearmost index.

A negative stop value such as '-1' would mean the same as "length minus one."

```
>>> theList[2:-1]
[3, 4, 5, 6, 7]
```

**Get The Slice From Start To The Second Index**
In slicing, if you don't mention the "start" point, then it indicates to begin slicing from the 0th index.

```
>>> theList[:2]
[1, 2]
```

**Slice From The Second Index To The End**

While slicing a list, if the stop value is missing, then it indicates to perform slicing to the end of the list. It saves us from passing the length of the list as the ending index.

```
>>> theList[2:]
[3, 4, 5, 6, 7, 8]
```

**Reverse A List Using The Slice Operator**

It is effortless to achieve this by using a special slice syntax (**::-1**). But please note that it is more memory intensive than an in-place reversal.

Here, it creates a shallow copy of the Python list which requires long enough space for holding the whole list.

```
>>> theList[::-1]
[8, 7, 6, 5, 4, 3, 2, 1]
```

Here, you need a little pause and understand, why is the '-1' after the second colon? It intends to increment the index every time by -1 and directs to traverse in the backward direction.

**Reverse A List But Leaving Values At Odd Indices**

Here, you can utilize the concept learned in the previous example.

```
>>> theList[::-2]
[8, 6, 4, 2]
```

We can skip every second member by setting the iteration to '-2'.

Create A Shallow Copy Of The Full List

```
>>> id(theList)
55530056
>>> id(theList[::])
55463496
```

Since all the indices are optional, so we can leave them out. It'll create a new copy of the sequence.

Copy Of The List Containing Every Other Element

```
>>> theList[::2]
[1, 3, 5, 7]
```

### 2.15 .7 Iterate A List In Python

Python provides a traditional **for-in loop** for iterating the list. The "for" **statement** makes it super easy to process the elements of a list one by one.

```
for element in theList:
 print(element)
```

If you wish to use both the index and the element, then call the **enumerate() function**.

```
for index, element in enumerate(theList):
 print(index, element)
```

If you only want the index, then call the range() and len() methods.

```
for index in range(len(theList)):
```

```
 print(index)
```

The list elements support the iterator protocol. To intentionally create an iterator, call the **built-in iter function**.

```
it = iter(theList)
element = it.next() # fetch first value
element = it.next() # fetch second value
```

Check out the below example.

**Traversing A List – Example**

```
theList = ['Python', 'C', 'C++', 'Java', 'CSharp']

for language in theList:
 print("I like", language)
```

**Output –**

```
I like Python
I like C
I like C++
I like Java
I like CSharp
```

**2.15.8 Add/Update Elements To A List:**

Unlike the string or tuple, Python list is a mutable object, so the values at each index can be modified.

You can use the **assignment operator (=)** to update an element or a range of items.

**Modifying A List Using Assignment Operator**

```
theList = ['Python', 'C', 'C++', 'Java', 'CSharp']

theList[4] = 'Angular'
print(theList)

theList[1:4] = ['Ruby', 'TypeScript', 'JavaScript']
print(theList)
```

**Output –**

```
['Python', 'C', 'C++', 'Java', 'Angular']
['Python', 'Ruby', 'TypeScript', 'JavaScript', 'Angular']
```

You can also refer the **adding/extending the list** section for updating the list.

**Modifying A List With Insert Method**

You can also push one item at the target location by calling the **insert() method**.

```
theList = [55, 66]

theList.insert(0,33)

print(theList)
```

**Output –**

```
[33, 55, 66]
```

To insert multiple items, you can use the slice assignment.

```
theList = [55, 66]

theList[2:2] = [77, 88]

print(theList)
```

**Output –**

```
[55, 66, 77, 88]
```

**Remove/Delete Elements From A List**

You can make use of the **'del' keyword** to remove one or more items from a list. Moreover, it is also possible to delete the entire list object.

**Delete List Elements Using Del Operator**

```
vowels = ['a','e','i','o','u']
# remove one item
del vowels[2]
# Result: ['a', 'e', 'o', 'u']
print(vowels)
# remove multiple items
del vowels[1:3]
# Result: ['a', 'u']
print(vowels)
# remove the entire list
del vowels

# NameError: List not defined
print(vowels)
```

**Delete List Elements Using Remove() And POP()**

You can call **remove() method** to delete the given element or the **pop() method** to take out an item from the desired index.

The **pop() method** deletes and sends back the last item in the absence of the index value. That's how you can define lists as stacks (i.e., FILO – First in, last out model).

```
vowels = ['a','e','i','o','u']

vowels.remove('a')

# Result: ['e', 'i', 'o', 'u']
print(vowels)

# Result: 'i'
print(vowels.pop(1))
```

```
# Result: ['e', 'o', 'u']
print(vowels)

# Result: 'u'
print(vowels.pop())

# Result: ['e', 'o']
print(vowels)

vowels.clear()

# Result: []
print(vowels)
```

Last but not least, you can also remove items by assigning a blank list with a slice of its elements.

```
vowels = ['a','e','i','o','u']

vowels[2:3] = []
print(vowels)

vowels[2:5] = []
print(vowels)
```

**Output-**

```
['a', 'e', 'o', 'u']
['a', 'e']
```

**Searching Elements In A List**

You can use the **Python 'in' operator** to check if an item is present in the list.

```
if value in theList:
    print("list contains", value)
```

Using the Python list **index() method**, you can find out the position of the first matching item.

```
loc = theList.index(value)
```

The index method performs a linear search and breaks after locating the first matching item. If the search ends without a result, then it throws a **ValueError** exception.

```
try:
    loc = theList.index(value)
except ValueError:
    loc = -1 # no match
```

If you would like to fetch the index for all matching items, then call **index()** in a loop by passing two arguments – the value and a starting index.

```
loc = -1
try:
    while 1:
        loc = theList.index(value, loc+1)
```

```
        print("match at", loc)
except ValueError:
    pass
```

A better version of the above code is to wrap the search logic inside a function and call that function from a loop.

Python list supports two methods **min(List)** and **max(List).** You can call them accordingly to find out the element carrying the minimum or the maximum value.

```
>>> theList = [1, 2, 33, 3, 4]
>>> low = min(theList)
>>> low
1
>>> high = max(theList)
>>> high
33
```

**2.15. 9 Sorting A List In Python:**

List's Sort() Method

Python list implements the **sort()** method for ordering (in both ascending and descending order) its elements in place.

```
theList.sort()
```

Please note that in place sorting algorithms are more efficient as they don't need temporary variables (such as a new list) to hold the result.

By default, the function **sort()** performs sorting in the ascending sequence.

```
theList = ['a','e','i','o','u']
theList.sort()
print(theList)
['a', 'e', 'i', 'o', 'u']
```

If you wish to sort in descending order, then refer the below example.

```
theList = ['a','e','i','o','u']
theList.sort(reverse=True)
print(theList)
['u', 'o', 'i', 'e', 'a']
```

Built-In Sorted() Function

You can use the **built-in sorted()** function to return a copy of the list with its elements ordered.

```
newList = sorted(theList)
```

By default, it also sorts in an ascending manner.

```
theList = ['a','e','i','o','u']
newList = sorted(theList)
print("Original list:", theList, "Memory addr:", id(theList))
print("Copy of the list:", newList, "Memory addr:", id(newList))
Original list: ['a', 'e', 'i', 'o', 'u'] Memory addr: 55543176
```

```
Copy of the list: ['a', 'e', 'i', 'o', 'u'] Memory addr: 11259528
```

You can turn on the **"reverse" flag** to **"True"** for enabling the descending order.

```
theList = ['a','e','i','o','u']
newList = sorted(theList, reverse=True)
print("Original list:", theList, "Memory addr:", id(theList))
print("Copy of the list:", newList, "Memory addr:", id(newList))
Original list: ['a', 'e', 'i', 'o', 'u'] Memory addr: 56195784
Copy of the list: ['u', 'o', 'i', 'e', 'a'] Memory addr: 7327368
```

### 2.15.10 Python List Methods:

| List Methods | Description |
|---|---|
| append() | It adds a new element to the end of the list. |
| extend() | It extends a list by adding elements from another list. |
| insert() | It injects a new element at the desired index. |
| remove() | It deletes the desired element from the list. |
| pop() | It removes as well as returns an item from the given position. |
| clear() | It flushes out all elements of a list. |
| index() | It returns the index of an element that matches first. |
| count() | It returns the total no. of elements passed as an argument. |
| sort() | It orders the elements of a list in an ascending manner. |
| reverse() | It inverts the order of the elements in a list. |
| copy() | It performs a shallow copy of the list and returns. |

### 2.15 .11 Python List Built-In Functions

| Function | Description |
|---|---|
| all() | It returns True if the list has elements with a True value or is blank. |
| any() | If any of the members has a True value, then it also returns True. |
| enumerate() | It returns a tuple with an index and value of all the list elements. |
| len() | The return value is the size of the list. |
| list() | It converts all iterable objects and returns as a list. |
| max() | The member having the maximum value |
| min() | The member having the minimum value |
| sorted() | It returns the sorted copy of the list. |
| sum() | The return value is the aggregate of all elements of a list. |

## 2.16 Tuples:

We saw earlier that a list is an **ordered mutable** collection. There's also an ordered **immutable** collection.

In Python these are called tuples and look very similar to lists, but typically written with () instead of []:

```
a_list = [1, 'two', 3.0]
```

```
a_tuple = (1, 'two', 3.0)
```

Similar to how we used list before, you can also create a tuple via tuple(1,2,3).

The difference being that tuples are immutable. This means no assignment, append, insert, pop, etc. Everything else works as it did with lists: indexing, getting the length, checking membership, etc.

Like lists, all of the common sequence operations are available.

Another thing to note is that strictly speaking, the comma is what makes the tuple, not the parentheses. In practice it is a good idea to include the parentheses for clarity and because they are needed in some situations to make operator precedence clear.

Let's look at a quick example:

```
>>> a_tuple = (1, 2, 3)

>>> another_tuple = 1, 2, 3

>>> a_tuple == b_tuple

True
```

This is also important if you need to make a single element tuple:

```
>>> x = ('one')

>>> y = ('one',)

>>> type(x)

str

>>> type(y)

tuple
```

- Tuples are lighter-weight and are more memory efficient and often faster if used in appropriate places.
- When using a tuple you protect against accidental modification when passing it between functions.
- Tuples, being immutable, can be used as a key in a dictionary, which we're about to learn about.

### 2.16.1 Constructing tuples:

To create a tuple, place values within brackets:

```
>>> l = (1, 2, 3)
>>> l[0]
1
```

It is also possible to create a tuple without parentheses, by using commas:

```
>>> l = 1, 2
>>> l
(1, 2)
```

If you want to create a tuple with a single element, you must use the comma:

```
>>> singleton = (1, )
```

You can repeat a tuples by multiplying a tuple by a number:

```
>>> (1,) * 5
(1, 1, 1, 1, 1)
```

Note that you can concatenate tuples and use augmented assignement (*=, +=):

```
>>> s1 = (1,0)
>>> s1 += (1,)
>>> s1
(1, 0, 1)
```

### 2.16.2 Tuple methods

Tuples are optimised, which makes them very simple objects. There are two methods available only:

- *index, to find occurence of a value*
- *count, to count the number of occurence of a value*

```
>>> l = (1,2,3,1)
>>> l.count(1)
2
>>> l.index(2)
1
```

### 2.8.3 Interests of tuples

So, Tuples are useful because there are

- *faster than lists*
- *protect the data, which is immutable*
- *tuples can be used as keys on dictionaries*

In addition, it can be used in different useful ways:

Tuples as key/value pairs to build dictionaries

```
>>> d = dict([('jan', 1), ('feb', 2), ('march', 3)])
>>> d['feb']
2
```

assigning multiple values

```
>>> (x,y,z) = ('a','b','c')
>>> x
'a'
>>> (x,y,z) = range(3)
>>> x
0
```

### 2.16.3.1. Tuple Unpacking

Tuple unpacking allows to extract tuple elements automatically is the list of variables on the left has the same number of elements as the length of the tuple

```
>>> data  = (1,2,3)
>>> x, y, z = data
>>> x
1
```

### 2.16.3.2 Tuple can be use as swap function

This code reverses the contents of 2 variables x and y:

```
>>> (x,y) = (y,x)
```

**Warning**

Consider the following function:

```
def swap(a, b):
    (b, a) = (a, b)
```

then:

```
a = 2
b = 3
swap(a, b)
#a is still 2 and b still 3 !! a and b are indeed passed by value not reference.
```

## 2.16.4 Misc

### 2.16.4.1. length

To find the length of a tuple, you can use the **len()** function:

```
>>> t= (1,2)
>>> len(t)
2
```

### 2.16.4.2 Slicing (extracting a segment)

```
>>> t = (1,2,3,4,5)
>>> t[2:]
(3, 4, 5)
```

### 2.16.4.3 Copy a tuple

To copy a tuple, just use the assignement:

```
>>> t = (1, 2, 3, 4, 5)
>>> newt = t
>>> t[0] = 5
>>> newt
(1, 2, 3, 4, 5)
```

**Warning**

You cannot copy a list with the = sign because lists are mutables. The = sign creates a reference not a copy. Tuples are immutable therefore a = sign does not create a reference but a copy as expected.

### 2.16.4.4. Tuple are not fully immutable !!

If a value within a tuple is mutable, then you can change it:

```
>>> t = (1, 2, [3, 10])
```

```
>>> t[2][0] = 9
>>> t
(1, 2, [9, 10])
```

### 2.16.4.5 Convert a tuple to a string

You can convert a tuple to a string with either:

```
>>> str(t)
```

or

```
>>> `t`
```

### 2.16.4.6. math and comparison

comparison operators and mathematical functions can be used on tuples. Here are some examples:

```
>>> t = (1, 2, 3)
>>> max(t)
3
```

## 2.16. 5 Tuple indexing and splitting

The indexing and slicing in tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the slice operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.

Tuple = ( 0, 1, 2, 3, 4, 5 )

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|

Tuple[0] = 0

Tuple[1] = 1

Tuple[2] = 2

Tuple[3] = 3

Tuple[4] = 4

Tuple[5] = 5

Tuple[0:] = (0, 1, 2, 3, 4, 5)

Tuple[:] = (0, 1, 2, 3, 4, 5)

Tuple[2:4] = (2, 3)

Tuple[1:3] = (1, 2)

Tuple[:4] = (0, 1, 2, 3)

Unlike lists, the tuple items can not be deleted by using the del keyword as tuples are immutable. To delete an entire tuple, we can use the del keyword with the tuple name.

Consider the following example.

```
tuple1 = (1, 2, 3, 4, 5, 6)
print(tuple1)
del tuple1[0]
print(tuple1)
del tuple1
print(tuple1)
```

**Output:**

```
(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "tuple.py", line 4, in <module>
    print(tuple1)
NameError: name 'tuple1' is not defined
```

Like lists, the tuple elements can be accessed in both the directions. The right most element (last) of the tuple can be accessed by using the index -1. The elements from left to right are traversed using the negative indexing.

Consider the following example.

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
print(tuple1[-4])
```

**Output:**

```
5
2
```

### 2.16.6 Basic Tuple operations:

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

| Operator | Description | Example |
|---|---|---|
| Repetition | The repetition operator enables the tuple elements to be repeated multiple times. | T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5) |

| | | |
|---|---|---|
| Concatenation | It concatenates the tuple mentioned on either side of the operator. | T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9) |
| Membership | It returns true if a particular item exists in the tuple otherwise false. | print (2 in T1) prints True. |
| Iteration | The for loop is used to iterate over the tuple elements. | for i in T1:<br>print(i)<br>**Output**<br>1<br>2<br>3<br>4<br>5 |
| Length | It is used to get the length of the tuple. | len(T1) = 5 |

### 2.16.7 Python Tuple inbuilt functions

| SN | Function | Description |
|---|---|---|
| 1 | cmp(tuple1, tuple2) | It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false. |
| 2 | len(tuple) | It calculates the length of the tuple. |
| 3 | max(tuple) | It returns the maximum element of the tuple. |
| 4 | min(tuple) | It returns the minimum element of the tuple. |
| 5 | tuple(seq) | It converts the specified sequence to the tuple. |

### 2.16.8 List VS Tuple

| SN | List | Tuple |
|---|---|---|

| | | |
|---|---|---|
| 1 | The literal syntax of list is shown by the []. | The literal syntax of the tuple is shown by the (). |
| 2 | The List is mutable. | The tuple is immutable. |
| 3 | The List has the variable length. | The tuple has the fixed length. |
| 4 | The list provides more functionality than tuple. | The tuple provides less functionality than the list. |
| 5 | The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed. | The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary. |

### 2.16.9 Nesting List and tuple

We can store list inside tuple or tuple inside the list up to any number of level.

Lets see an example of how can we store the tuple inside the list.

```
Employees = [(101, "Ayush", 22), (102, "john", 29), (103, "james", 45), (104, "Ben", 34)]
print("----Printing list----");
for i in Employees:
   print(i)
Employees[0] = (110, "David",22)
print();
print("----Printing list after modification----");
for i in Employees:
   print(i)
```

**Output:**

```
----Printing list----
(101, 'Ayush', 22)
(102, 'john', 29)
(103, 'james', 45)
(104, 'Ben', 34)

----Printing list after modification----

(110, 'David', 22)
(102, 'john', 29)
(103, 'james', 45)
(104, 'Ben', 34)
```

## 2.17. Dictionary:

Dictionaries are a common feature of modern languages (often known as maps, associative arrays, or hashmaps) which let you associate pairs of values together. The term dictionary is a nod to this, as you can think of them as being terms/descriptions in a sense.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any python object whereas the keys are the immutable python object, i.e., Numbers, string or tuple.

- A dictionary in Python is an unordered set of *key: value* pairs.
- Unlike lists, dictionaries are inherently **orderless**. The *key: value* pairs appear to be in a certain order but it is irrelevant.
- Each KEY must be unique, but the VALUES may be the same for two or more keys.
- If you assign a value to a key then later in the same dictionary have the same key assigned to a new value, the previous value will be overwritten.
- Instead of using a number to index the items (as you would in a list), you must use the specific key,

### 2.17.1 Creating a Dictionary:

Dictionary is listed in curly brackets, inside these curly brackets, keys and values are declared. Each key is separated from its value by a colon (:) while each element is separated by commas.

The keys would need to be of an immutable type, i.e., data-types for which the keys cannot be changed at runtime such as int, string, tuple, etc. The values can be of any type. Individual pairs will be separated by a comma(“,”) and the whole thing will be enclosed in curly braces({...}).

**Properties of Dictionary Keys**

There are two important points while using dictionary keys

- More than one entry per key is not allowed ( no duplicate key is allowed)
- The values in the dictionary can be of any type while the keys must be immutable like numbers, tuples or strings.
- Dictionary keys are case sensitive- Same key name but with the different case are treated as different keys in Python dictionaries.

For example, you can have the fields “city”, “name,” and “food” for keys in a dictionary and assign the key,value pairs to the dictionary variable person1_information.

```
>>> person_information = {'city': 'San Francisco', 'name': 'Sam', "food": "shrimps"}
>>> type(person1_information)
<class 'dict'>
>>> print(person1_information)
{'city': 'San Francisco', 'name': 'Sam', 'food': 'shrimps'}
```

***Get the values in a Dictionary***

Example, a dictionary is initialized with keys “city”, “name,” and “food” and you can retrieve the value corresponding to the key “city.”

```
>>> create a dictionary person1_information
>>> person1_information = {'city': 'San Francisco', 'name': 'Sam', "food":        "shrimps"}
>>> print the dictionary
>>> print(person1_information["city"])
San Francisco
```

You can also use the get method to retrieve the values in a dict. The only difference is that in the get method, you can set a default value. In direct referencing, if the key is not present, the interpreter throws KeyError.

```
>>> # create a small dictionary
>>> alphabets = {1: ‘a’}
>>> # get the value with key 1
>>> print(alphabets.get(1))
'a'
>>> # get the value with key 2. Pass “default” as default. Since key 2 does not exist, you get “default” as the return value.
>>> print(alphabets.get(2, "default"))
'default'
>>> # get the value with key 2 through direct referencing
>>> print(alphabets[2])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 2
```

A dictionary key must be of immutable data-type so lists cannot be used as a key. If you want to use a list as a key, you must first convert it to an immutable type. One way to do this is to convert the list to a corresponding tuple, which is a data-type very similar to a list.

**2.17.2 A list can be converted to a tuple and vice versa as in the following example:**

```
>>> a = [1,2,3]
>>> b = tuple(a)
>>> c = list(b)
```

Once a list is converted to a tuple, it can be used as a dictionary key.

```
>>> a = [1,2,3]
>>> b = tuple(a)
>>> d = {b:1}
```

We can add a new key-value pair to a dictionary as follows:

```
>>> a = {}
>>> a[1]=’a’
```

In the first line above, we created an empty dictionary.
In the next line, we added a new key-value pair, where 1 is the key and ‘a’ is the value.

We can remove a key-value pair from a dictionary by using del as follows:

>>> del a[1]

Below are some useful methods for dictionary, where d is a dictionary object.

d.keys() Returns the list of keys in the dictionary.
d.has_key(x) Returns whether the dictionary has a key x.
d.items() Returns a list of key-value pairs in the dictionary.
d.values() Returns a list of values in the dictionary.
d.get(x,y) Returns d[x], if the dictionary has a key x.
Otherwise, it returns y.

### 2.17.3 Built-in Dictionary functions:

The built-in python dictionary methods along with the description are given below.

| SN | Function | Description |
|---|---|---|
| 1 | cmp(dict1, dict2) | It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false. |
| 2 | len(dict) | It is used to calculate the length of the dictionary. |
| 3 | str(dict) | It converts the dictionary into the printable string representation. |
| 4 | type(variable) | It is used to print the type of the passed variable. |

### 2.9.4 Built-in Dictionary methods

The built-in python dictionary methods along with the description are given below.

| SN | Method | Description |
|---|---|---|
| 1 | clear() | Removes all Items |
| 2 | copy() | Returns Shallow Copy of a Dictionary |
| 3 | fromkeys() | creates dictionary from given sequence |
| 4 | get() | Returns Value of The Key |
| 5 | items() | returns view of dictionary's (key, value) pair |
| 6 | keys() | Returns View Object of All Keys |
| 7 | popitem() | Returns & Removes Element From Dictionary |

| SN | Method | Description |
|---|---|---|
| 8 | setdefault() | Inserts Key With a Value if Key is not Present |
| 9 | pop() | removes and returns element having given key |
| 10 | values() | returns view of all values in dictionary |
| 11 | update() | Updates the Dictionary |
| 12 | any() | Checks if any Element of an Iterable is True |
| 13 | all() | returns true when all elements in iterable is true |
| 14 | ascii() | Returns String Containing Printable Representation |
| 15 | bool() | Converts a Value to Boolean |
| 16 | dict() | Creates a Dictionary |
| 17 | enumerate() | Returns an Enumerate Object |
| 18 | filter() | constructs iterator from elements which are true |
| 19 | iter() | returns iterator for an object |
| 20 | len() | Returns Length of an Object |
| 21 | max() | returns largest element |
| 22 | min() | returns smallest element |
| 23 | map() | Applies Function and Returns a List |
| 24 | sorted() | returns sorted list from a given iterable |
| 25 | sum() | Add items of an Iterable |
| 26 | zip() | Returns an Iterator of Tuples |

**Lists** and **dictionaries** are two of the most frequently used Python types. As you have seen, they have several similarities, but differ in how their elements are accessed. Lists elements are accessed by numerical index based on order, and dictionary elements are accessed by key

# Python Built-In Functions with examples for to improve coding

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

| | |
|---|---|
| frozenset() | Returns a frozenset object |
| getattr() | Returns the value of the specified attribute (property or method) |
| globals() | Returns the current global symbol table as a dictionary |
| hasattr() | Returns True if the specified object has the specified attribute (property/method) |
| hash() | Returns the hash value of a specified object |
| help() | Executes the built-in help system |
| hex() | Converts a number into a hexadecimal value |
| id() | Returns the id of an object |
| input() | Allowing user input |
| int() | Returns an integer number |
| isinstance() | Returns True if a specified object is an instance of a specified object |
| issubclass() | Returns True if a specified class is a subclass of a specified object |
| iter() | Returns an iterator object |
| len() | Returns the length of an object |
| list() | Returns a list |
| locals() | Returns an updated dictionary of the current local symbol table |
| map() | Returns the specified iterator with the specified function applied to each item |
| max() | Returns the largest item in an iterable |
| memoryview() | Returns a memory view object |
| min() | Returns the smallest item in an iterable |
| next() | Returns the next item in an iterable |
| object() | Returns a new object |
| oct() | Converts a number into an octal |
| open() | Opens a file and returns a file object |
| ord() | Convert an integer representing the Unicode of the specified character |
| pow() | Returns the value of x to the power of y |

| | |
|---|---|
| print() | Prints to the standard output device |
| property() | Gets, sets, deletes a property |
| range() | Returns a sequence of numbers, starting from 0 and increments by 1 (by default) |
| repr() | Returns a readable version of an object |
| reversed() | Returns a reversed iterator |
| round() | Rounds a numbers |
| set() | Returns a new set object |
| setattr() | Sets an attribute (property/method) of an object |
| slice() | Returns a slice object |
| sorted() | Returns a sorted list |
| @staticmethod() | Converts a method into a static method |
| str() | Returns a string object |
| sum() | Sums the items of an iterator |
| tuple() | Returns a tuple |
| type() | Returns the type of an object |
| vars() | Returns the __dict__ property of an object |
| zip() | Returns an iterator, from two or more iterators |

**1. abs()**

The abs() is one of the most popular Python built-in functions, which returns the absolute value of a number. A negative value's absolute is that value is positive.

```
>>> abs(-7)
7
>>> abs(7)
7
>>> abs(0)
```

**2. all()**

The all() function takes a container as an argument. This Built in Functions returns True if all values in a python iterable have a Boolean value of True. An empty value has a Boolean value of False.

```
>>> all({'*',','})
False
>>> all([' ',' ',' '])
True
```

**3. any()**
Like all(), it takes one argument and returns True if, even one value in the iterable has a Boolean value of True.

```
>>> any((1,0,0))
True
>>> any((0,0,0))
False
```

**4. ascii()**
It is important Python built-in functions, returns a printable representation of a **python object** (like a string or a **Python list**). Let's take a Romanian character.

```
>>> ascii('ș')
"'\\u0219'"
```

Since this was a non-ASCII character in python, the interpreter added a backslash (\) and escaped it using another backslash.

```
>>> ascii('ușor')
"'u\\u0219or'"
```

Let's apply it to a list.

```
>>> ascii(['s','ș'])
"['s', '\\u0219']"
```

**5. bin()**
bin() converts an integer to a binary string. We have seen this and other functions in our article on **Python Numbers**.

```
>>> bin(7)
'0b111'
```

We can't apply it on floats, though.

```
>>> bin(7.0)
Traceback (most recent call last):
File "<pyshell#20>", line 1, in <module>
bin(7.0)
TypeError: 'float' object cannot be interpreted as an integer
```

**6. bool()**
bool() converts a value to Boolean.

```
>>> bool(0.5)
True
>>> bool('')
False
>>> bool(True)
True
```

**7. bytearray()**
bytearray() returns a python array of a given byte size.

```
>>> a=bytearray(4)
>>> a
bytearray(b'\x00\x00\x00\x00')
>>> a.append(1)
>>> a
bytearray(b'\x00\x00\x00\x00\x01')
```

```
>>> a[0]=1
>>> a
bytearray(b'\x01\x00\x00\x00\x01')
>>> a[0]
1
```

Let's do this on a list.

```
>>> bytearray([1,2,3,4])
bytearray(b'\x01\x02\x03\x04')
```

**8. bytes():**

bytes() returns an immutable bytes object.

```
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
>>> bytes([1,2,3,4,5])
b'\x01\x02\x03\x04\x05'
>>> bytes('hello','utf-8')
b'hello'
```

Here, utf-8 is the encoding.

Both bytes() and bytearray() deal with raw data, but bytearray() is mutable, while bytes() is immutable.

```
>>> a=bytes([1,2,3,4,5])
>>> a
b'\x01\x02\x03\x04\x05'
>>> a[4]=
3
Traceback (most recent call last):
File "<pyshell#46>", line 1, in <module>
a[4]=3
TypeError: 'bytes' object does not support item assignment
```

Let's try this on bytearray().

```
>>> a=bytearray([1,2,3,4,5])
>>> a
bytearray(b'\x01\x02\x03\x04\x05')
>>> a[4]=3
>>> a
bytearray(b'\x01\x02\x03\x04\x03')
```

**9. callable()**

callable() tells us if an object can be called.

```
>>> callable([1,2,3])
False
>>> callable(callable)
True
>>> callable(False)
False
>>> callable(list)
True
```

A function is callable, a list is not. Even the callable() python Built In function is callable.

**10. chr()**

chr() Built In function returns the character in python for an ASCII value.

```
>>> chr(65)
'A'
>>> chr(97)
'a'
>>> chr(9)
'\t'
>>> chr(48)
'0'
```

**11. classmethod()**

classmethod() returns a class method for a given method.

```
>>> class fruit:
def sayhi(self):
print("Hi, I'm a fruit")

>>> fruit.sayhi=classmethod(fruit.sayhi)
>>> fruit.sayhi()
Hi, I'm a fruit
```

When we pass the method sayhi() as an argument to classmethod(), it converts it into a python class method one that belongs to the class. Then, we call it like we would call any static **method in python** without an object.

**12. compile()**

compile() returns a Python code object. We use Python in built function to convert a string code into object code.

```
>>> exec(compile('a=5\nb=7\nprint(a+b)','','exec'))
12
```

Here, 'exec' is the mode. The parameter before that is the filename for the file form which the code is read.
Finally, we execute it using exec().

**13. complex()**

complex() function creates a complex number. We have seen this is our article on **Python Numbers**.

```
>>> complex(3)
(3+0j)
>>> complex(3.5)
(3.5+0j)
>>> complex(3+5j)
(3+5j)
```

**14. delattr()**

delattr() takes two arguments- a class, and an attribute in it. It deletes the attribute.

```
>>> class fruit:
size=7

>>> orange=fruit()
>>> orange.size
```

```
7
>>> delattr(fruit,'size')
>>> orange.size
Traceback (most recent call last):
File "<pyshell#95>", line 1, in <module>
orange.size
AttributeError: 'fruit' object has no attribute 'size'
```

**15. dict()**

dict(), as we have seen it, creates a **python dictionary**.

```
>>> dict()
{}
>>> dict([(1,2),(3,4)])
{1: 2, 3: 4}
```

This was about dict() Python Built In function

**16. dir()**

dir() returns an object's attributes.

```
>>> class fruit:
size=7
shape='round'
>>> orange=fruit()
>>> dir(orange)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'shape', 'size']
```

**17. divmod()**

divmod() in Python built-in functions, takes two parameters, and returns a tuple of their quotient and remainder. In other words, it returns the floor division and the modulus of the two numbers.

```
>>> divmod(3,7)
(0, 3)
>>> divmod(7,3)
(2, 1)
```

If you encounter any doubt in Python Built-in Function, Please Comment.

**18. enumerate()**

This Python Built In function returns an enumerate object. In other words, it adds a counter to the iterable.

```
>>> for i in enumerate(['a','b','c']):
print(i)
(0, 'a')
(1, 'b')
(2, 'c')
```

**19. eval()**

This Function takes a string as an argument, which is parsed as an expression.

```
>>> x=7
>>> eval('x+7')
14
>>> eval('x+(x%2)')
8
```

**20. exec()**

exec() runs Python code dynamically.

```
>>> exec('a=2;b=3;print(a+b)')
5
>>> exec(input("Enter your program"))
Enter your programprint(2+3)
5
```

**21. filter()**

Like we've seen in **python Lambda Expressios**, filter() filters out the items for which the condition is True.

```
>>> list(filter(lambda x:x%2==0,[1,2,0,False]))
[2, 0, False]
```

**22. float()**

This Python Built In function converts an int or a compatible value into a float.

```
>>> float(2)
2.0
>>> float('3')
3.0
>>> float('3s')
Traceback (most recent call last):
File "<pyshell#136>", line 1, in <module>
float('3s')
ValueError: could not convert string to float: '3s'
>>> float(False)
0.0
>>> float(4.7)
4.7
```

**23. format()**

We have seen this Python built-in function, one in our lesson on **Python Strings**.

```
>>> a,b=2,3
>>> print("a={0} and b={1}".format(a,b))
a=2 and b=3
>>> print("a={a} and b={b}".format(a=3,b=4))
a=3 and b=4
```

**24. frozenset()**

frozenset() returns an immutable frozenset object.

```
>>> frozenset((3,2,4))
frozenset({2, 3, 4})
```

Read **Python Sets and Booleans** for more on frozenset.

**25. getattr()**
getattr() returns the value of an object's attribute.

```
>>> getattr(orange,'size')
7
```

**26. globals()**
This Python built-in functions, returns a dictionary of the current global symbol table.

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'fruit': <class '__main__.fruit'>, 'orange': <__main__.fruit object at 0x05F937D0>, 'a': 2, 'numbers': [1, 2, 3], 'i': (2, 3), 'x': 7, 'b': 3}
```

**27. hasattr()**
Like delattr() and getattr(), hasattr() Python built-in functions, returns True if the object has that attribute.

```
>>> hasattr(orange,'size')
True
>>> hasattr(orange,'shape')
True
>>> hasattr(orange,'color')
False
```

**28. hash()**
hash() function returns the hash value of an object. And in Python, everything is an object.

```
>>> hash(orange)
6263677
>>> hash(orange)
6263677
>>> hash(True)
1
>>> hash(0)
0
>>> hash(3.7)
644245917
>>> hash(hash)
25553952
```

This was all about hash() Python In Built function

**29. help()**
To get details about any module, keyword, symbol, or topic, we use the help() function.

```
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.
```

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

```
help> map
Help on class map in module builtins:
class map(object)
| map(func, *iterables) --> map object
|
| Make an iterator that computes the function using arguments from
| each of the iterables. Stops when the shortest iterable is exhausted.
|
| Methods defined here:
|
| __getattribute__(self, name, /)
| Return getattr(self, name).
|
| __iter__(self, /)
| Implement iter(self).
|
| __new__(*args, **kwargs) from builtins.type
| Create and return a new object. See help(type) for accurate signature.
|
| __next__(self, /)
| Implement next(self).
|
| __reduce__(...)
| Return state information for pickling.
help> You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

**30. hex()**

Hex() Python built-in functions, converts an integer to hexadecimal.

```
>>> hex(16)
‘0x10’
>>> hex(False)
‘0x0’
```

**31. id() Function**

id() returns an object’s identity.

```
>>> id(orange)
100218832
```

```
>>> id({1,2,3})==id({1,3,2})
True
```

**32. input()**

Input() Python built-in functions, reads and returns a line of string.

```
>>> input("Enter a number")
Enter a number7
'7'
```

Note that this returns the input as a string. If we want to take 7 as an integer, we need to apply the int() function to it.

```
>>> int(input("Enter a number"))
Enter a number7
7
```

**33. int()**

int() converts a value to an integer.

```
>>> int('7')
7
```

**34. isinstance()**

We have seen this one in previous lessons. isinstance() takes a variable and a class as arguments. Then, it returns True if the variable belongs to the class. Otherwise, it returns False.

```
>>> isinstance(0,str)
False
>>> isinstance(orange,fruit)
True
```

**35. issubclass()**

This Python Built In function takes two arguments- two **python classes**. If the first class is a subclass of the second, it returns True. Otherwise, it returns False.

```
>>> issubclass(fruit,fruit)
True
>>> class fruit:
pass
>>> class citrus(fruit):
pass
>>> issubclass(fruit,citrus)
False
```

**36. iter()**

Iter() Python built-in functions, returns a **python iterator** for an object.

```
>>> for i in iter([1,2,3]):
print(i)
1
2
3
```

**37. len()**

We've seen len() so many times by now. It returns the length of an object.

```
>>> len({1,2,2,3})
3
```
Here, we get 3 instead of 4, because the set takes the value '2' only once.

**38. list()**

list() creates a list from a sequence of values.

```
>>> list({1,3,2,2})
[1, 2, 3]
```

**39. locals()**

This function returns a dictionary of the current local symbol table.

```
>>> locals()
```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'fruit': <class '__main__.fruit'>, 'orange': <__main__.fruit object at 0x05F937D0>, 'a': 2, 'numbers': [1, 2, 3], 'i': 3, 'x': 7, 'b': 3, 'citrus': <class '__main__.citrus'>}

**40. map()**

Like filter(), map() Python built-in functions, takes a function and applies it on an iterable. It maps True or False values on each item in the iterable.

```
>>> list(map(lambda x:x%2==0,[1,2,3,4,5]))
[False, True, False, True, False]
```

**41. max()**

A no-brainer, max() returns the item, in a sequence, with the highest value of all.

```
>>> max(2,3,4)
4
>>> max([3,5,4])
5
>>> max('hello','Hello')
'hello'
```

**42. memoryview()**

memoryview() shows us the memory view of an argument.

```
>>> a=bytes(4)
>>> memoryview(a)
<memory at 0x05F9A988>
>>> for i in memoryview(a):
print(i)
```

**43. min()**

min() returns the lowest value in a sequence.

```
>>> min(3,5,1)
1
>>> min(True,False)
False
```

**44. next()**

This Python Built In function returns the next element from the iterator.

```
>>> myIterator=iter([1,2,3,4,5])
>>> next(myIterator)
1
>>> next(myIterator)
2
>>> next(myIterator)
3
>>> next(myIterator)
4
>>> next(myIterator)
5
```

Now that we've traversed all items, when we call next(), it raises StopIteration.

```
>>> next(myIterator)
Traceback (most recent call last):
File "<pyshell#392>", line 1, in <module>
next(myIterator)
StopIteration
```

**45. object()**

Object() Python built-in functions, creates a featureless object.

```
>>> o=object()
>>> type(o)
<class 'object'>
>>> dir(o)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__'] Here, the function type() tells us that it's an object. dir() tells us the object's attributes. But since this does not have the __dict__ attribute, we can't assign to arbitrary attributes.
```

**46. oct()**

oct() converts an integer to its octal representation.

```
>>> oct(7)
'0o7'
>>> oct(8)
'0o10'
>>> oct(True)
'0o1'
```

**47. open()**

open() lets us open a file. Let's change the current working directory to Desktop.

```
>>> import os
>>> os.chdir('C:\\Users\\lifei\\Desktop')
```

Now, we open the file 'topics.txt'.

```
>>> f=open('topics.txt')
>>> f
<_io.TextIOWrapper name='topics.txt' mode='r' encoding='cp1252'>
>>> type(f)
```

<class '_io.TextIOWrapper'>
To read from the file, we use the read() method.
```
>>> print(f.read())
DBMS mappings
projection
union
rdbms vs dbms
doget dopost
how to add maps
OOT
SQL queries
Join
Pattern programs
Output
Default constructor in inheritance
```

**48. ord()**

The function ord() returns an integer that represents the Unicode point for a given Unicode character.

```
>>> ord('A')
65
>>> ord('9')
57
```

This is complementary to chr().

```
>>> chr(65)
'A'
```

**49. pow()**

pow() takes two arguments- say, x and y. It then returns the value of x to the power of y.

```
>>> pow(3,4)
81
>>> pow(7,0)
1
>>> pow(7,-1)
0.14285714285714285
>>> pow(7,-2)
0.02040816326530612
```

**50. print()**

We don't think we need to explain this anymore. We've been seeing this function since the beginning of this article.

```
>>> print("Okay, next function, please!")
Okay, next function, please!
```

**51. property()**

The function property() returns a property attribute. Alternatively, we can use the syntactic sugar @property. We will learn this in detail in our tutorial on **Python Property**.

**52. range()**

We've taken a whole tutorial on this. Read up **range() in Python**.

```
>>> for i in range(7,2,-2):
print(i)
7
5
3
```

**53. repr()**

repr() returns a representable string of an object.

```
>>> repr("Hello")
"'Hello"'
>>> repr(7)
'7'
>>> repr(False)
'False'
```

**54. reversed()**

This functions reverses the contents of an iterable and returns an iterator object.

```
>>> a=reversed([3,2,1])
>>> a
<list_reverseiterator object at 0x02E1A230>
>>> for i in a:
print(i)
1
2
3
>>> type(a)
<class 'list_reverseiterator'>
```

**55. round()**

round() rounds off a float to the given number of digits (given by the second argument).

```
>>> round(3.777,2)
3.78
>>> round(3.7,3)
3.7
>>> round(3.7,-1)
0.0
>>> round(377.77,-1)
380.0
```

The rounding factor can be negative.

**56. set()**

Of course, set() returns a set of the items passed to it.

```
>>> set([2,2,3,1])
{1, 2, 3}
```

Remember, a set cannot have duplicate values, and isn't indexed, but is ordered. Read on Sets and Booleans for the same.

**57. setattr()**

Like getattr(), setattr() sets an attribute's value for an object.

```
>>> orange.size
```

```
7
>>> orange.size=8
>>> orange.size
8
```

**58. slice()**

slice() returns a slice object that represents the set of indices specified by range(start, stop, step).

```
>>> slice(2,7,2)
slice(2, 7, 2)
```

We can use this to iterate on an iterable like a **string in python**.

```
>>> 'Python'[slice(1,5,2)]
'yh'
```

**59. sorted()**

Like we've seen before, sorted() prints out a sorted version of an iterable. It does not, however, alter the iterable.

```
>>> sorted('Python')
['P', 'h', 'n', 'o', 't', 'y']
>>> sorted([1,3,2])
[1, 2, 3]
```

**60. staticmethod()**

staticmethod() creates a static method from a function. A static method is bound to a class rather than to an object. But it can be called on the class or on an object.

```
>>> class fruit:
def sayhi():
print("Hi")
>>> fruit.sayhi=staticmethod(fruit.sayhi)
>>> fruit.sayhi()
Hi
```

You can also use the syntactic sugar @staticmethod for this.

```
>>> class fruit:
@staticmethod
def sayhi():
print("Hi")
>>> fruit.sayhi()
Hi
```

**61. str()**

str() takes an argument and returns the string equivalent of it.

```
>>> str('Hello')
'Hello'
>>> str(7)
'7'
>>> str(8.7)
'8.7'
>>> str(False)
'False'
>>> str([1,2,3])
```

‘[1, 2, 3]’

**62. sum()**

The function sum() takes an iterable as an argument, and returns the sum of all values.

```
>>> sum([3,4,5],3)
15
```

**63. super()**

super() returns a proxy object to let you refer to the parent class.

```
>>> class person:
def __init__(self):
print("A person")
>>> class student(person):
def __init__(self):
super().__init__()
print("A student")
>>> Avery=student()
A person
A student
```

**64. tuple()**

As we’ve seen in our tutorial on Python Tuples, the function tuple() lets us create a tuple.

```
>>> tuple([1,3,2])
(1, 3, 2)
>>> tuple({1:'a',2:'b'})
(1, 2)
```

**65. type()**

We have been seeing the type() function to check the type of object we’re dealing with.

```
>>> type({})
<class ‘dict’>
>>> type(set())
<class ‘set’>
>>> type(())
<class ‘tuple’>
>>> type((1))
<class ‘int’>
>>> type((1,))
<class ‘tuple’>
```

**66. vars()**

vars() function returns the __dict__ attribute of a class.

```
>>> vars(fruit)
mappingproxy({‘__module__’: ‘__main__’, ‘size’: 7, ‘shape’: ’round’, ‘__dict__’: <attribute ‘__dict__’ of ‘fruit’ objects>, ‘__weakref__’: <attribute ‘__weakref__’ of ‘fruit’ objects>, ‘__doc__’: None})
```

**67. zip()**

zip() returns us an iterator of tuples.

```
>>> set(zip([1,2,3],['a','b','c']))
```

{(1, ‘a’), (3, ‘c’), (2, ‘b’)}
>>> **set**(**zip**([1,2],[3,4,5]))
{(1, 3), (2, 4)}
>>> a=**zip**([1,2,3],['a','b','c'])
To unzip this, we write the following code.
>>> x,y,z=a
>>> x
(1, ‘a’)
>>> y
(2, ‘b’)
>>> z
(3, ‘c’)

## 1.11. Implementation of classes and objects in Python:

A class is a collection of objects. Unlike the primitive data structures, classes are data structures that the user defines. They make the code more manageable.

### 1.11.1 Creating Class and Objects:

In Python, Use the keyword class to define a Class. In the class definition, the first string is docstring which, is a brief description of the class.
The docstring is not mandatory but recommended to use. We can get docstring using __doc__ attribute. Use the following syntax to create a class.

**Syntax**

```
class classname:
    '''documentation string'''
   class_suite
```

- **Documentation string**: represent a description of the class. It is optional.
- **class_suite**: class suite contains class attributes and methods

We can create any number of objects of a class. use the following syntax to create an object of a class.

```
reference_variable = classname()
```

**OOP Example: Creating Class and Object in Python**

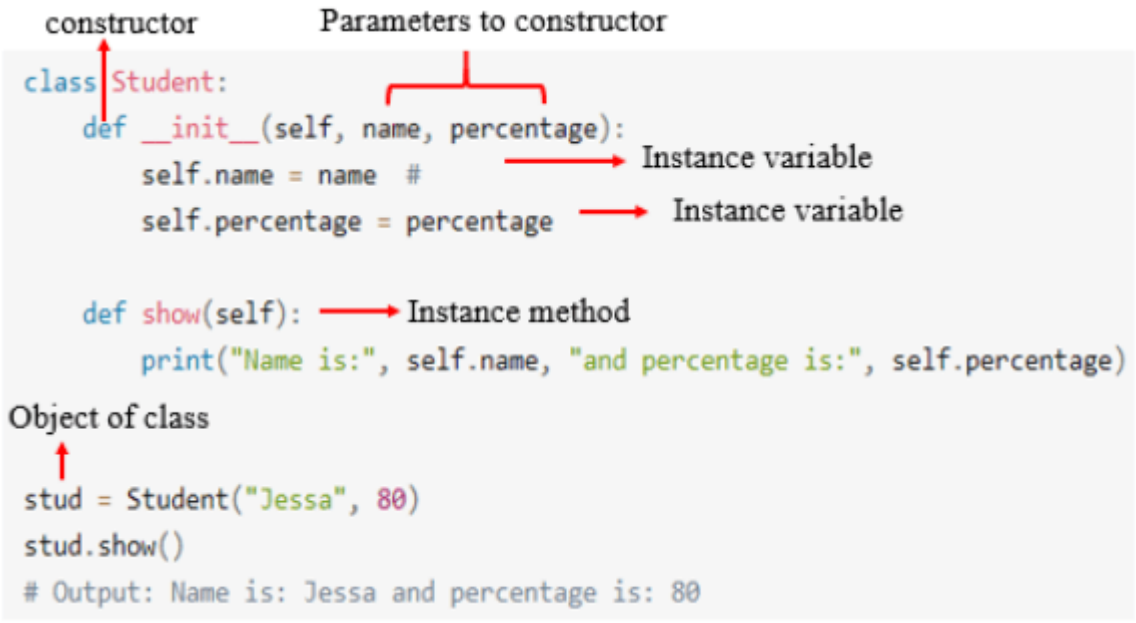
```
class Employee:
    # class variables
    company_name = Jewel Software'

    # constructor to initialize the object
    def __init__(self, name, salary):
        # instance variables
        self.name = name
        self.salary = salary

    # instance method
    def show(self):
        print('Employee:', self.name, self.salary, self.company_name)

# create first object
emp1 = Employee("Ravi Raju", 60000)
```

```
emp1.show()

# create second object
emp2 = Employee(" Joseph", 90000)
emp2.show()
```

**Output**:

```
Employee: Ravi Raju 60000 Jewel Software

Employee: Joseph 90000 Jewel Software
```

- In the above example, we created a Class with the name Employee.
- Next, we defined two attributes name and salary.
- Next, in the __init__() method, we initialized the value of attributes. This method is called as soon as the object is created. The init method initializes the object.
- Finally, from the Employee class, we created two objects, Ravi Raju and Joseph.
- Using the object, we can access and modify its attributes.




**instance variables and methods**

### *Constructors in Python*

In Python, a **constructor** is a special type of method used to initialize the object of a Class. The constructor will be executed automatically when the object is created. If we create three objects, the constructor is called three times and initialize each object.

The main purpose of the constructor is to declare and initialize instance variables. It can take at least one argument that is self. The __init()__ method is called the constructor in Python. In other words, the name of the constructor should be __init__(self).

A constructor is optional, and if we do not provide any constructor, then Python provides the default constructor. Every class in Python has a constructor, but it's not required to define it.

### 1.11.2 Class Attributes and Methods

When we design a class, we use instance variables and class variables.

In Class, attributes can be defined into two parts:

- **Instance variables**: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor ( the __init__() method of a class).
- **Class Variables**: A class variable is a variable that is declared inside of class, but outside of any instance method or __init()__ method.

Inside a Class, we can define the following three types of methods.

- **Instance method**: Used to access or modify the object attributes. If we use instance variables inside a method, such methods are called instance methods.
- **Class method**: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.
- **Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't have access to the class attributes.

### 1.11.3 Self Argument:

Whenever we create a class in Python, the programmer needs a way to access its *attributes* and *methods*. In most languages, there is a fixed syntax assigned to refer to attributes and methods; for example, C++ uses this for reference.

In Python, the word self is the first parameter of methods that represents the instance of the class. Therefore, in order to call attributes and methods of a class, the programmer needs to use self.

It is not mandatory to name the first parameter as a self. We can give any name whatever we like, but it has to be the first parameter of an instance method.

**Example**

```
class Student:
    # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # self points to the current object
    def show(self):
        # access instance variable using self
        print(self.name, self.age)

# creating first object
emma = Student('Ishvika', 02)
emma.show()

# creating Second object
kelly = Student('Ammulu', 15)
kelly.show()
```

**Output**

```
Ishvika 02

Ammulu 15
```

**Example : 2**

```
class food():

# init method or constructor

def __init__(self, fruit, color):
self.fruit = fruit
self.color = color

def show(self):
print("fruit is", self.fruit)
print("color is", self.color )

apple = food("apple", "red")
grapes = food("grapes", "green")

apple.show()
grapes.show()
```

**Output:**
Fruit is apple
color is red
Fruit is grapes
color is green

**Python Class self Constructor**

self is also used to refer to a variable field within the class. Let's take an example and see how it works:

```
class Person:


# name made in constructor
def __init__(self, John):
self.name = John

def get_person_name(self):
return self.name
```

In the above example, self refers to the name variable of the entire Person class. Here, if we have a variable within a method, self will not work. That variable is simply existent only while that method is running and hence, is local to that method. For defining global fields or the variables of the complete class, we need to define them outside the class methods.

### 1.11.4 The __init__Method():

In object-oriented programming, **A constructor is a special method used to create and initialize an object of a** class. This method is defined in the class.

- The constructor is executed automatically at the time of object creation.
- The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.

For example, when we execute obj = Sample(), Python gets to know that obj is an object of class Sample and calls the constructor of that class to create an object.

**Note**: In Python, internally, the __new__ is the method that creates the object, and __del__ method is called to destroy the object when the reference count for that object becomes zero.

In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**

1. Internally, the __new__ is the method that creates the object
2. And, using the __init__() method we can implement constructor to initialize the object.

**Syntax** of a constructor:

```
def __init__(self):
    # body of the constructor
```

Where,

- def: The keyword is used to define function.
- __init__() Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated.
- self: The first argument self refers to the current object. It binds the instance to the __init__() method. It's usually named self to follow the naming convention.

**Note**: The __init__() method arguments are optional. We can define a constructor with any number of arguments.

**Example: Create a Constructor in Python**

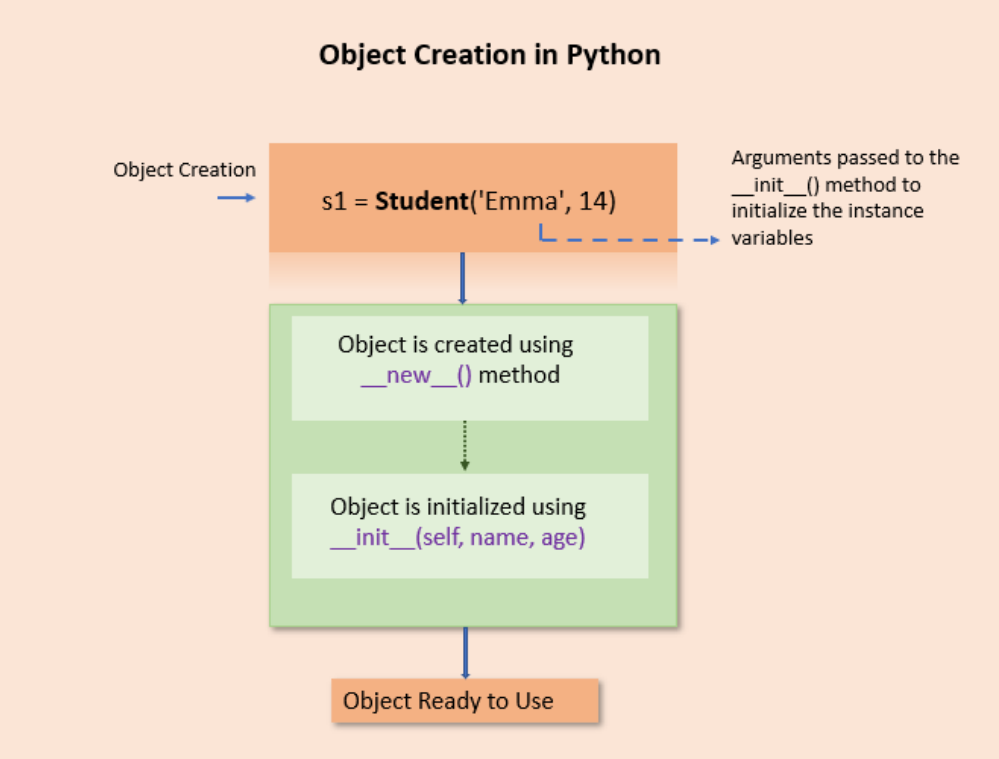
In this example, we'll create a Class **Student** with an instance variable student name. we'll see how to use a constructor to initialize the student name at the time of object creation.

```
class Student:

    # constructor
    # initialize instance variable
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')

    # instance Method
    def show(self):
        print('Hello, my name is', self.name)


# create object using constructor
s1 = Student('Ravi')
s1.show()
```

**Output:**

```
Inside Constructor
All variables initialized

Hello, my name is Ravi
```

- In the above example, an object s1 is created using the constructor
- While creating a Student object name is passed as an argument to the __init__() method to initialize the object.
- Similarly, various objects of the Student class can be created by passing different names as arguments.




**Create an object in Python using a constructor**

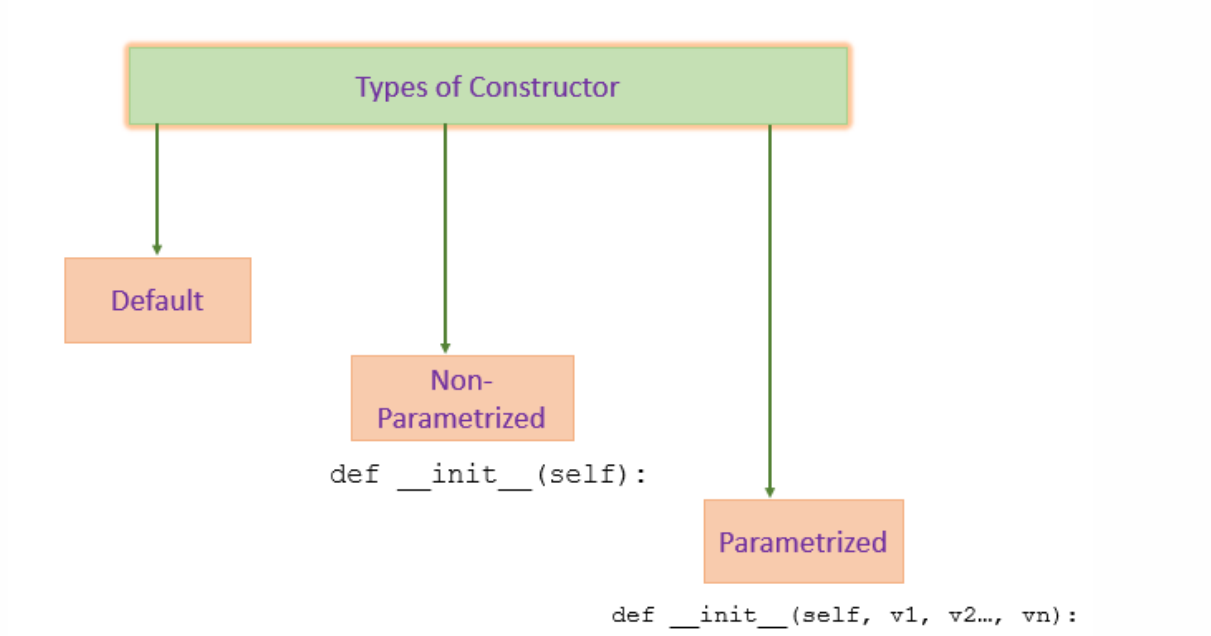
**Note**:

- For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times.
- In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional.
- Python will provide a default constructor if no constructor is defined.

### Types of Constructors:

In Python, we have the following three types of constructors.

- Default Constructor
- Non-parametrized constructor
- Parameterized constructor




### 1.11.4.1 Default Constructor:

Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body.

If you do not implement any constructor in your class or forget to declare it, the Python inserts a default constructor into your code on your behalf. This constructor is known as the default constructor.

It does not perform any task but initializes the objects. It is an empty constructor without a body.

**Note**:

- The default constructor is not present in the source py file. It is inserted into the code during compilation if not exists. See the below image.
- If you implement your constructor, then the default constructor will not be added.

**Example**:

```
class Employee:

    def display(self):
        print('Inside Display')

emp = Employee()
```

```
emp.display()
```

**Output:**

```
Inside Display
```

As you can see in the example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation.

**1.11.4.2 Non-Parametrized Constructor:**

A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.

This constructor doesn't accept the arguments during object creation. Instead, it initializes every object with the same set of values.

```
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "Jewel Software"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

**Output**

```
Name: Jewel Software Address: ABC Street
```

As you can see in the example, we do not send any argument to a constructor while creating an object.

**1.11. 4.3. Parameterized Constructor**

A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.

The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments.

For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor.

**Example**:

```
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    # display object
    def show(self):
        print(self.name, self.age, self.salary)

# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()

kelly = Employee('Kelly', 25, 8500)
kelly.show()
```

**Output**

```
Emma 23 7500

Kelly 25 8500
```

In the above example, we define a parameterized constructor which takes three parameters.

### 1.12. The_del_Method()/ Destructors:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically. The **del**() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

**Syntax of destructor declaration**

```
def __del__(self):

  # body of destructor
```

**Example :**

```
 Python program to illustrate destructor

class Employee:

    # Initializing

    def __init__(self):

        print('Employee created.')

    # Deleting (Calling destructor)

    def __del__(self):

        print('Destructor called, Employee deleted.')

obj = Employee()

del obj
```

**output :**

```
Employee created.
Destructor called, Employee deleted.
```

## 3.5 Access specifiers or Access modifiers in Python Programming

**Access Modifiers:** Access specifiers or access modifiers in python programming are used to limit the access of class variables and class methods outside of class while implementing the concepts of inheritance. This can be achieved by: Public, Private and Protected keyword.

**There are three types of access specifiers or access modifiers**

- **1).** Public access modifier
- **2).** Private access modifier
- **3).** Protected access modifier

We can easily inherit the properties or behaviour of any class using the concept of inheritance. But some classes also holds the data (class variables and class methods) that we don't want other classes to inherit. So, to prevent that data we used access specifiers in python.

**Note:** Access modifiers in python are very helpful when we are using the concepts of inheritance. We can also apply the concept of access modifiers to class methods.

| Access Specifiers | Same Class | Same Package | Derived Class | Other Class |
|---|---|---|---|---|
| Public | Yes | Yes | Yes | Yes |
| Protected | Yes | Yes | Yes | No |
| Private | Yes | No | No | No |

we will see the syntax and then types of access specifiers (public, private and protected) in Python. Later a detailed example of each type of access specifiers.

**Syntax example** :

```
#defining class Student
class Student:
    #constructor is defined
    def __init__(self, name, age, salary):
        self.age = age              # public Attribute
        self._name = name           # protected Attribute
        self.__salary = salary      # private Attribute

    def _funName(self):             # protected method
        pass

    def __funName(self):            # private method
        pass

# object creation
```

```
obj = Student("pythonlobby",21,45000)
```

### 3.5.1 Public Access Modifier in Python

All the variables and methods (member functions) in python are by default public. Any instance variable in a class followed by the **'self'** keyword ie**. self.var_name** are public accessed.

**syntax:**

```
# Syntax_public_access_modifiers

# defining class Student
class Student:
    # constructor is defined
    def __init__(self, age, name):
        self.age = age            # public Attribute
        self.name = name          # public Attribute

# object creation
obj = Student(35,"Ravi Raju B")
print(obj.age)
print(obj.name)
```

Output:

```
35

Ravi Raju B
```

### 3.5.2 Private Access Modifier

Private members of a class (variables or methods) are those members which are only accessible inside the class. We cannot use private members outside of class.

It is also not possible to inherit the private members of any class (parent class) to derived class (child class). Any instance variable in a class followed by self keyword and the variable name starting with double underscore ie. **self.__varName** are the private accessed member of a class.

**Syntax:**

```
# Private_access_modifiers
class Student:
    def __init__(self, age, name):
        self.__age = age

        def __funName(self):
            self.y = 34
            print(self.y)

class Subject(Student):
    pass

obj = Student(21,"pythonlobby")
obj1 = Subject

# calling by object reference of class Student
```

```
print(obj.__age)
print(obj.__funName())

# calling by object reference of class Subject
print(obj1.__age)
print(obj1.__funName())
```

Output:

**Example 2:**

```
# Example_of_using_private_access_modifiers
class Student:
    def __init__(self):
        self.name = "Jessy Joy"  # Public
        self.__age = 39          # Private

class Subject(Student):
    pass

# object creation
obj = Student()
obj1 = Subject()

# calling using object ref. of Student class
print(obj.name)  # No Error
print(obj1.name) # No Error

# calling using object ref. of Subject class
print(obj.__age)  # Error
print(obj1.__age)  # Error
```

Output:

```
Jessy Joy

Jessy Joy

Error : AttributeError: 'Student' object has no attribute 'age'

Error : AttributeError: 'Subject' object has no attribute 'age'
```

---

### 3.5.3. Protected Access Modifier

Protected variables or we can say protected members of a class are restricted to be used only by the member functions and class members of the same class. And also it can be accessed or inherited by its derived class ( child class ). We can modify the values of protected variables of a class. The syntax we follow to make any variable **protected** is to write variable name followed by a single underscore (_) ie. **_varName.**

- **Note:** We can access protected members of class outside of class even we can modify its value also. Now the doubt that arises is, public access modifiers follow the same except its syntax. Actually, protected access modifiers are designed so that responsible programmer would identify by their name convention and do the required operation only on that protected class members or class methods.

**Syntax and Example 3:**

```
#Syntax_protected_access_modifiers
class Student:
    def __init__(self):
        self._name = "Information Technology"

    def _funName(self):
        return "Method Here"

class Subject(Student):
    pass

obj = Student()
obj1 = Subject()

# calling by obj. ref. of Student class
print(obj._name)     # Information Technology
print(obj._funName())     # Method Here
# calling by obj. ref. of Subject class
print(obj1._name)     # Information Technology
print(obj1._funName())     # Method Here
```

```
Information Technology
Method Here


Information Technology
Method Here
```

## 3.6 Accessing Class Attributes and Methods in Python:

Classes are the blueprint from which the objects are created. Each class in python can have many attributes including a function as an attribute.

Attributes of a class are function objects that define corresponding methods of its instances. They are used to implement access controls of the classes. Attributes of a class can also be accessed using the following built-in methods and functions :

1. **getattr() –** This function is used to access the attribute of object.
2. **hasattr() –** This function is used to check if an attribute exist or not.

3. **setattr() –** This function is used to set an attribute. If the attribute does not exist, then it would be created.
4. **delattr() –** This function is used to delete an attribute. If you are accessing the attribute after deleting it raises error "class has no attribute".

The following methods are explained with the example given below :

**# Python code for accessing attributes of class**

```
class emp:

    name=' Ravi Raju'

    salary='50000'

    def show(self):

        print (self.name)

        print (self.salary)

e1 = emp()

# Use getattr instead of e1.name

print (getattr(e1,'name'))

# returns true if object has attribute

print (hasattr(e1,'name'))

 # sets an attribute

setattr(e1,'height',152)

  # returns the value of attribute name height

print (getattr(e1,'height'))

# delete the attribute
```

delattr(emp,'salary')

Output :

Ravi Raju

True

152

Example: 2

```
class StateInfo:
   StateName='Telangana'
   population='3.5 crore'

   def func1(self):
      print("Hello from my function")

print getattr(StateInfo,'StateName')

# returns true if object has attribute
print hasattr(StateInfo,'population')

setattr(StateInfo,'ForestCover',39)

print getattr(StateInfo,'ForestCover')

print hasattr(StateInfo,'func1')
```

Output
Telangana
True
39
True

## 3.7 Inheritance:

Inheritance is the ability to 'inherit' features or attributes from already written classes into newer classes we make. These features and attributes are defined data structures and the functions we can perform with them, a.k.a. Methods. It promotes code reusability, which is considered one of the best industrial coding practices as it makes the codebase modular.

In python inheritance, new class/es inherits from older class/es. The new class/es copies all the older class's functions and attributes without rewriting the syntax in the new class/es. These new classes are called derived classes, and old ones are called base classes.

For example, inheritance is often used in biology to symbolize the transfer of genetic traits from parents to their children. Similarly, we have parent classes (Base classes) and child classes (derived classes). In Inheritance, we derive classes from other already existing

classes. The existing classes are the parent/base classes from which the attributes and methods are inherited in the child classes.

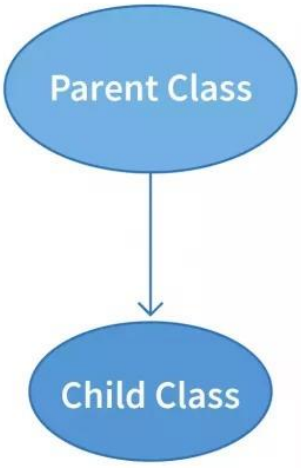
### 3.7.1 Types of Inheritance in Python

Now that we are all set with the prerequisites to understand how inheritance in python is carried out, let's look at various inheritance types.

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchiel Inheritance
5. Hybrid Inheritance

### 3.7.1.2. Single Inheritance in Python:

Single Inheritance is the simplest form of inheritance where a single child class is derived from a single parent class. Due to its candid nature, it is also known as **Simple Inheritance**.





**Example:**

```
# python 3 syntax
# single inheritance example

class parent:               # parent class
    def func1(self):
        print("Hello Parent")

class child(parent):
  # child class
```

```
    def func2(self):                # we include the parent class
        print("Hello Child")   # as an argument in the child
                               # class

# Driver Code
test = child()              # object created
test.func1()                # parent method called via child object
test.func2()                # child method called
```

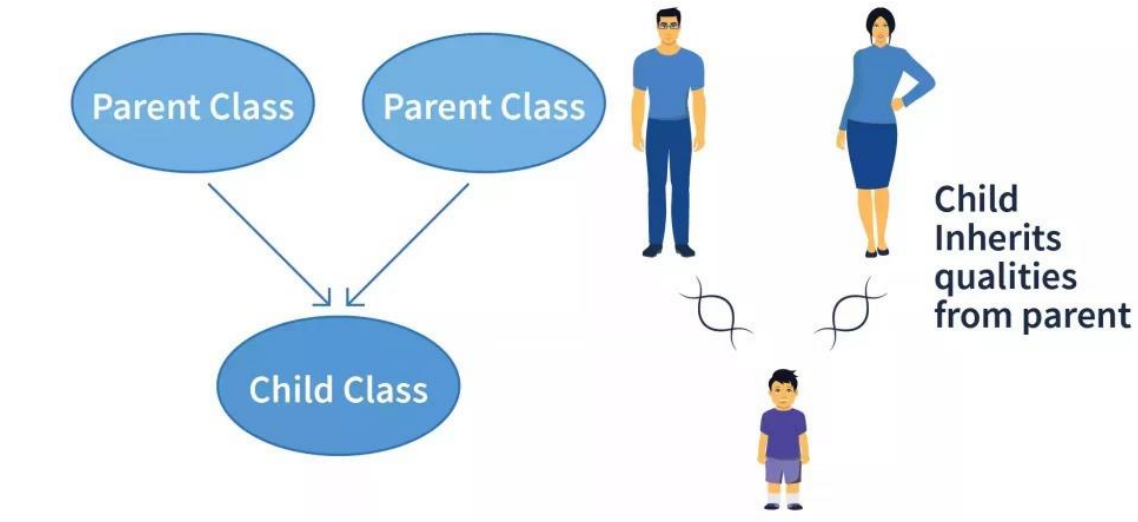
**Output:**

```
> Hello Parent
> Hello Child
```

### 3.7.1.3. Multiple Inheritance in Python:

In multiple inheritance, a single child class is inherited from two or more parent classes. It means the child class has access to all the parent classes' methods and attributes.

However, if two parents have the same “named” methods, the child class performs the method of the first parent in order of reference. To better understand which class’s methods shall be executed first, we can use the Method Resolution Order function (mro). It tells the order in which the child class is interpreted to visit the other classes.




**Example:**

Basic implementation of multiple inheritance

```
# python 3 syntax
# multiple inheritance example
```

```
class parent1:                  # first parent class
    def func1(self):
        print("Hello Parent1")

class parent2:                  # second parent class
    def func2(self):
        print("Hello Parent2")

class parent3:                  # third parent class
    def func2(self):                # the function name is same as parent2
        print("Hello Parent3")

class child(parent1, parent2, parent3):     # child class
    def func3(self):                # we include the parent classes
        print("Hello Child")        # as an argument comma separated

# Driver Code
test = child()          # object created
test.func1()            # parent1 method called via child
test.func2()            # parent2 method called via child instead of parent3
test.func3()            # child method called

# to find the order of classes visited by the child class, we use __mro__ on the child class
print(child.__mro__)
```

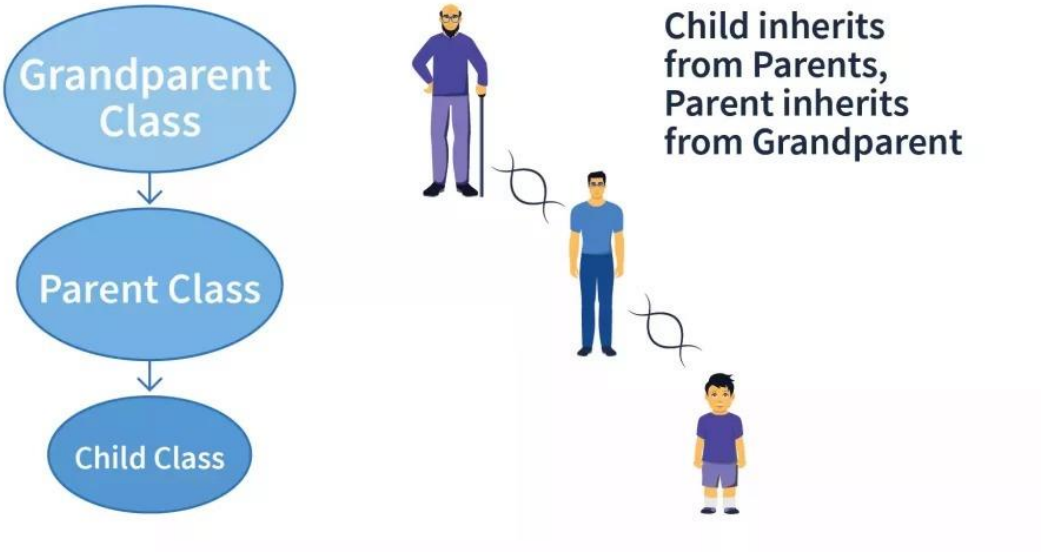
**Output:**

```
> Hello Parent1
> Hello Parent2
> Hello Child
>(<class '__main__.child'>, <class '__main__.parent1'>, <class '__main__.parent2'>, <class
'__main__.parent3'>, <class 'object'>)
```

As we can see with the help of **mro**, the child class first visits itself, then the first parent class, referenced before the second parent class. Similarly, it visits the second parent class before the third parent class, and that's why it performs the second parent's function rather than the third parent's. Finally, it visits any objects that may have been created.

### 3.7.1.4 Multilevel Inheritance in Python:

In multilevel inheritance, we go beyond just a parent-child relation. We introduce grandchildren, great-grandchildren, grandparents, etc. We have seen only two levels of inheritance with a superior parent class/es and a derived class/es, but here we can have multiple levels where the parent class/es itself is derived from another class/es.




Example:

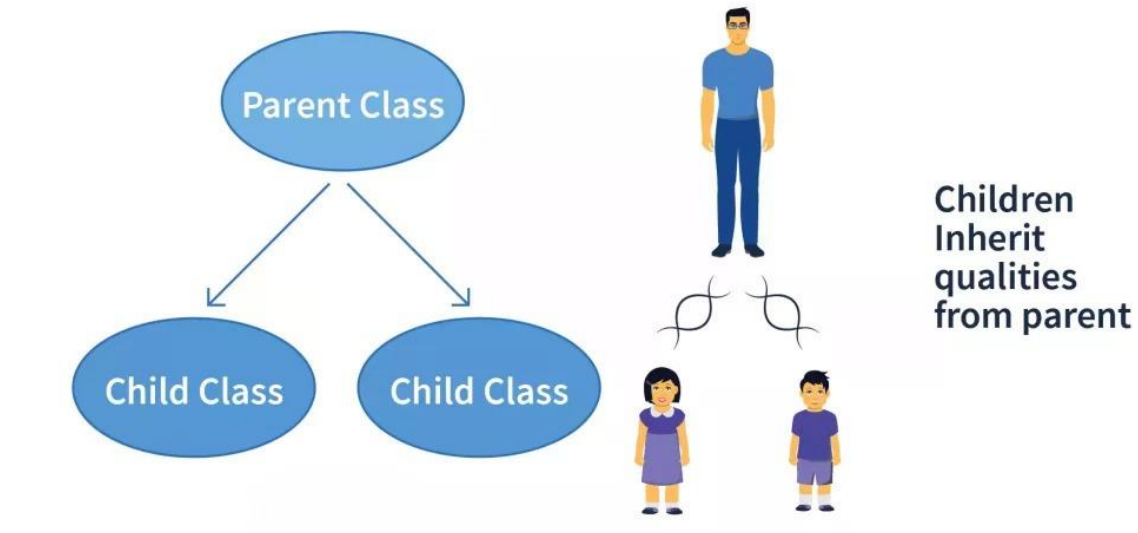
```
class grandparent:              # first level
    def func1(self):
        print("Hello Grandparent")

class parent(grandparent):      # second level
    def func2(self):
        print("Hello Parent")

class child(parent):            # third level
    def func3(self):
        print("Hello Child")


# Driver Code
test = child()              # object created
test.func1()                # 3rd level calls 1st level
test.func2()                # 3rd level calls 2nd level
test.func3()                # 3rd level calls 3rd level
```

**Output:**

```
> Hello Grandparent
> Hello Parent
> Hello Child
```

### 3.7.1.5 Hierarchical Inheritance in Python:

Hierarchical Inheritance is the right opposite of multiple inheritance. It means that, there are multiple derived child classes from a single parent class.




**Example:**

```
# python 3 syntax
# hierarchical inheritance example

class parent:                   # parent class
    def func1(self):
        print("Hello Parent")

class child1(parent):               # first child class
    def func2(self):
        print("Hello Child1")


class child2(parent):               # second child class
    def func3(self):
        print("Hello Child2")


# Driver Code
test1 = child1()                  # objects created
test2 = child2()

test1.func1()                   # child1 calling parent method
test1.func2()                   # child1 calling its own method

test2.func1()                   # child2 calling parent method
test2.func3()                   # child2 calling its own method
```

**Output:**

```
> Hello Parent
```

```
> Hello Child1
> Hello Parent
> Hello Child2
```

### 3.7.1.6 Hybrid Inheritance in Python:

Hybrid Inheritance is the mixture of two or more different types of inheritance. Here we can have many relationships between parent and child classes with multiple levels.


**Example:**

```
# python 3 syntax
# hybrid inheritance example

class parent1:                  # first parent class
    def func1(self):
        print("Hello Parent")


class parent2:                  # second parent class
    def func2(self):
        print("Hello Parent")

class child1(parent1):              # first child class
    def func3(self):
        print("Hello Child1")
```

```
class child2(child1, parent2):          # second child class
    def func4(self):
        print("Hello Child2")


# Driver Code
test1 = child1()                    # object created
test2 = child2()

test1.func1()                   # child1 calling parent1 method
test1.func3()                   # child1 calling its own method

test2.func1()                   # child2 calling parent1 method
test2.func2()                   # child2 calling parent2 method
test2.func3()                   # child2 calling child1 method
test2.func4()                   # child2 calling its own method
```

**Output:**

```
> Hello Parent1
> Hello Child1
> Hello Parent1
> Hello Parent2
> Hello Child1
> Hello Child2
```

This example shows a combination of three types of python inheritance.

Parent1 -> Child1 : Single Inheritance

Parent1 -> Child1 -> Child2 : Multi – Level Inheritance

Parent1 -> Child2 <- Parent2 : Multiple Inheritance

### 3.8 Special Functions in Python Inheritance:

Python is a very versatile and user-friendly language. It provides some amazing in-built functions that make our lives easier when understanding inheritance, especially of a complex nature.

**super() function**

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method already provided by one of its super-classes or parent classes. This discrepancy is caused due to similar naming convention of the methods. Commonly we can see this situation when the parent's **init()** is overridden by the child's **init()**, and hence the child class cannot inherit attributes from the parent class.

Similarly, we can face this same problem with methods other than **init** but having the same naming convention across parent and child classes.

One solution is to call the parent method inside the child method.

```
# python 3 syntax
# solution to method overriding - 1

class parent:                    # parent class

    def __init__(self):          # __init__() of parent
        self.attr1 = 50
        self.attr2 = 66

class child(parent):             # child class

    def __init__(self):          # __init__() of child
        parent.__init__(self)      # calling parent's __init__()
        self.attr3 = 45

test = child()                   # object initiated

print (test.attr1)               # parent attribute called via child
```

**Output:**

```
> 50
```

Another way to solve this problem without explicitly typing out the parent name is to use super(). It automatically references the parent/base class from which the child class is derived. It is extremely helpful to call overridden methods in classes with many methods.

**Example:**

```
# python 3 syntax
# solution to method overriding - 2

class parent:                    # parent class

    def display(self):           # display() of parent
        print("Hello Parent")

class child(parent):             # child class

    def display(self):           # display() of child
        super().display()        # referencing parent via super()
        print("Hello Child")

test = child()                   # object initiated
```

```
test.display()                # display of both activated
```

**Output:**

```
> Hello Parent
> Hello Child
```

Alternately we can also call super() with the following syntax:

```
super(child,self).display()
```

Here, the first parameter references the child class/subclass in the function.

**issubclass()** : The issubclass() function is a convenient way to check whether a class is the child of the parent class. In other words, it checks if the first class is derived from the second class. If the classes share a parent-child relationship, it returns a boolean value of True. Otherwise, False.

**isinstance()** isinstance() is another inbuilt function of python which allows us to check whether an object is an instance of a particular class or any of the classes it has been derived from. It takes two parameters, i.e. the object and the class we need to check it against. It returns a boolean value of True if the object is an instance and otherwise, False.

**Example:**

```
# python 3 syntax
# issubclass() and isinstance() example

class parent:                 # parent class
    def func1():
        print("Hello Parent")

class child(parent):            # child class
    def func2():
        print("Hello Child")


# Driver Code

print(issubclass(child,parent))        # checks if child is subclass of parent

print(issubclass(parent,child))        # checks if parent is subclass of child

A = child()                  # objects initialized
B = parent()

print(isinstance(A,child))              # checks if A is instance of child
print(isinstance(A,parent))             # checks if A is instance of parent
print(isinstance(B,child))              # checks if B is instance of child
print(isinstance(B,parent))            # checks if B is instance of parent
```

**Output:**

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

### 3.9 Advantages of Inheritance in Python:

- **Modular Codebase:** Increases modularity, i.e. breaking down codebase into modules, making it easier to understand. Here, each class we define becomes a separate module that can be inherited separately by one or many classes.
- **Code Reusability**: the child class copies all the attributes and methods of the parent class into its class and use. It saves time and coding effort by not rewriting them, thus following modularity paradigms.
- **Less Development and Maintenance Costs:** changes need to be made in the base class, all derived classes will automatically follow.

### 3.10 Disadvantages of Inheritance in Python:

- **Decreases the Execution Speed**: loading multiple classes because they are interdependent
- **Tightly Coupled Classes**: this means that even though parent classes can be executed independently, child classes cannot be executed without defining their parent classes.

### 3.11 Static methods :

A static method is a method[member function] that don't use argument self at all. To declare a static method, proceed it with the statement "@staticmethod".

```
# Python code for accessing methods using static method
class test:
    @staticmethod
    def square(x):
        test.result = x*x

# object 1 for class
t1=test()

# object 2 for class
t2 = test()
t1.square(2)

# printing result for square(2)
print (t1.result)
t2.square(3)

# printing result for square(3)
```

```
print (t2.result)

# printing the last value of result as we declared the method static
print (t1.result)
```

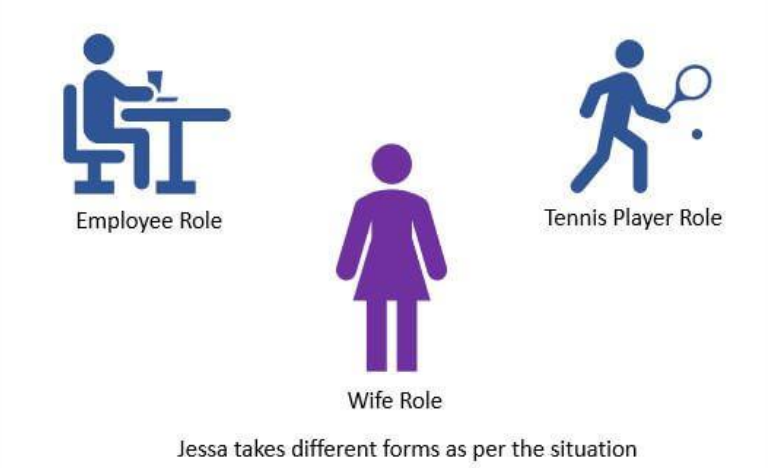
Output :

```
4

9

9
```

# 4. Introduction:

Polymorphism in Python is the ability of an object to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

For example, Jessa acts as an employee when she is at the office. However, when she is at home, she acts like a wife. Also, she represents herself differently in different places. Therefore, the same person takes different forms as per the situation.




A person takes different forms. In polymorphism, a method can **process objects differently depending on the class type or data type**. Let's see simple examples to understand it better.

### Polymorphism in Built-in function len()

The built-in function len() calculates the length of an object depending upon its type. If an object is a string, it returns the count of characters, and If an object is a list, it returns the count of items in a list.

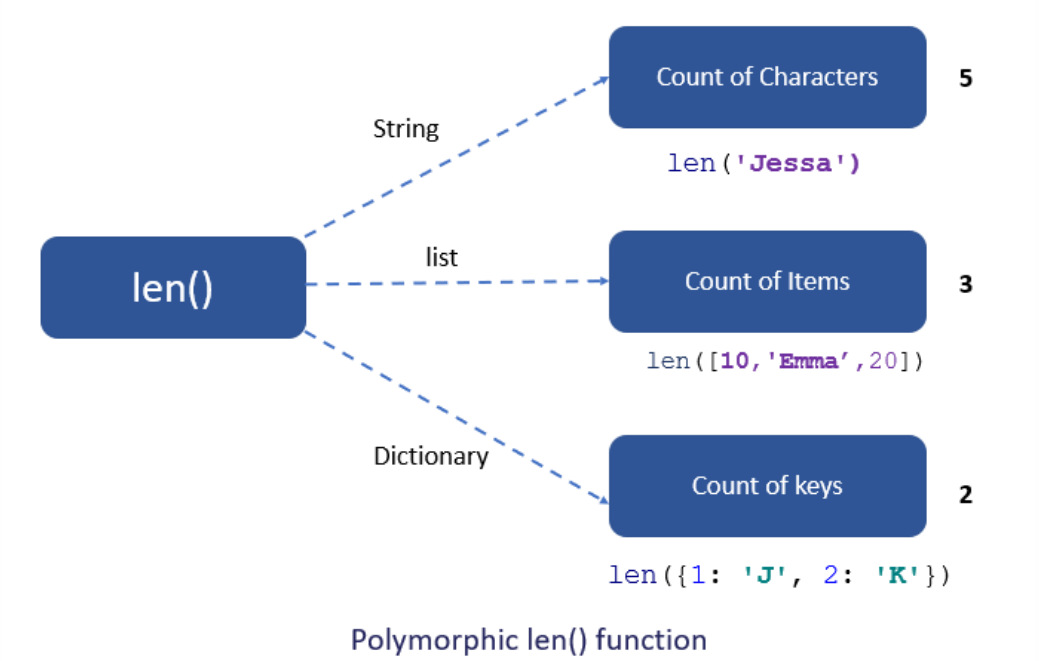
The len() method treats an object as per its class type.

**Example**:

```
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'

# calculate count
print(len(students))
print(len(school))
```

**Output**

```
3
10
```




Polymorphic len() function

## 4.1 Operator Overloading in Python:

Operator overloading means changing the default behavior of an operator depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes. For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings. The operator + is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

**Example**:

```
# add 2 numbers
print(100 + 200)

# concatenate two strings
print('Jess' + 'Roy')

# merger two list
print([10, 20, 30] + ['jessa', 'emma', 'kelly'])
```

**Output**:

```
300
JessRoy
[10, 20, 30, 'jessa', 'emma', 'kelly']
```

**Overloading + operator for custom objects**

Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn't add two objects. See the following example for more details.

**Example**:

```
class Book:
    def __init__(self, pages):
        self.pages = pages

# creating two objects
b1 = Book(400)
b2 = Book(300)

# add two objects
print(b1 + b2)
```

**Output**

```
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

We can overload + operator to work with custom objects also. Python provides some special or magic function that is automatically invoked when associated with that particular operator.

For example, when we use the + operator, the magic method __add__() is automatically invoked. Internally + operator is implemented by using __add__() method. We have to override this method in our class if you want to add two custom objects.

**Example**:

```
class Book:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages

b1 = Book(400)
b2 = Book(300)
print("Total number of pages: ", b1 + b2)
```

**Output**

```
Total number of pages: 700
```

Example Code:

```
# Python Program to perform addition
# of two complex numbers using binary
# + operator overloading.

class complex:
        def __init__(self, a, b):
                self.a = a
                self.b = b
        # adding two objects
        def __add__(self, other):
                return self.a + other.a, self.b + other.b
        def __str__(self):
                return self.a, self.b
Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

**Output :**

(3, 5)

## Overloading the * Operator

The * operator is used to perform the multiplication. Let's see how to overload it to calculate the salary of an employee for a specific period. Internally * operator is implemented by using the __mul__() method.

**Example**:

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

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days


class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days


emp = Employee("Jessa", 800)
timesheet = TimeSheet("Jessa", 50)
print("salary is: ", emp * timesheet)
```

**Output**

```
Wroked for 50 days
salary is:  40000
```

### Magic Methods

In Python, there are different magic methods available to perform overloading operations. The below table shows the magic methods names to overload the mathematical operator, assignment operator, and relational operators in Python.

| Operator Name | Symbol | Magic method |
|---|---|---|
| Addition | + | __add__(self, other) |
| Subtraction | - | __sub__(self, other) |
| Multiplication | * | __mul__(self, other) |
| Division | / | __div__(self, other) |
| Floor Division | // | __floordiv__(self,other) |
| Modulus | % | __mod__(self, other) |
| Power | ** | __pow__(self, other) |
| Increment | += | __iadd__(self, other) |
| Decrement | -= | __isub__(self, other) |
| Product | *= | __imul__(self, other) |
| Division | /+ | __idiv__(self, other) |
| Modulus | %= | __imod__(self, other) |
| Power | **= | __ipow__(self, other) |
| Less than | < | __lt__(self, other) |
| Greater than | > | __gt__(self, other) |
| Less than or equal to | <= | __le__(self, other) |
| Greater than or equal to | >= | __ge__(self, other) |

| Operator Name | Symbol | Magic method |
|---|---|---|
| Equal to | == | __eq__(self, other) |
| Not equal | != | __ne__(self, other) |

## 4.2 Polymorphism With Inheritance:

Polymorphism is mainly used with inheritance. In inheritance, child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

Using method overriding polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class. This **process of re-implementing the inherited method in the child class** is known as Method Overriding.

**Advantage of method overriding**

- It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
- Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code

# Key features of Method Overriding in Python

These are some of the key features and advantages of method overriding in Python --

- Method Overriding is derived from the concept of object oriented programming
- Method Overriding allows us to change the implementation of a function in the child class which is defined in the parent class.
- Method Overriding is a part of the inheritance mechanism
- Method Overriding avoids duplication of code
- Method Overriding also enhances the code adding some additional properties.

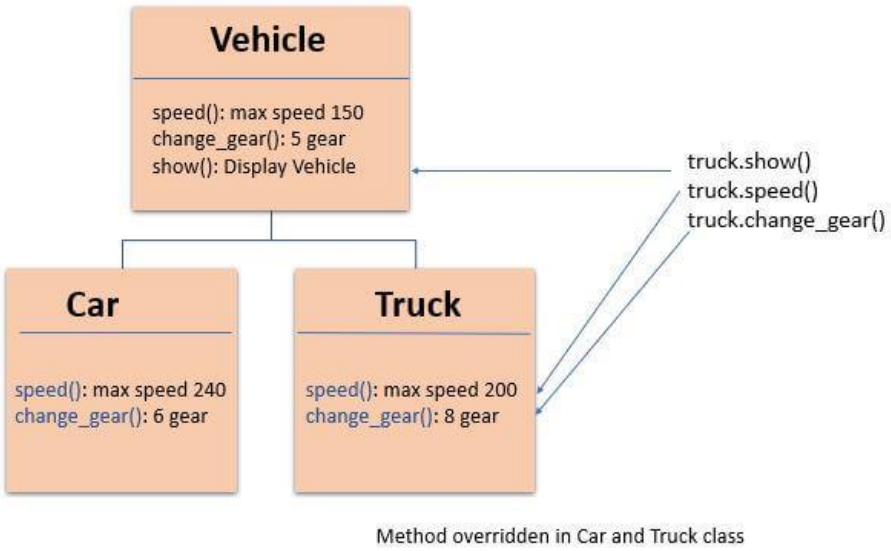
# Prerequisites for method overriding

There are certain prerequisites for method overriding in Python. They're discussed below --

1. Method overriding cannot be done within a class. So,we need to derive a child class from a parent class. Hence **Inheritance** is mandatory.
2. The method must have the **same name** as in the parent class
3. The method must have the **same number of parameters** as in the parent class.

In polymorphism, **Python first checks the object's class type and executes the appropriate method** when we call the method. For example, If you create the Car object, then Python calls the speed() method from a Car class.
Let's see how it works with the help of an example.

**Example: Method Overriding**
In this example, we have a vehicle class as a parent and a 'Car' and 'Truck' as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same instance method name in each class but with a different implementation. Using this code can be extended and easily maintained over time.




Polymorphism with Inheritance

```
class Vehicle:

    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def show(self):
        print('Details:', self.name, self.color, self.price)

    def max_speed(self):
        print('Vehicle max speed is 150')

    def change_gear(self):
        print('Vehicle change 6 gear')

# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')
```

```
    def change_gear(self):
        print('Car change 7 gear')


# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()

# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

**Output**:

```
Details: Car x1 Red 20000
Car max speed is 240
Car change 7 gear

Details: Truck x1 white 75000
Vehicle max speed is 150
Vehicle change 6 gear
```

As you can see, due to polymorphism, the Python interpreter recognizes that the max_speed() and change_gear() methods are overridden for the car object. So, it uses the one defined in the child class (Car)
On the other hand, the show() method isn't overridden in the Car class, so it is used from the Vehicle class.

## Overrride Built-in Functions

In Python, we can change the default behavior of the built-in functions. For example, we can change or extend the built-in functions such as len(), abs(), or divmod() by redefining them in our class. Let's see the example.

**Example**
In this example, we will redefine the function len()

```
class Shopping:
    def __init__(self, basket, buyer):
        self.basket = list(basket)
        self.buyer = buyer
```

```
    def __len__(self):
        print('Redefine length')
        count = len(self.basket)
        # count total items in a different way
        # pair of shoes and shir+pant
        return count * 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')
print(len(shopping))
```

**Output**

```
Redefine length
4
```

## Polymorphism In Class methods

Polymorphism with class methods is useful when we group different objects having the same method. we can add them to a list or a tuple, and we don't need to check the object type before calling their methods. Instead, Python will check object type at runtime and call the correct method. Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

**Example**

In the below example, fuel_type() and max_speed() are the instance methods created in both classes.

```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

ferrari = Ferrari()
bmw = BMW()

# iterate objects of same type
for car in (ferrari, bmw):
    # call methods without checking class of object
```

```
    car.fuel_type()
    car.max_speed()
```

**Output**

```
Petrol
Max speed 350

Diesel
Max speed is 240
```

As you can see, we have created two classes Ferrari and BMW. They have the same instance method names fuel_type() and max_speed(). However, we have not linked both the classes nor have we used inheritance.

We packed two different objects into a tuple and iterate through it using a car variable. It is possible due to polymorphism because we have added the same method in both classes Python first checks the object's class type and executes the method present in its class.

## 4.3. Polymorphism with Function and Objects

We can create polymorphism with a function that can take any object as a parameter and execute its method without checking its class type. Using this, we can call object actions using the same function instead of repeating method calls.

**Example**

```
class Ferrari:
    def fuel_type(self):
        print("Petrol")

    def max_speed(self):
        print("Max speed 350")

class BMW:
    def fuel_type(self):
        print("Diesel")

    def max_speed(self):
        print("Max speed is 240")

# normal function
def car_details(obj):
    obj.fuel_type()
    obj.max_speed()

ferrari = Ferrari()
bmw = BMW()

car_details(ferrari)
```

```
car_details(bmw)
```

**Output**

```
Petrol
Max speed 350
Diesel
Max speed is 240
```

### Polymorphism In Built-in Methods

The word polymorphism is taken from the Greek words poly (many) and morphism (forms). It means a **method can process objects differently depending on the class type or data type.**

The built-in function reversed(obj) returns the iterable by reversing the given object. For example, if you pass a string to it, it will reverse it. But if you pass a list of strings to it, it will return the iterable by reversing the order of elements (it will not reverse the individual string).

Let us see how a built-in method process objects having different data types.

**Example**:

```
students = ['Emma', 'Jessa', 'Kelly']
school = 'ABC School'

print('Reverse string')
for i in reversed('PYnative'):
    print(i, end=' ')

print('\nReverse list')
for i in reversed(['Emma', 'Jessa', 'Kelly']):
    print(i, end=' ')
```

**Output**:

```
Reverse string
e v i t a n Y P

Reverse list
Kelly Jessa Emma
```

### Method Overloading

The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.

**Example**

```
def addition(a, b):
```

```
    c = a + b
    print(c)


def addition(a, b, c):
    d = a + b + c
    print(d)


# the below line shows an error
# addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)
```

To overcome the above problem, we can use different ways to achieve the method overloading. In Python, to overload the class method, we need to write the method's logic so that different code executes inside the function depending on the parameter passes.
For example, the built-in function range() takes three parameters and produce different result depending upon the number of parameters passed to it.
**Example**:

```
for i in range(5): print(i, end=', ')
print()
for i in range(5, 10): print(i, end=', ')
print()
for i in range(2, 12, 2): print(i, end=', ')
```

**Output**:

```
0, 1, 2, 3, 4,
5, 6, 7, 8, 9,
2, 4, 6, 8, 10,
```

Let's assume we have an area() method to calculate the area of a square and rectangle. The method will calculate the area depending upon the number of parameters passed to it.

- If one parameter is passed, then the area of a square is calculated
- If two parameters are passed, then the area of a rectangle is calculated.

**Example: User-defined polymorphic method**

```
class Shape:
    # function with two default parameters
    def area(self, a, b=0):
        if b > 0:
            print('Area of Rectangle is:', a * b)
        else:
```

```
        print('Area of Square is:', a ** 2)

square = Shape()
square.area(5)

rectangle = Shape()
rectangle.area(5, 3)
```

**Output**:

```
Area of Square is: 25
Area of Rectangle is: 15
```

## 4..4 EXCEPTION HANDLING:

**Introduction:**

**Errors** are nothing but mistakes in the code which are potentially harmful.

**Errors and exceptions:**

Errors or mistakes in a program are often referred to as bugs. They are almost always the fault of the programmer. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

1. Syntax errors / Compile time errors
2. Runtime errors
3. Logical errors

### 4.4.1 Syntax errors:

Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything. Syntax errors are mistakes in the use of the Python language, and are analogous to spelling or grammar mistakes in a language like English.
Common Python syntax errors include:

1. leaving out a keyword
2. putting a keyword in the wrong place
3. leaving out a symbol, such as a colon, comma or brackets
4. misspelling a keyword
5. incorrect indentation
6. empty block

Python will do its best to tell you where the error is located, but sometimes its messages can be misleading: for example, if you forget to escape a quotation mark inside a string you may get a syntax error referring to a place later in your code, even though that is not the real source of the problem.

Here are some examples of syntax errors in Python:
myfunction(x, y):

```
    return x + y

else:
    print("Hello!")

if mark >= 50
    print("You passed!")

if arriving:
    print("Hi!")
esle:
    print("Bye!")

if flag:
print("Flag is set!")
```

### 4.4.2 Runtime errors:

If a program is syntactically correct – that is, free of syntax errors – it will be run by the Python interpreter. However, the program may exit unexpectedly during execution if it encounters a runtime error – a problem which was not detected when the program was parsed, but is only revealed when a particular line is executed. When a program comes to a halt because of a runtime error, we say that it has crashed.

Some examples of Python runtime errors:

1. division by zero
2. performing an operation on incompatible types
3. using an identifier which has not been defined
4. accessing a list element, dictionary value or object attribute which doesn't exist
5. trying to access a file which doesn't exist

Runtime errors often creep in if you don't consider all possible values that a variable could contain, especially when you are processing user input. You should always try to add checks to your code to make sure that it can deal with bad input and edge cases gracefully.

### 4.4.3 Logical errors:

Logical errors are the most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred. You will have to find the problem on your own by reviewing all the relevant parts of your code – although some tools can flag suspicious code which looks like it could cause unexpected behaviour.

Sometimes these kinds of errors are caused by programmer carelessness. Here are some examples of mistakes which lead to logical errors:

1. using the wrong variable name
2. indenting a block to the wrong level

3. using integer division instead of floating-point division
4. getting operator precedence wrong
5. making a mistake in a boolean expression
6. off-by-one, and other numerical errors

## 4.5 Excepptions:

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.
An Exception is an error that happens during the execution of a program. Whenever there is an error, Python generates an exception that could be handled. It basically prevents the program from getting crashed.

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error, you can use exception handling technique.

A list of common exceptions that can be thrown from a normal python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

**Python Built-in Exceptions:**

| Exception | Cause of Error |
|---|---|
| AssertionError | Raised when assert statement fails. |
| AttributeError | Raised when attribute assignment or reference fails. |
| EOFError | Raised when the input() functions hits end-of-file condition. |

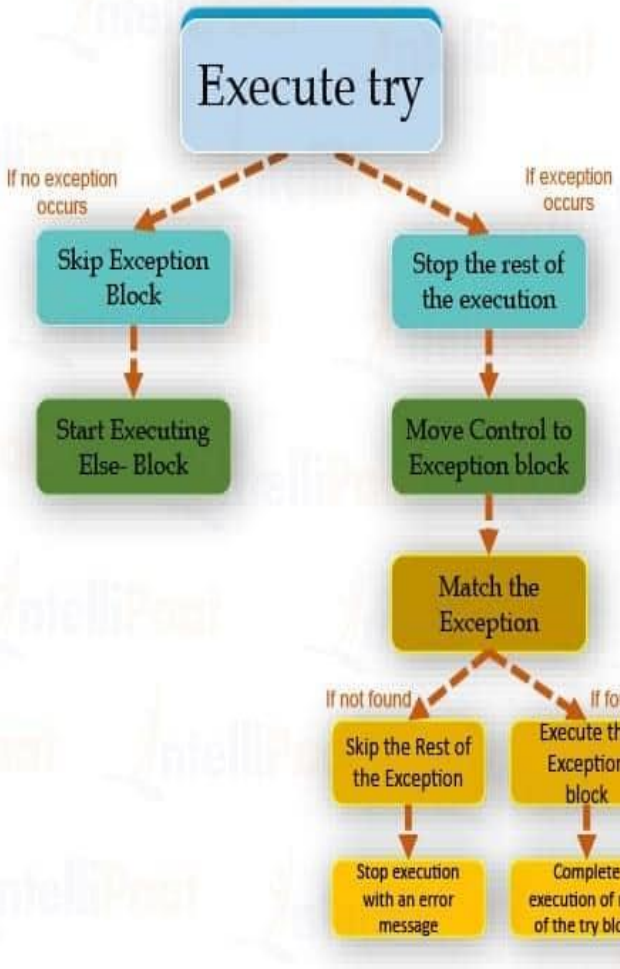
| | |
|---|---|
| FloatingPointError | Raised when a floating point operation fails. |
| ZeroDivisionError | Raised when second operand of division or modulo operation is zero. |
| ImportError | Raised when the imported module is not found. |
| IndexError | Raised when index of a sequence is out of range. |
| KeyError | Raised when a key is not found in a dictionary. |
| KeyboardInterrupt | Raised when the user hits interrupt key (Ctrl+c or delete). |
| MemoryError | Raised when an operation runs out of memory. |
| NameError | Raised when a variable is not found in local or global scope. |
| NotImplementedError | Raised by abstract methods. |
| OSError | Raised when system operation causes system related error. |
| OverflowError | Raised when result of an arithmetic operation is too large to be represented. |
| ReferenceError | Raised when a weak reference proxy is used to access a garbage collected referent. |
| RuntimeError | Raised when an error does not fall under any other category. |
| StopIteration | Raised by next() function to indicate that there is no further item to be returned by iterator. |
| SyntaxError | Raised by parser when syntax error is encountered. |
| IndentationError | Raised when there is incorrect indentation. |
| TabError | Raised when indentation consists of inconsistent tabs and spaces. |
| SystemError | Raised when interpreter detects internal error. |
| SystemExit | Raised by sys.exit() function. |
| TypeError | Raised when a function or operation is applied to an object of incorrect type. |
| UnboundLocalError | Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. |

| | |
|---|---|
| UnicodeError | Raised when a Unicode-related encoding or decoding error occurs. |
| UnicodeEncodeError | Raised when a Unicode-related error occurs during encoding. |
| UnicodeDecodeError | Raised when a Unicode-related error occurs during decoding. |
| UnicodeTranslateError | Raised when a Unicode-related error occurs during translating. |
| ValueError | Raised when a function gets argument of correct type but improper value. |
| GeneratorExit | Raise when a generator's close() method is called. |

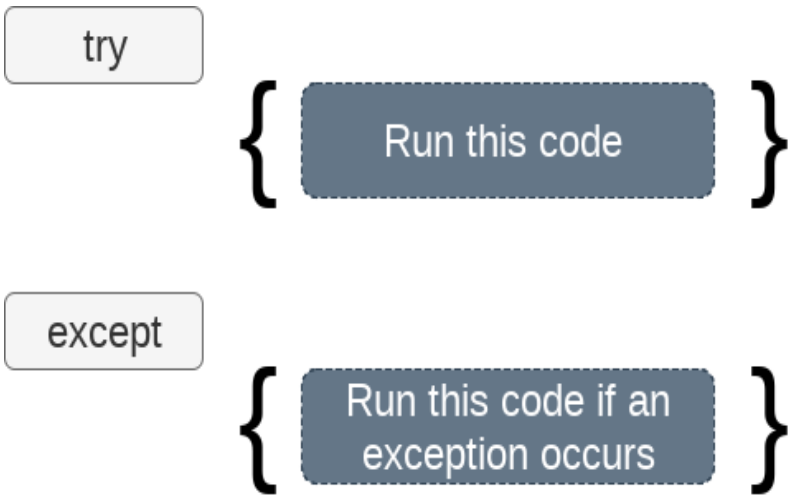
## 4.6 Handling Exceptions:

In Python, exceptions can be handled by two new methods:

- Try: Catches exceptions raised by Python or a program
- Raise: A custom exception which triggers an exception manually



### 4.6.1 try and except block:




Syntax:

```
try:
   #block of code

except Exception1:
   #block of code

except Exception2:
   #block of code

#other code
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

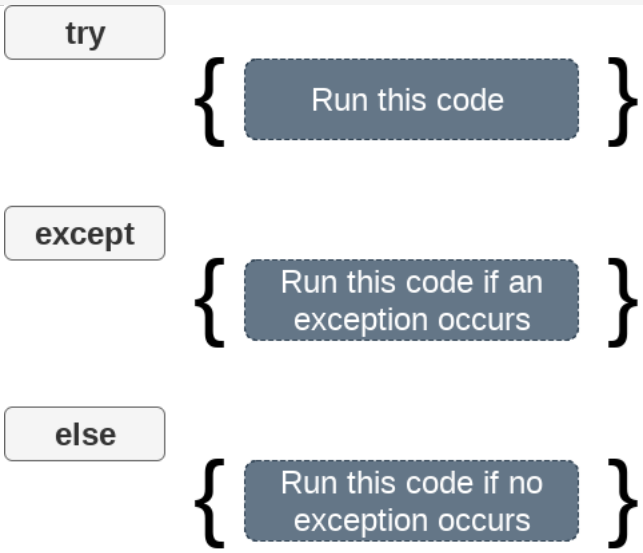
## 4.7 Multiple Except Blocks:

### 4.7.1 try and except block with else:

It gets initiated with a try header line which is followed by a block of indented statements and then by one or more optional except clauses and then at the end an optional else clause can be used as shown below:

The syntax to use the else statement with the try-except statement is given below.

```
try:
        Your statements

except Exception_1:
        If there is Exception_1 then execute this block- statement

except Exception_2:
        If there is Exception_2 then execute this block-statement

else:
        if no exception was raised-statement
```




Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except Exception:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

**Output:**

```
Enter a:10
Enter b:2
a/b = 5
Hi I am else block
```

## 4.7.2 Try-finally Clause:

A try statement can have more than once except clause, It can also have optional else and/or finally statement.

```
try:
   <body>
except <ExceptionType1>:
   <handler1>
except <ExceptionTypeN>:
   <handlerN>
except:
   <handlerExcept>
else:
   <process_else>
finally:
   <process_finally>
```

except clause is similar to elif . When exception occurs, it is checked to match the exception type in except clause. If match is found then handler for the matching case is executed. Also note that in last except clause ExceptionType is omitted. If exception does not match any exception type before the last except clause, then the handler for last except clause is executed.

**Note**: Statements under the else clause run only when no exception is raised.

**Note**: Statements in finally block will run every time no matter exception occurs or not.

Example:

```
num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
result = num1 / num2
print("Result is", result)

except ZeroDivisionError:
   print("Division by zero is error !!")

except SyntaxError:
   print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
   print("Wrong input")

else:
   print("No exceptions")

finally:
   print("This will execute no matter what")
```

### 4.7.3 Raising exceptions:

**To raise your exceptions from your own methods you need to use raise keyword like this**

```
raise ExceptionClass("Your argument")
```

Example:

```
def enterage(age):
    if age < 0:
        raise ValueError("Only positive integers are allowed")

    if age % 2 == 0:
        print("age is even")
    else:
        print("age is odd")

try:
    num = int(input("Enter your age: "))
    enterage(num)

except ValueError:
    print("Only positive integers are allowed")
except:
    print("something is wrong")
```

Output:

```
Enter your age: 12
age is even
Enter your age: -12
Only integers are allowed
```

# 5.1 Numpy Package:

NumPy is one of the most widely used open-source Python libraries, focusing on scientific computation. It features built-in mathematical functions for quick computation and supports big matrices and multidimensional data. "Numerical Python" is defined by the term "NumPy." It can be used in linear algebra, as a multi-dimensional container for generic data, and as a random number generator, among other things. Some of the important functions in NumPy are arcsin(), arccos(), tan(), radians(), etc.

NumPy Array is a Python object which defines an N-dimensional array with rows and columns. In Python, NumPy Array is preferred over lists because it takes up less memory and is faster and more convenient to use. NumPy library in Python has functions for working in domain of Fourier transform, linear algebra, and matrices. Python NumPy is an open-source project that can be used freely.

Numpy is considered as one of the most popular machine learning library in Python.
TensorFlow and other libraries uses Numpy internally for performing multiple operations on Tensors. Array interface is the best and the most important feature of Numpy.
Features Of Numpy

- **Interactive:** Numpy is very interactive and easy to use.
- **Mathematics:** Makes complex mathematical implementations very simple.
- **Intuitive:** Makes coding real easy and grasping the concepts is easy.
- **Lot of Interaction:** Widely used, hence a lot of open source contribution.

The NumPy interface can be used to represent images, sound waves, and other binary raw streams as an N-dimensional array of real values for visualization. Numpy knowledge is required for full-stack developers to implement this library for machine learning.

**How to install NumPy Python?**
Installing the NumPy library is a straightforward process. You can use pip to install the library.Go to the command line and type the following:

- pip install numpy

- If you are using Anaconda distribution, then you can use conda to install NumPy. conda install numpy
- Once the installation is complete, you can verify it by importing the NumPy library in the python interpreter. One can use the numpy library by importing it as shown below.
- import numpy
- If the import is successful, then you will see the following output.

>>> import numpy
>>> numpy.__version__ '1.17.2'

With NumPy, you can easily create arrays, which is a data structure that allows you to store multiple values in a single variable.
In particular, NumPy arrays provide an efficient way of storing and manipulating data. NumPy also includes a number of functions that make it easy to perform mathematical operations on arrays. This can be really useful for scientific or engineering applications. And if you're working with data from a Python script, using NumPy can make your life a lot easier.

Example Program on Numpy:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
a= np.array([1, 2, 3,4,5], ndmin = 2)
print(a)
a = np.array([1, 2, 3], dtype = complex)
print(a)
a = np.array([[1,2,3],[4,5,6]])
print(a.shape)
a.shape = (3,2)
print(a)
a = np.arange(24)
print(a)
a= a.reshape(2,4,3)
print(a)
```

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2)
print(a[s])
a = np.arange(10)
b = a[2:7:2]
print (b)
a = np.arange(10)
b = a[5]
print(b)
a = np.arange(10)
print (a[2:])
```

**Output:**

```
[1 2 3 4 5]
[[1 2 3 4 5]]
```

[1.+0.j 2.+0.j 3.+0.j]
(2, 3)
[[1 2]
[3 4]
[5
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[[ 0 1 2]
[ 3 4 5]
[ 6 7 8]
[ 9 10 11]]
[[12 13 14]
[15 16 17]
[18 19 20]
[21 22 23
[2 4 6]
[2 4 6]
5
[2 3 4 5 6 7 8 9]

## 5.1.1.NumPy Creating Arrays

NumPy in Python is a library that is used to work with arrays and was created in 2005 by Travis Oliphant

Arrays are different from Python lists in several ways.

- First, NumPy arrays are multi-dimensional, while Python lists are one-dimensional.
- Second, NumPy arrays are homogeneous, while Python lists are heterogeneous. This means that all the elements of a NumPy array must be of the same type.
- Third, NumPy arrays are more efficient than Python lists.NumPy arrays can be created in several ways. One way is to create an array from a Python list.

Once you have created a NumPy array, you can manipulate it in various ways.
For example, you can change the shape of an array, or you can index into an array to access its elements. You can also perform mathematical operations on NumPy arrays, such as addition, multiplication, and division.

**Creating an Array:**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

**Output:**

[1 2 3 4 5]

**Dimensions- Arrays:**

**0-D Arrays:**

The following code will create a zero-dimensional array with a value 36.

```
import numpy as np
arr = np.array(36)
print(arr)
```

**Output:**

36

**1-Dimensional Array:**
The array that has Zero Dimensional arrays as its elements is a uni-dimensional or 1-D array.
The code below creates a 1-D array,

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

**Output:**

```
[1 2 3 4 5]
```

**Two Dimensional Arrays:**
2-D Arrays are the ones that have 1-D arrays as its element. The following code will create a 2-D array with 1,2,3 and 4,5,6 as its values.

```
import numpy as np
3
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1)
```

Output:

```
[[1 2 3]
[4 5 6]]
```

**Three Dimensional Arrays:**
Let us see an example of creating a 3-D array with two 2-D arrays:

```
import numpy as np
arr1 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]]) print(arr1)
```

Output:

```
[[[1 2 3]
[4 5 6]]
[[1 2 3]
[4 5 6]]]
```

To identify the dimensions of the array, we can **use ndim** as shown below:

```
import numpy as np
a = np.array(36)
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(d.ndim)
```

**Output:**

```
0
3
```

**Operations using NumPy**
Using NumPy, a developer can perform the following operations −

- **Mathematical and logical operations on arrays.**
- **Fourier transforms and routines for shape manipulation.**
- **Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.**

Every item in a ndarray takes the same size as the block in the memory. Each element in ndarray is an object of the **data-type object (called dtype**).

```
# minimum dimensions
import numpy as np
a = np.array([1, 2, 3,4,5], ndmin = 2)
print a
```

The output is as follows −

```
[[1, 2, 3, 4, 5]]
```

**Example**

```
# dtype parameter
import numpy as np
a = np.array([1, 2, 3], dtype = complex)
print a
```

The output is as follows −

```
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

## 5.1.2.NumPy – Data Types:

Here is a list of the different Data Types in NumPy:

- **bool :** Boolean (True or False) stored as a byte
- **int_ :** Default integer type (same as C long; normally either int64 or int32)
- **intc :** Identical to C int (normally int32 or int64)
- **intp:** An integer used for indexing (same as C ssize_t; normally either int32 or int64)
- **int8 :** Byte (-128 to 127)
- **int16:** Integer (-32768 to 32767)
- **float_ :** Shorthand for float64
- **float64:** Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
- **complex_:** Shorthand for complex128
- **complex64:** Complex number, represented by two 32-bit floats (real and imaginary components)
- **complex128:** Complex number, represented by two 64-bit floats (real and imaginary components)

NumPy numerical types are instances of dtype (data-type) objects, each having unique characteristics. The dtypes are available as np.bool_, np.float32, etc.

## 5.1.3. Functions of Numpy Array:

- **Data Type Objects (dtype)**

A data type object describes the interpretation of a fixed block of memory corresponding to an array

- **ndarray.ndim**

This array attribute returns the number of array dimensions.

**Example 1**

```
# an array of evenly spaced numbers
import numpy as np
a = np.arange(24)
```

```
print a
```

The output is as follows −

```
[0 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 17 18 19 20 21 22 23]
```

**Example 2**

```
# this is one dimensional array
import numpy as np
a = np.arange(24)
a.ndim
# now reshape it
b = a.reshape(2,4,3)
print b
# b is having three dimensions
```

The output is as follows −

```
[[[ 0,  1,  2]
[ 3,  4,  5]
[ 6,  7,  8]
[ 9, 10, 11]]
[[12, 13, 14]
[15, 16, 17]
[18, 19, 20]
[21, 22, 23]]]
```

- **numpy.itemsize**

This array attribute returns the length of each element of array in bytes.

**Example 1**

```
# dtype of array is int8 (1 byte)
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.int8)
print x.itemsize
```

The output is as follows −

```
1
```

**Example 2**

```
# dtype of array is now float32 (4 bytes)
import numpy as np
x = np.array([1,2,3,4,5], dtype = np.float32)
print x.itemsize
```

The output is as follows −

```
4
```

- **NumPy Array Shape**

The shape of an array is nothing but the number of elements in each dimension. To get the shape of an array, we can use a .shape attribute that returns a tuple indicating the number of elements.

```
import numpy as np
array1 = np.array([[2, 3, 4,5], [ 6, 7, 8,9]])
print(array1.shape)
Output: (2,4)
```

- **NumPy Array Reshape**

**1-D to 2-D:**

Array reshape is nothing but changing the shape of the array, through which one can add or remove a number of elements in each dimension. The following code will convert a 1-D array into 2-D array. The resulting will have 3 arrays having 4 elements

```
import numpy as np
array_1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr1 = array_1.reshape(3, 4)
print(newarr1)
```

Output:

```
[[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]]
```

**1-D to 3-D:**

The outer dimension will contain two arrays that have three arrays with two elements each.

```
import numpy as np
array_1= np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr1 = array_1.reshape(2, 3, 2)
print(newarr1)
```

Output:

```
[[[ 1 2]
[ 3 4]
[ 5 6]]
[[ 7 8]
[ 9 10]
[11 12]]]
```

- **Flattening arrays:**

Converting higher dimensions arrays into one-dimensional arrays is called flattening of arrays.

```
import numpy as np
arr1= np.array([[4,5,6], [7, 8, 9]])
newarr1 = arr1.reshape(-1)
print(newarr1)
```

Output :

```
[1 2 3 4 5 6]
```

- **NumPy Array Iterating**

Iteration through the arrays is possible using for loop.

Example 1:

```
import numpy as np
arr1 = np.array([1, 2, 3])
for i in arr1:
print(i)
```

Output: 1
2
3

Example 2:

```
import numpy as np
arr = np.array([[4, 5, 6], [1, 2, 3]])
for x in arr:
print(x)
```

Output: [4, 5, 6]
[1, 2, 3]
Example3:

```
import numpy as np
array1 = np.array([[1, 2, 3], [4, 5, 6]])
for x in array1:
for y in x:
print(y)
```

- **NumPy Array Join**

Joining is an operation of combining one or two arrays into a single array. In Numpy, the arrays are joined by axes. The concatenate() function is used for this operation, it takes a sequence of arrays that are to be joined, and if the axis is not specified, it will be taken as 0.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
finalarr = np.concatenate((arr1, arr2))
print(finalarr)
```

Output: [1 2 3 4 5 6]
The following code joins the specified arrays along the rows

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
finalarr = np.concatenate((arr1, arr2), axis=1)
print(finalarr)
```

**Output:**
[[1 2 5 6]
[3 4 7 8]]

- **NumPy – Broadcasting:**

The term **broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed.

**Example 1**

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
c = a * b
print c
```

Its output is as follows −[10 40 90 160]
If the dimensions of the two arrays are dissimilar, element-to-element operations are not possible. However, operations on arrays of non-similar shapes is still possible in NumPy, because of the broadcasting capability. The smaller array is **broadcast** to the size of the larger array so that they have compatible shapes.

## 5.1.4 NumPy – Indexing & Slicing:

Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.
As mentioned earlier, items in ndarray object follows zero-based index. Three types of indexing methods are available − **field access, basic slicing** and **advanced indexing**.
Basic slicing is an extension of Python's basic concept of slicing to n dimensions. A Python slice object is constructed by giving **start, stop**, and **step** parameters to the built-in **slice** function. This slice object is passed to the array to extract a part of array.

**Example 1**

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2)
print a[s]
```

Its output is as follows −
[2 4 6]
n the above example, an **ndarray** object is prepared by **arange()** function. Then a slice object is defined with start, stop, and step values 2, 7, and 2 respectively. When this slice object is passed to the ndarray, a part of it starting with index 2 up to 7 with a step of 2 is sliced.
The same result can also be obtained by giving the slicing parameters separated by a colon : (start:stop:step) directly to the **ndarray** object.

**Example 2**

```
import numpy as np
a = np.arange(10)
b = a[2:7:2]
print b
```

Here, we will get the same output − [2 4 6]
If only one parameter is put, a single item corresponding to the index will be returned. If a: is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with: between them) is used, items between the two indexes (not including the stop index) with default step one are sliced.

**Example 3**

```
# slice single item
import numpy as np
a = np.arange(10)
b = a[5]
print b
```

Its output is as follows −
5

**Example 4**

```
# slice items starting from index
import NumPy as np
a = np.arange(10)
print a[2:]
```

Now, the output would be −
[2 3 4 5 6 7 8 9]

**Example 5**

```
# slice items between indexes
import numpy as np
```

```
a = np.arange(10)
print a[2:5]
```

Here, the output would be −

```
[2  3  4]
```

The above description applies to multi-dimensional **ndarray** too.

- **NumPy Array Split**

As we know, split does the opposite of join operation. Split breaks a single array as specified. The function array_split() is used for this operation and one has to pass the number of splits along with the array.

```
import numpy as np
arr1 = np.array([7, 8, 3, 4, 1, 2])
finalarr = np.array_split(arr1, 3)
print(finalarr)
```

Output: [array([7, 8]), array([3, 4]), array([1, 2])]

Look at an exceptional case where the no of elements is less than required and observe the output

Example :

```
import numpy as np
array_1 = np.array([4, 5, 6,1,2,3])
finalarr = np.array_split(array_1, 4)
print(finalarr)
```

Output : [array([4, 5]), array([6, 1]), array([2]), array([3])]

- **NumPy – Advanced Indexing**

It is possible to make a selection from ndarray that is a non-tuple sequence, ndarray object of integer or Boolean data type, or a tuple with at least one item being a sequence object. Advanced indexing always returns a copy of the data. As against this, the slicing only presents a view.

There are two types of advanced indexing − **Integer** and **Boolean**.

**Integer Indexing**

This mechanism helps in selecting any arbitrary item in an array based on its N-dimensional index. Each integer array represents the number of indexes into that dimension. When the index consists of as many integer arrays as the dimensions of the target ndarray, it becomes straightforward.

In the following example, one element of the specified column from each row of ndarray object is selected. Hence, the row index contains all row numbers, and the column index specifies the element to be selected.

**Example 1**

```
import numpy as np
x = np.array([[1, 2], [3, 4], [5, 6]])
y = x[[0,1,2], [0,1,0]]
print y
```

Its output would be as follows −

```
[1  4  5]
```

The selection includes elements at (0,0), (1,1) and (2,0) from the first array.

In the following example, elements placed at corners of a 4X3 array are selected. The row indices of selection are [0, 0] and [3,3] whereas the column indices are [0,2] and [0,2].

Advanced and basic indexing can be combined by using one slice (:) or ellipsis (…) with an index array. The following example uses a slice for the advanced index for column. The result is the same when a slice is used for both. But advanced index results in copy and may have different memory layout.

- **Boolean Array Indexing**

This type of advanced indexing is used when the resultant object is meant to be the result of Boolean operations, such as comparison operators.

**Example 1**

In this example, items greater than 5 are returned as a result of Boolean indexing.

```
import numpy as np
x = np.array([[ 0,  1,  2],[ 3,  4,  5],[ 6,  7,  8],[ 9, 10, 11]])
print 'Our array is:'
print x
print '\n'
# Now we will print the items greater than 5
print 'The items greater than 5 are:'
print x[x > 5]
```

The output of this program would be −

```
Our array is:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
The items greater than 5 are:
[ 6  7  8  9 10 11]
```

## 5.1.5. NumPy – Mathematical Functions

Quite understandably, NumPy contains a large number of various mathematical operations. NumPy provides standard trigonometric functions, functions for arithmetic operations, handling complex numbers, etc.

- **Trigonometric Functions**

NumPy has standard trigonometric functions which return trigonometric ratios for a given angle in radians.

**Example**

```
import numpy as np
a = np.array([0,30,45,60,90])
print 'Sine of different angles:'
# Convert to radians by multiplying with pi/180
print np.sin(a*np.pi/180)
print '\n'
print 'Cosine values for angles in array:'
print np.cos(a*np.pi/180)
print '\n'
print 'Tangent values for given angles:'
print np.tan(a*np.pi/180)
```

Here is its output −

```
Sine of different angles:
[ 0.         0.5        0.70710678  0.8660254   1.        ]
Cosine values for angles in array:
[  1.00000000e+00   8.66025404e-01   7.07106781e-01   5.00000000e-01
```

6.12323400e-17]
Tangent values for given angles:
[ 0.00000000e+00 5.77350269e-01 1.00000000e+00 1.73205081e+00
1.63312394e+16]

- **NumPy – Statistical Functions**

NumPy has quite a few useful statistical functions for finding minimum, maximum, percentile standard deviation and variance, etc. from the given elements in the array. The functions are explained as follows −

**numpy.amin() and numpy.amax()numpy.amin() and numpy.amax()**

These functions return the minimum and the maximum from the elements in the given array along the specified axis.

**Example**

```
import numpy as np
a = np.array([[3,7,5],[8,4,3],[2,4,9]])
print 'Our array is:'
print a
print '\n'
print 'Applying amin() function:'
print np.amin(a,1)
print '\n'
print 'Applying amin() function again:'
print np.amin(a,0)
print '\n'
print 'Applying amax() function:'
print np.amax(a)
print '\n'
print 'Applying amax() function again:'
print np.amax(a, axis = 0)
```

It will produce the following output −

```
Our array is:
[[3 7 5]
[8 4 3]
[2 4 9]]
Applying amin() function:
[3 3 2]
Applying amin() function again:
[2 4 3]
Applying amax() function:
9
Applying amax() function again:
[8 7 9]
```

- **Square Root & Standard Deviation**

There are various mathematical functions that can be performed using python numpy. You can find the square root, standard deviation of the array. So, let's implement these operations:

```
1  import numpy as np
2  a=np.array([(1,2,3),(3,4,5,)])
3  print(np.sqrt(a))
4  print(np.std(a))
```

Output – [[ 1. 1.41421356 1.73205081]
[ 1.73205081 2. 2.23606798]]
1.29099444874

## 5.2. Pandas Package:

Pandas is a BSD (Berkeley Software Distribution) licensed open source library. This popular library is widely used in the field of data science. They are primarily used for data analysis, manipulation, cleaning, etc. Pandas allow for simple data modeling and data analysis operations without the need to switch to another language such as R.

Pandas make sure that the entire process of manipulating data will be easier. Support for operations such as Re-indexing, Iteration, Sorting, Aggregations, Concatenations and Visualizations are among the feature highlights of Pandas. Pandas have so many inbuilt methods for grouping, combining data, and filtering, as well as time-series functionality.

Usually, Python libraries use the following types of data:

- Data in a dataset.
- Time series containing both ordered and unordered data.
- Rows and columns of matrix data are labelled.
- Unlabeled information
- Any other type of statistical information

Pandas can do a wide range of tasks, including:

- The data frame can be sliced using Pandas.
- Data frame joining and merging can be done using Pandas.
- Columns from two data frames can be concatenated using Pandas.
- In a data frame, index values can be changed using Pandas.
- In a column, the headers can be changed using Pandas.
- Data conversion into various forms can also be done using Pandas and many more.

Data is used across industries to drive business decisions, and data scientists rely on Pandas, a Python library, to organize it in meaningful ways. Pandas is so crucial in these three, real world examples.

Pandas is a popular, powerful Python library and its main function is for data exploration, manipulation, and analysis. One of the main reasons it's so popular is because real-world data can be *messy* and professionals spend a lot of time cleaning it up before they can work with it. Pandas's superpower is making that data simple to import, clean, and transform to be usable for analysis. It also helps data scientists prepare data before they train models.

**Real World Examples of Pandas**
Since Pandas is used to prepare and explore data for preliminary analysis, it's used across industries and by many levels of data professionals. Here are 3 examples of how Pandas is used in the real world.

1. **Netflix Recommendations:**
Data scientists for video subscription services like Netflix build recommendation systems in order to offer suggestions to their customers. Before data scientists can build and train their recommendation model, they have to go through pre-processing to understand the data. Pandas is an excellent library for all of the pre-analysis and exploration.

2. **Churn Rate in Banking:**
Customer churn is a measurement that captures the ratio of customers that drop a product or service. Essentially, churn rate is capturing how many customers have been lost. For example, a bank might look at which customers closed their accounts, or switched to a different product offering. It's an important metric for determining the quality or target demographics for a product, as well as when and why a customer might leave. A data scientist might look at this metric and figure out the characteristics of the lost customer, such as if most of them were female or paid by credit card. They'll also look and see what types of customers stayed and used certain products.

3. **Retail Sales Data Analytics:**
Retailers are also interested in data and pulling key findings from their customer data to help improve their product and service. A data scientist or data analyst for a retailer may pull all kinds of customer data using Pandas — and this data may be a very large set that draws from all departments. Their job is then to figure out trends to allow stores to make better decisions.

**Example Program:**

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
print(S)
print(S.index)
print(S.values)
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
fruits = ['apples','oranges','cherries','pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
print(S)
fruits = ['apples','oranges','cherries','pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S:", sum(S))
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges','cherries','pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

**OUTPUT:-**

```
0   11
1   28
2   72
3    3
4    5
5    8
dtype: int64
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'><class 'numpy.ndarray'>
apples      20
oranges     33
cherries    52
pears       10
dtype: int64apples      37
oranges     46
cherries    83
pears       42
dtype: int64sum of S: 115
cherries       83.0
oranges        46.0
peaches         NaN
pears         42.0
raspberries     NaN
dtype: float64>
```

## 5.2.1. Pandas Features:

1. Data structures with labelled axes supporting automatic or explicit data alignment capable of handling both time-series and non-time-series data
2. Ability to add and remove columns on the fly
3. Flexible handling of missing data
4. SQL-like merge and other relational operations
5. Tools for reading and writing data between in-memory data structures and different file formats (csv, xls, HDF5, SQL databases)
6. Reshaping and pivoting of data sets
7. Label-based slicing, fancy indexing, and sub setting of large data sets
8. Group by engine allowing split-apply-combine operations on data sets
9. Time series-functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging
10. Hierarchical axis indexing to work with high-dimensional data in a lower-dimensional data structure

## 5.2.2. Data Sets:

The word pandas is an acronym which is derived from "Python and data analysis" and "panel data".

It is used for data manipulation and analysis. It provides special data structures and operations for the manipulation of numerical tables and time series.

We will start with the following two important data structures of Pandas:

- Series and
- DataFrame

### 5.2.2.1 Series Data Set:

A Series is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

We define a simple Series object in the following example by instantiating a Pandas Series object with a list. We will later see that we can use other data objects for example Numpy arrays and dictionaries as well to instantiate a Series object.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
Print(S)
```

**OUTPUT:**

```
0   11
1   28
2   72
3    3
4    5
5    8
dtype: int64
```

We haven't defined an index in our example, but we see two columns in our output: The right column contains our data, whereas the left column contains the index. Pandas created a default index starting with 0 going to 5, which is the length of the data minus 1.

We can directly access the index and the values of our Series S:

```
print(S.index)
print(S.values)
```

**OUTPUT:**

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

If we compare this to creating an array in numpy, we will find lots of similarities:

```
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
```

**OUTPUT:**

```
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

So far our Series have not been very different to ndarrays of Numpy. This changes, as soon as we start defining Series objects with individual indices:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
Print(S)
```

**OUTPUT:**

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

A big advantage to NumPy arrays is obvious from the previous example: We can use arbitrary indices.

If we add two series with the same indices, we get a new series with the same index and the correponding values will be added:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

**OUTPUT:**

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
sum of S:  115
```

The indices do not have to be the same for the Series addition. The index will be the "union" of both indices. If an index doesn't occur in both Series, the value for this Series will be NaN:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

**OUTPUT:**

```
cherries       83.0
oranges        46.0
peaches         NaN
pears         42.0
raspberries     NaN
dtype: float64
```

In principle, the indices can be completely different, as in the following example. We have two indices. One is the Turkish translation of the English fruit names:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

fruits_tr = ['elma', 'portakal', 'kiraz', 'armut']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits_tr)
print(S + S2)
```

**OUTPUT:**

```
apples     NaN
armut      NaN
cherries   NaN
elma       NaN
kiraz      NaN
oranges    NaN
pears      NaN
portakal   NaN
dtype: float64
```

**Indexing**

It's possible to access single values of a Series.

```
print(S['apples'])
```

**OUTPUT:**

```
20
```

This lookes like accessing the values of dictionaries through keys.

However, Series objects can also be accessed by multiple indexes at the same time. This can be done by packing the indexes into a list. This type of access returns a Pandas Series again:

```
print(S[['apples', 'oranges', 'cherries']])
```

**OUTPUT:**

```
apples      20
oranges     33
cherries    52
dtype: int64
```

Similar to Numpy we can use scalar operations or mathematical functions on a series:

```
import numpy as np
print((S + 3) * 4)
print("======================")
print(np.sin(S))
```

**OUTPUT:**

```
apples      92
oranges     144
cherries    220
pears       52
dtype: int64
======================
apples      0.912945
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

**Creating Series Objects from Dictionaries**

We can even use a dictionary to create a Series object. The resulting Series contains the dict's keys as the indices and the values as the values.

```
cities = {"London":    8615246,
          "Berlin":    3562166,
          "Madrid":    3165235,
          "Rome":      2874038,
          "Paris":     2273305,
          "Vienna":    1805681,
          "Bucharest": 1803425,
          "Hamburg":   1760433,
          "Budapest":  1754000,
          "Warsaw":    1740119,
          "Barcelona": 1602386,
          "Munich":    1493900,
          "Milan":     1350680}

city_series = pd.Series(cities)
print(city_series)
```

**OUTPUT:**

```
London      8615246
Berlin      3562166
Madrid      3165235
Rome        2874038
Paris       2273305
Vienna      1805681
Bucharest   1803425
Hamburg     1760433
Budapest    1754000
Warsaw      1740119
Barcelona   1602386
Munich      1493900
Milan       1350680
dtype: int64
```

## 5.2.3 Creating, Describing and Manipulating DataFrame With Examples:

One simplest way to create a pandas DataFrame is by using its constructor. Besides this, there are many other ways to create a DataFrame in pandas. For example, creating DataFrame from a list, created by reading a CSV file, creating it from a Series, creating empty DataFrame, and many more.

Python pandas is widely used for data science/data analysis and machine learning applications. It is built on top of another popular package named Numpy, which provides scientific computing in Python. pandas DataFrame is a 2-dimensional labeled data structure with rows and columns (columns of potentially different types like integers, strings, float, None, Python objects e.t.c). You can think of it as an excel spreadsheet or SQL table.

- **Create pandas DataFrame**

One of the easiest ways to create a pandas DataFrame is by using its constructor. DataFrame constructor takes several optional params that are used to specify the characteristics of the DataFrame.

Below is the **syntax of the DataFrame constructor**.

**# DataFrame constructor syntax**

**pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)**

Now, let's create a DataFrame from a list of lists (with a few rows and columns).

```
# Create pandas DataFrame from List
import pandas as pd
technologies = [ ["Spark",20000, "30days"],
                 ["Pandas",25000, "40days"], ]
```

```
df=pd.DataFrame(technologies)
print(df)
```

Since we have not given index and column labels, DataFrame by default assigns incremental sequence numbers as labels to both rows and columns.

Output:

```
      0     1      2
0  Spark  20000  30days
1  Pandas  25000  40days
```

Column names with sequence numbers don't make sense as it's hard to identify what data holds on each column hence, it is always best practice to provide column names that identify the data it holds. Use column param and index param to provide column & custom index respectively to the DataFrame.

```
# Add Column & Row Labels to the DataFrame

column_names=["Courses","Fee","Duration"]

row_label=["a","b"]

df=pd.DataFrame(technologies,columns=column_names,index=row_label)

print(df)
```

Yields below output. Alternatively, you can also add columns labels to the existing DataFrame.

```
  Courses    Fee Duration

a   Spark  20000   30days

b  Pandas  25000   40days
```

By default, pandas identify the data types from the data and assign's to the DataFrame. df.dtypes returns the data type of each column.

```
Courses     object

Fee         int64

Duration    object

dtype: object
```

You can also assign custom data types to columns.

```
# set custom types to DataFrame

types={'Courses': str,'Fee':float,'Duration':str}
```

```
df=df.astype(types)
```

- **Create DataFrame from the Dic (dictionary).**

Another most used way to create pandas DataFrame is from the python Dict (dictionary) object. This comes in handy if you wanted to convert the dictionary object into DataFrame. Key from the Dict object becomes column and value convert into rows.

```
# Create DataFrame from Dict

technologies = {

   'Courses':["Spark","Pandas"],

   'Fee' :[20000,25000],

   'Duration':['30days','40days']

         }

df = pd.DataFrame(technologies)

print(df)
```

- **Create DataFrame with Index**

By default, DataFrame add's a numeric index starting from zero. It can be changed with a custom index while creating a DataFrame.

```
# Create DataFrame with Index.

technologies = {

   'Courses':["Spark","Pandas"],

   'Fee' :[20000,25000],

   'Duration':['30days','40days']

         }

index_label=["r1","r2"]

df = pd.DataFrame(technologies, index=index_label)

print(df)
```

- **Creating Dataframe from list of dicts object**

Sometimes we get data in JSON string (similar dict), you can convert it to DataFrame as shown below.

```
# Creates DataFrame from list of dict

technologies = [{'Courses':'Spark', 'Fee': 20000, 'Duration':'30days'},

    {'Courses':'Pandas', 'Fee': 25000, 'Duration': '40days'}]



df = pd.DataFrame(technologies)

print(df)
```

- **Creating DataFrame From Series**

By using concat() method you can create Dataframe from multiple Series. This takes several params, for the scenario we use list that takes series to combine and axis=1 to specify merge series as columns instead of rows.

```
# Create pandas Series

courses = pd.Series(["Spark","Pandas"])

fees = pd.Series([20000,25000])

duration = pd.Series(['30days','40days'])

# Create DataFrame from series objects.

df=pd.concat([courses,fees,duration],axis=1)

print(df)

#Outputs

#       0     1       2

#0   Spark  20000  30days

#1  Pandas  25000  40days
```

- **Add Column Labels**

As you see above, by default concat() method doesn't add column labels. You can do so as below.

```
# Assign Index to Series

index_labels=['r1','r2']
```

```
courses.index = index_labels

fees.index = index_labels

duration.index = index_labels

# Concat Series by Changing Names

df=pd.concat({'Courses': courses,

        'Course_Fee': fees,

        'Course_Duration': duration},axis=1)

print(df)

# Outputs

#   Courses  Course_Fee Course_Duration

#r1   Spark       20000         30days

#r2  Pandas       25000         40days
```

- **Creating DataFrame using zip() function**

Multiple lists can be merged using zip() method and the output is used to create a DataFrame.

```
# Create Lists

Courses = ['Spark', 'Pandas']

Fee = [20000,25000]

Duration = ['30days','40days']


# Merge lists by using zip().

tuples_list = list(zip(Courses, Fee, Duration))

df = pd.DataFrame(tuples_list, columns = ['Courses', 'Fee', 'Duration'])
```

- **Create an empty DataFrame in pandas:**

Sometimes you would need to create an empty pandas DataFrame with or without columns. This would be required in many cases, below is one example.

While working with files, sometimes we may not receive a file for processing, however, we still need to create a DataFrame manually with the same column names we expect. If we don't create with the same columns, our operations/transformations (like union's) on DataFrame fail as we refer to the columns that may not be present.

To handle situations similar to these, we always need to create a DataFrame with the expected columns, which means the same column names and datatypes regardless of the file exists or empty file processing.

```
# Create Empty DataFrame

df = pd.DataFrame()

print(df)

# Outputs

#Empty DataFrame

#Columns: []

#Index: []
```

To create an empty DataFrame with just column names but no data.

```
# Create Empty DataFraem with Column Labels

df = pd.DataFrame(columns = ["Courses","Fee","Duration"])

print(df)

# Outputs

#Empty DataFrame

#Columns: [Courses, Fee, Duration]

#Index: []
```

## 5.3.5 Import DataFrame From CSV File:

In real-time we are often required to read the contents of CSV files and create a DataFrame. In pandas, creating a DataFrame from CSV is done by using pandas.read_csv() method. This returns a DataFrame with the contents of a CSV file.

```
# Create DataFrame from CSV file
df = pd.read_csv('data_file.csv')
```

- **Create From Another DataFrame**

Finally, you can also copy a DataFrame from another DataFrame using copy() method.

```
# Copy DataFrame to another
df2=df.copy()
print(df2)
```

## 5.2.6 Missing Values in Pandas:

- **NaN - Missing** Data

One problem in dealing with data analysis tasks consists in missing data. Pandas makes it as easy as possible to work with missing data.

If we look once more at our previous example, we can see that the index of our series is the same as the keys of the dictionary we used to create the cities_series. Now, we want to use an index which is not overlapping with the dictionary keys. We have already seen that we can pass a list or a tuple to the keyword argument 'index' to define the index. In our next example, the list (or tuple) passed to the keyword parameter 'index' will not be equal to the keys. This means that some cities from the dictionary will be missing and two cities ("Zurich" and "Stuttgart") don't occur in the dictionary.

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

**OUTPUT:**

```
London      8615246.0
Paris       2273305.0
Zurich            NaN
Berlin      3562166.0
Stuttgart         NaN
Hamburg     1760433.0
dtype: float64
```

Due to the Nan values the population values for the other cities are turned into floats. There is no missing data in the following examples, so the values are int:

```
my_cities = ["London", "Paris", "Berlin", "Hamburg"]

my_city_series = pd.Series(cities,
```

```
                    index=my_cities)
my_city_series
```

**OUTPUT:**

```
London     8615246
Paris      2273305
Berlin     3562166
Hamburg    1760433
dtype: int64
```

**The Methods isnull() and notnull()**

We can see, that the cities, which are not included in the dictionary, get the value NaN assigned. NaN stands for "not a number". It can also be seen as meaning "missing" in our example.

We can check for missing values with the methods isnull and notnull:

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
          "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,  index=my_cities)
print(my_city_series.isnull())
```

**OUTPUT:**

```
London      False
Paris       False
Zurich       True
Berlin      False
Stuttgart    True
Hamburg     False
dtype: bool
```

```
print(my_city_series.notnull())
```

**OUTPUT:**

```
London       True
Paris        True
Zurich      False
Berlin       True
Stuttgart   False
Hamburg      True
dtype: bool
```

**Connection between NaN and None**

We get also a NaN, if a value in the dictionary has a None:

```
d = {"a":23, "b":45, "c":None, "d":0}
S = pd.Series(d)
print(S)
```

**OUTPUT:**

```
a    23.0
b    45.0
c     NaN
d     0.0
dtype: float64
```

```
pd.isnull(S)
```

**OUTPUT:**

```
a    False
b    False
c     True
d    False
dtype: bool
```

```
pd.notnull(S)
```

**OUTPUT:**

```
a     True
b     True
c    False
d     True
dtype: bool
```

**Filtering out Missing Data**

It's possible to filter out missing data with the Series method dropna. It returns a Series which consists only of non-null data:

```
print(my_city_series.dropna())
```

**OUTPUT:**

```
London     8615246.0
Paris      2273305.0
Berlin     3562166.0
Hamburg    1760433.0
dtype: float64
```

**Filling in Missing Data**

In many cases you don't want to filter out missing data, but you want to fill in appropriate data for the empty gaps. A suitable method in many situations will be fillna:

```
print(my_city_series.fillna(0))
```

**OUTPUT:**

```
London       8615246.0
Paris        2273305.0
Zurich             0.0
Berlin       3562166.0
Stuttgart          0.0
Hamburg      1760433.0
dtype: float64
```

Okay, that's not what we call "fill in appropriate data for the empty gaps". If we call fillna with a dict, we can provide the appropriate data, i.e. the population of Zurich and Stuttgart:

```
missing_cities = {"Stuttgart":597939, "Zurich":378884}
my_city_series.fillna(missing_cities)
```

**OUTPUT:**

```
London      8615246.0
Paris       2273305.0
Zurich       378884.0
Berlin      3562166.0
Stuttgart    597939.0
Hamburg     1760433.0
dtype: float64
```

We still have the problem that integer values - which means values which should be integers like number of people - are converted to float as soon as we have NaN values. We can solve this problem now with the method 'fillna':

```
cities = {"London":   8615246,
          "Berlin":   3562166,
          "Madrid":   3165235,
          "Rome":     2874038,
          "Paris":    2273305,
          "Vienna":   1805681,
          "Bucharest":1803425,
          "Hamburg":  1760433,
          "Budapest": 1754000,
          "Warsaw":   1740119,
          "Barcelona":1602386,
          "Munich":   1493900,
          "Milan":    1350680}

my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities, index=my_cities)
my_city_series = my_city_series.fillna(0).astype(int)
print(my_city_series)
```

**OUTPUT:**

```
London      8615246
Paris       2273305
Zurich            0
Berlin      3562166
Stuttgart         0
Hamburg     1760433
dtype: int64
```

# 5.3 Matplotlib:

Matplotlib is one such solution for Python users that need to visualize their data in order to make essential statistical conclusions. It is a complete plotting package that is useful for Python and NumPy users. This article will assist you in comprehending the matplotlib library, which is frequently used in the industry. Matplotlib includes a wide range of graphical tools and is simple to use.

**Scope**

- We will learn about matplotlib in Python in this tutorial.
- Why is it so popular when it comes to data visualization?
- Using several examples to discuss matplotlib in details
- The functionalities of Matplotlib are properly discussed in this article.

**Introduction**

Data visualization is one of the important skills that data scientists are expected to have. Visualization techniques can be used to understand and address the majority of business challenges. **Exploratory Data Analysis (EDA)** and Graphical Plots are the two main components of visualization. Effective visualization helps users in understanding data trends and solving business problems more efficiently. Another advantage of visualization is that it reduces complex data to a more readable manner.

A visual is significantly simpler to understand than text for most people. Visualization is the most effective communication tool for analyzing and interpreting data. It allows customers to quickly interpret vast amounts of data. Trends, correlations, patterns, distributions, and so on can all be better understood through data visualization.

For data visualization, there are numerous tools and technologies available in the market, with Python being the most popular. Python has many libraries for data visualization; a few of the most **prominent graphic libraries** are:

- Matplotlib
- Seaborn
- Pandas visualization
- Plotly

**What is Matplotlib in Python?**

Matplotlib is the basic plotting library of the Python programming language. Among Python visualization packages, it is the most widely used.

Matplotlib is exceptionally fast at a variety of operations. It can export visualizations to all popular formats, including **PDF, SVG, JPG, PNG, BMP, and GIF**.

It can create line **graphs, scatter plots, histograms, bar charts, error charts, pie charts, box plots**, and many other visualization styles. 3D charting is also possible with Matplotlib.

Matplotlib serves as the foundation for several Python libraries. Matplotlib is used to build pandas and Seaborn, for example. They make it possible to access Matplotlib's methods with less code.

John Hunter founded the Matplotlib project in 2002. Matplotlib was created during post-doctoral study in Neurobiology to show **Electrocorticography (ECoG)** data from epileptic patients.

Matplotlib, an open-source plotting toolkit for the Python programming language, has become the most extensively used plotting library. It was used to visualize data during the 2008 landing of the **Phoenix spacecraft**.

**Examples of Matplotlib in Python**

**Example 1:** Create several line charts on a shared plot by plotting two data ranges on the same chart. The data range(s) to be plotted are as follows:

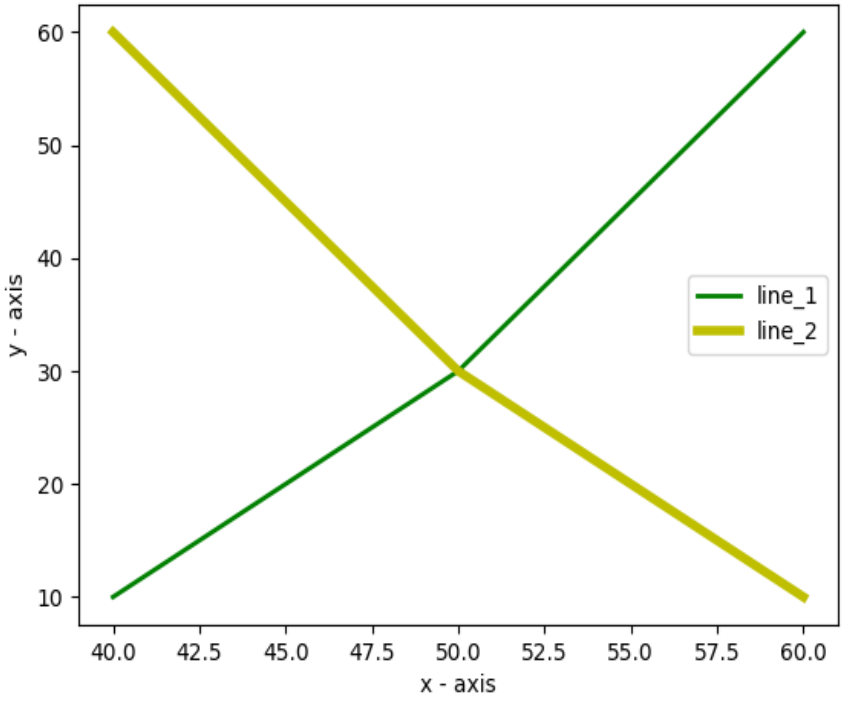
line 1 points x1 = [40,50,60] y1 = [10,30,60]

line 2 points x2 = [40,50,60] y2 = [60,30,10]

The legend can be seen in the **upper left corner**. The X axis is labeled "X – Axis," and the Y axis is labeled "Y – Axis." The lines should be of varying width and color.

```
import matplotlib.pyplot as plt
# points at line 1
x1 = [40,50,60]
y1 = [10,30,60]
# points at line 2
x2 = [40,50,60]
y2 = [60,30,10]

plt.xlabel('x - axis')
plt.ylabel('y - axis')

# Display the figure.
plt.plot(x1,y1, color='g', linewidth = 2,  label = 'line_1')
plt.plot(x2,y2, color='y', linewidth = 4,  label = 'line_2')

plt.legend()
plt.show()
```

**Output:**



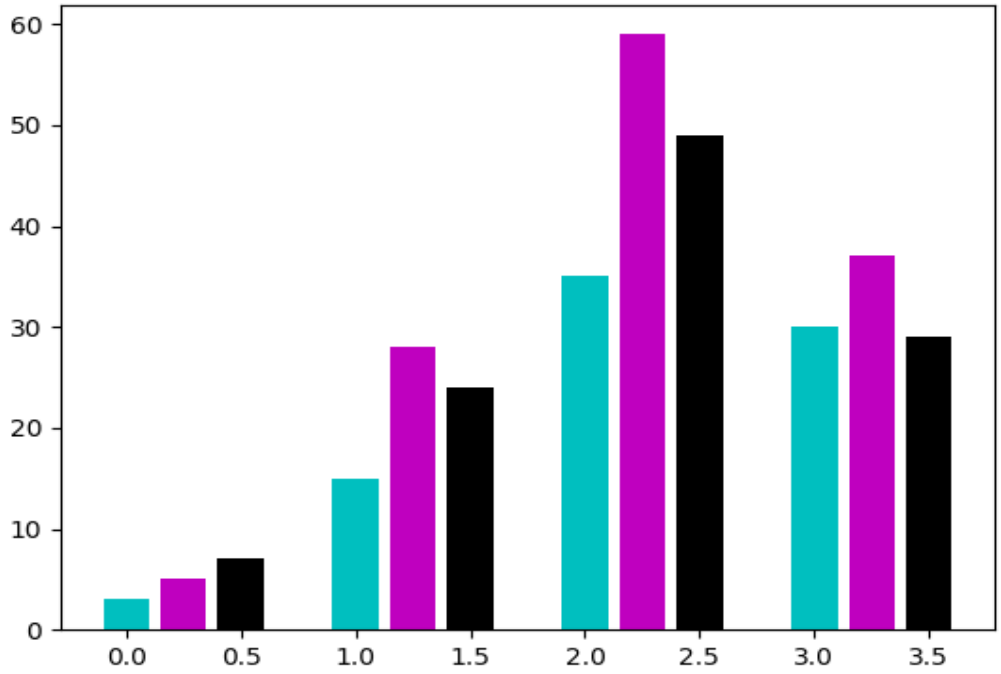

**Example 2:**

Make a bar chart using the following data from three runners over four marathons:

Runners = [[3,15,35,30],[5,28,59,37], [7,24,49,29]]

```
import numpy as np
import matplotlib.pyplot as plt
Run = [[3,15,35,30],[5,28,59,37], [7,24,49,29]]
X=np.arange(4)
plt.bar(X+0.00,Run[0],color='c',width=0.20)
plt.bar(X+0.25,Run[1],color='m',width=0.20)
plt.bar(X+0.5,Run[2],color='k',width=0.20)
plt.show()
```

**Output:**

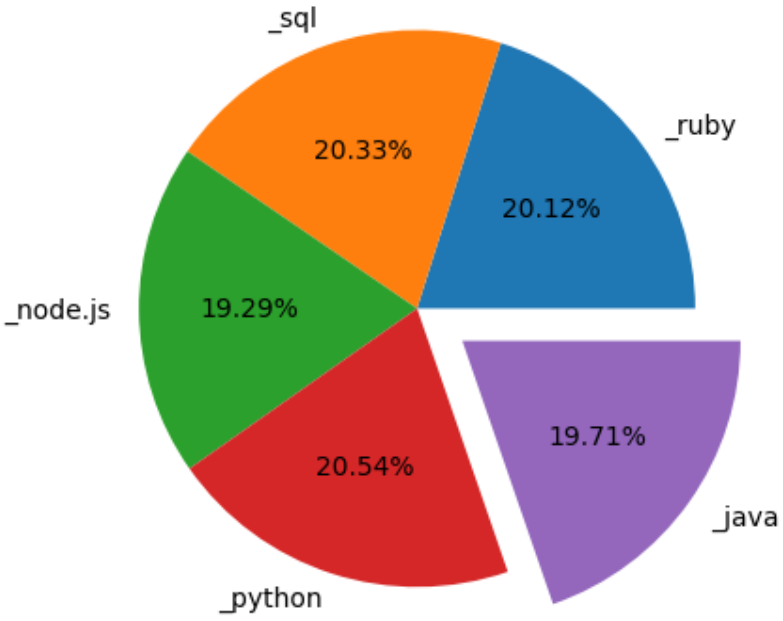



**Example 3:** Create a pie chart for a student's grades, using the following data:

* _RUBY - 97 * _SQL - 98 * _NODE.JS - 93 * _PYTHON- 99 * _JAVA - 95

```
import matplotlib.pyplot as plt
_Marks=[97,98,93,99,95]
_Subjects=["_ruby","_sql","_node.js","_python","_java"]
plt.axis("equal")
plt.pie(_Marks,labels=_Subjects,explode=[0,0,0,0,0.2],autopct="%1.2f%%")
plt.show()
```

**Output:**




**Features of Matplotlib**

- It is a data visualization package for the Python programming language.
- It is the most basic and widely used method for plotting data in Python.
- It includes tools for creating publication-standard plots and figures in a number of export formats and environments **(pycharm, jupyter notebook) across platforms**.
- It includes a procedural interface called **Pylab**, which is supposed to behave similarly to MATLAB, a programming language popular among scientists and researchers. **MATLAB is a commercial application** that is not open source.
- It's comparable to MATLAB plotting in that it gives users complete control over fonts, lines, colours, styles, and axes attributes.
- Matplotlib with **NumPy** might be considered the open source version of MATLAB.
- It is a great approach to create high-quality static graphics for publications and professional presentations.
- It also works with a variety of different third-party libraries and packages, allowing matplotlib to expand its capabilities.
- It is clear that matplotlib with its various compatible third-party libraries provide users with powerful tools to visualize a variety of data.

**Applications of Matplotlib:**

Matplotlib creates significant figures in a number of physical and graphical formats across various platforms.

- Matplotlib is a Python library that can be used in scripts.
- Matplotlib is a Python/IPython library that can be used in shells.

- Web application servers can utilise Matplotlib.
- Matplotlib is a graphical user interface toolkit that may be used in a variety of graphical user interface toolkits.

## 5.4 Tkinter Package:

**Tkinter** was created to equip modern developers with a standard interface to the Tk GUI toolkit with its Python bindings. In Tkinter's world, most of the **visual elements that we're familiar with are called widgets,** and each of these widgets offers a different level of customizability.
Tkinter comes baked into current Python installers for all major operating systems and offers a host of commonly used elements that we're sure you must be familiar with.

Some of those visual elements have been listed below:

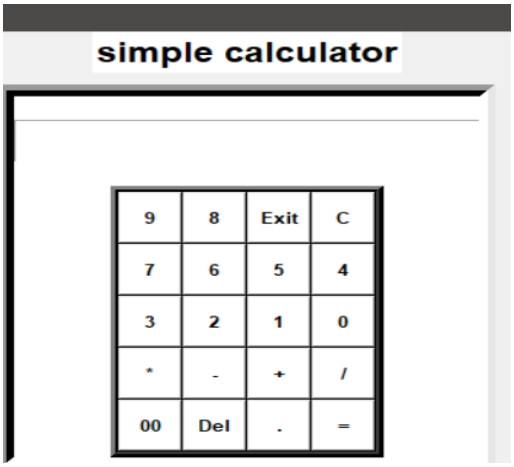
- Frame: for providing a structure to your application
- Buttons: used for taking input from the user
- Checkbuttons: used for making selections
- Labels: for displaying textual information
- File Dialogs: for uploading or downloading files to/from the application
- Canvas: provides a space for drawing/painting things like graphs and plots

**Example Program:** simple calculator using tkinter?

```
import sys
import pyttsx3
from tkinter import *
def command(event):
    global userval
    input=event.widget.cget("text")
    if input=="Exit":
system.say('Exit successfully.')
system.runAndWait()
quit()
elif input=="C":
userval.set("")
entry.update()
system.say('Your screen is cleared.')
system.runAndWait()
elif input=="=":
ans=userval.get()
userval.set(eval(ans))
entry.update()
system.say(f'Youranswerr is {userval.get()}')
system.runAndWait()
elif input=="Del":
        value=userval.get()
        v=value.replace(value[-1:],"")
userval.set(v)
entry.update()
    else:
userval.set(userval.get()+input)
```

```
entry.update()
root=Tk()
system = pyttsx3.init()
root.geometry("400x400")
Label(text="simple calculator",font="nothing 18 bold",fg="black",bg="white").pack()
list=["9","8","Exit","C","7","6","5","4",
"3","2","1","0","*","-","+","/","00","Del",".","="]
f1=Frame(root,borderwidth=10,relief=SUNKEN,bg="white")
f1.pack(pady=10)
userval=StringVar()
entry=Entry(f1,textvariable=userval,font="Arial 20 bold")
entry.pack(pady=20)
f2=Frame(f1,borderwidth=5,bg="black",relief=RAISED)
f2.pack()
def button():
    a=0
    z=0
    for i,item in enumerate(list):
        b=Button(f2,text=list[i],height=2,width=4,font="dontknow 10 bold",bg="white")
b.bind("<Button-1>",command)
        if i%4==0:
            a+=1
        if i%4==0:
            z=0
b.grid(row=a,column=z)
        z+=1
button()
root.mainloop()
```

OUTPUT:-

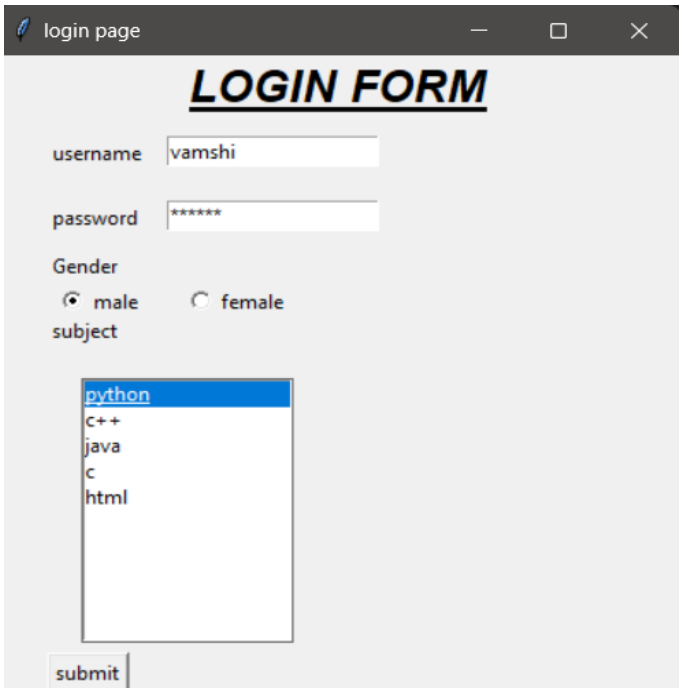



Example 2: write a program to build login page using tkinter?

```
import tkinter as tk
from tkinter import *
top = tk.Tk()
top.title("login page")
top.geometry("400x400")
lb0= tk.Label(top,text="LOGIN FORM")
lb0.config(font=('callibri',20,'italic bold underline'))
lb0.pack()
Label(top,text='username').place(x=30,y=50)
Entry(top).place(x=100,y=50)
```

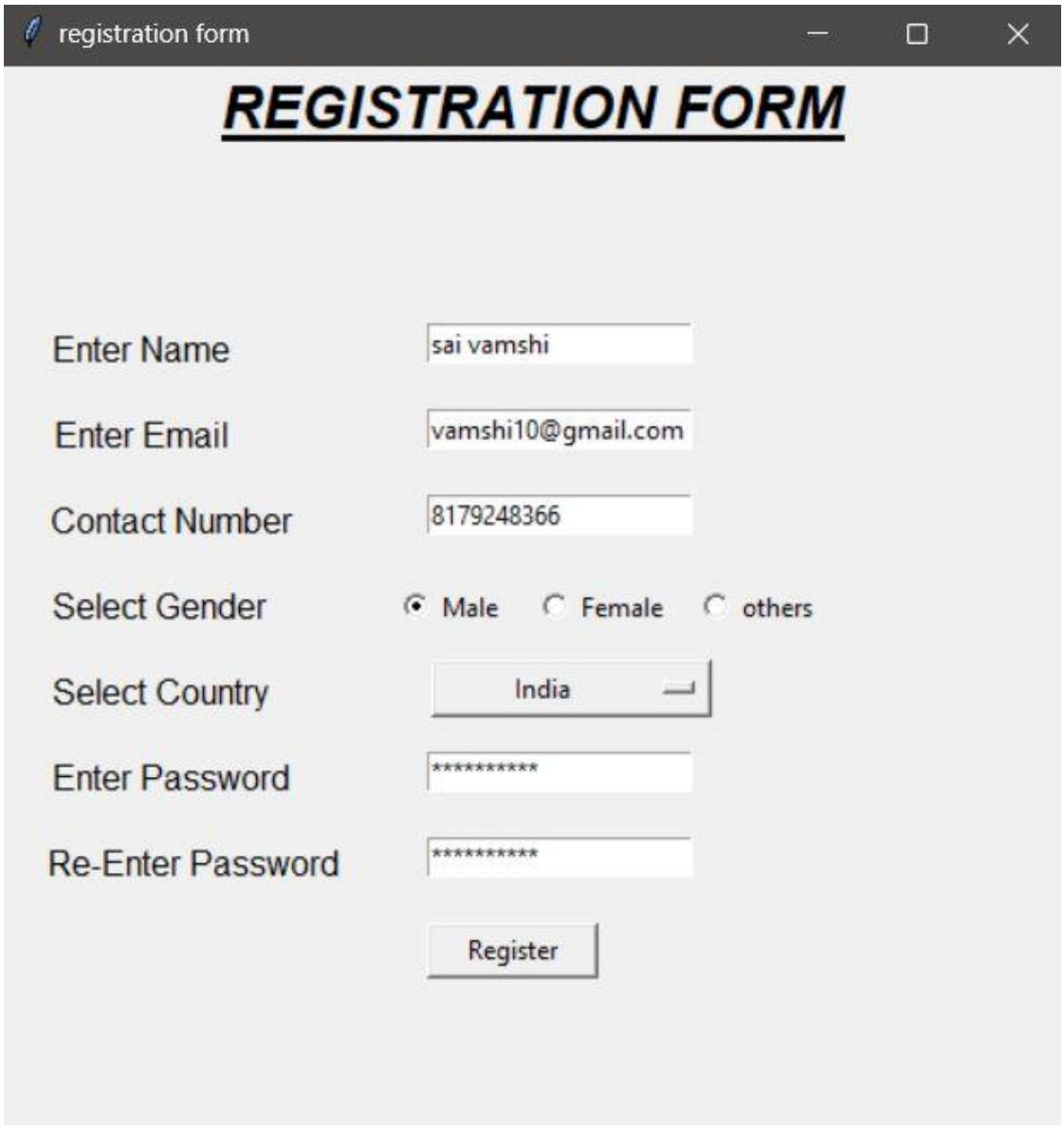
```
Label(top,text='password').place(x=30,y=90)
Entry(top,show='*').place(x=100,y=90)
Label(top,text="Gender").place(x=30,y=120)
vars = IntVar()
Radiobutton(top,text='male',padx=5,variable=vars, value=1).place(x=30,y=140)
Radiobutton(top,text='female',padx=10,variable=vars, value=2).place(x=100,y=140)
Label(top,text='subject').place(x=30,y=160)
lb=Listbox(top)
lb.insert(1,'python')
lb.insert(2,'c++')
lb.insert(3,'java')
lb.insert(4,'c')
lb.insert(5,'html')
lb.pack()
lb.place(x=50,y=200)
Button(top,text="submit").place(x=30,y=370)
top.mainloop()
```

OUT PUT :-



Example 3: write a program to build Registration page using tkinter?

```
import tkinter as tk
from tkinter import *
base = tk.Tk()
base.geometry("500x500")
base.title("registration form")
lb0= tk.Label(base,text="REGISTRATION FORM")
lb0.config(font=('callibri',20,'italic bold underline'))
lb0.pack()
lb1= Label(base, text="Enter Name", width=10, font=("arial",12))
lb1.place(x=19, y=120)
en1= Entry(base)
en1.place(x=200, y=120)
lb3= Label(base, text="Enter Email", width=10, font=("arial",12))
lb3.place(x=19, y=160)
en3= Entry(base)
en3.place(x=200, y=160)
lb4= Label(base, text="Contact Number", width=13,font=("arial",12))
lb4.place(x=19, y=200)
en4= Entry(base)
en4.place(x=200, y=200)
lb5= Label(base, text="Select Gender", width=15, font=("arial",12))
lb5.place(x=5, y=240)
vars = IntVar()
Radiobutton(base, text="Male", padx=5,variable=vars, value=1).place(x=180, y=240)
Radiobutton(base, text="Female", padx =10,variable=vars, value=2).place(x=240,y=240)
Radiobutton(base, text="others", padx=15, variable=vars, value=3).place(x=310,y=240)
list_of_cntry = ("United States", "India", "Nepal", "Germany")
cv = StringVar()
drplist= OptionMenu(base, cv, *list_of_cntry)
drplist.config(width=15)
lb2= Label(base, text="Select Country", width=13,font=("arial",12))
lb2.place(x=14,y=280)
drplist.place(x=200, y=275)
lb6= Label(base, text="Enter Password", width=13,font=("arial",12))
lb6.place(x=19, y=320)
en6= Entry(base, show='n')
en6.place(x=200, y=320)
lb7= Label(base, text="Re-Enter Password", width=15,font=("arial",12))
lb7.place(x=21, y=360)
en7 =Entry(base, show='n')
en7.place(x=200, y=360)
Button(base, text="Register", width=10).place(x=200,y=400)
base.mainloop()
```




### 5.5. Date & Time:

DateTime module is provided in Python to work with dates and times. In python DateTime, is an inbuilt module rather than being a primitive data type, We just have to import the module mentioned above to work with dates as date object.

There are several classes in the datetime python module of python which help to deal with the date & time. Moreover, there are so many functions of these classes from which we can extract the date and time.

## How to Use Date and Datetime Class?

Firstly, you need to import the python datetime module of python using the following line of code, so that you can use its inbuilt classes to get current date & time etc.

```
import datetime
```

1. Now, we can use the date class of the datetime module. This class helps to convert the numeric date attributes to date format in the form of YYYY-MM-DD. This class accepts 3 attributes Year, Month & Day in the following order (Year, Month, Day).

**Syntax**

```
        import datetime

        var1 = datetime.date(YYYY, MM, DD)

# This will convert the numeral date to the date object
```

**Example**

```
        import datetime

        userdate = datetime.date(2021, 10, 20)

        print('userdate: ', userdate)
        print('type of userdate: ', type(userdate))
```

**Output:**

```
success
    userdate:  2021-10-20
    type of userdate:  <class 'datetime.date'>
```

**Explanation of code**

- In the above code we imported the python datetime library.
    - We then saved the date time object into the variable.
    - Then we print the variable and also the type of variable whose output is down below.
- Now, we can use the datetime python class of the datetime module.
    - This class helps to convert the numeric date & time attributes to date-time format in the form of YYYY-MM-DD hr:min:s:ms.

This class accepts 7 attributes Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds in the following order (Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds).

**Syntax**

```
        import datetime

        var1 = datetime.datetime(YYYY, MM, DD, hr, min, s, ms)

        # This will convert the numeral date to the datetime object in the form
of
           YYYY-MM-DD hr:min:s:ms.
```

**Example**

```
import datetime

userdatetime = datetime.datetime(2019, 5, 10, 17, 30, 20, 154236)

print('userdatetime: ', userdatetime)
print('type of userdatetime: ', type(userdatetime))
```

**Output:**

```
success
    userdatetime:  2019-05-10 14:30:20.154236
    type of userdatetime:  <class 'datetime.datetime'>
```

**Explanation of code**

- In the above code we imported the datetime library.
- We then saved the date time object into the variable.
- Then we print the variable and also the type of variable whose output is down below.

Example :Write a python program on time and date modules.

- import datetime
- x = datetime.datetime.now()
- print(x.strftime("%a"))
- x = datetime.datetime.now()
- print(x.strftime("%A"))
- x = datetime.datetime.now()
- print(x.strftime("%w"))
- x = datetime.datetime.now()
- print(x.strftime("%d"))
- x = datetime.datetime.now()
- print(x.strftime("%c"))
- x = datetime.datetime.now()
- print(x.strftime("%x"))
- x = datetime.datetime.now()
- print(x.strftime("%j"))
-
- OUTPUT:-
- Tue
- Tuesday
- 2
- 20
- Tue Dec 20 21:14:45 2022
- 12/20/22
- 354

# Classes of DateTime Python Module:

## Date

Now, we can use the date class of the datetime module.

This class helps to convert the numeric date attributes to date format in the form of YYYY-MM-DD.

This class accepts 3 attributes Year, Month & Day in the following order (Year, Month, Day).

### Syntax

```
import datetime

var1 = datetime.date(YYYY, MM, DD)

# This will convert the numeral date to the date object
```

### Example

```
import datetime

userdate = datetime.date(2021, 9, 15)

print('userdate: ', userdate)
print('type of userdate: ', type(userdate))
```

### Output

```
userdate:  2021-09-15
type of userdate:  <class 'datetime.date'>
```

### Explanation of code

- In the above code we imported the datetime library.
- We then saved the date time object into the variable.
- Then we print the variable and also the type of variable whose output is down below.

As we can see that we have changed the numeral data to the datetime object and the same can be verified by the type of object printed below.

## Time

Now, we can use the time class of the python datetime module.

Time class returns the local time of the area where user is present.

Time class accepts 5 attributes hour, minute, second, microsecond & fold in the following order (hour, minute, second, microsecond, fold).

All the above mention attributes are optional, but the initial value of all the attributes are 0.

## Points to Remember About Time Class

- If we don't pass any attribute in time class then it will return the default time which is 00:00:00.
- By specifying particular attribute in the time class then it will set its value and give the time as per the user requirement.

**Syntax**

```
import datetime

# returns the time with all it's value as 0 like 00:00:00
var1 = datetime.time()

# returns the time with the value which are specified by the user in the attributes
var2 = datetime.time(hour=?, minute=?, second=?, microsecond=?)
```

**Example**

```
import datetime

time1 = datetime.time()
print('without passing any attribute: ', time1)

time2 = datetime.time(hour=12, minute=55, second=50)
print('by passing hour, minute and second attributes: ', time2)

time1 = time1.replace(hour=17)
print('by replacing the hour attribute in time1: ', time1)

time2 = time2.replace(minute=00)
print('by replacing the minute attribute in time2: ', time2)
```

**Output**

```
without passing any attribute:  00:00:00
by passing hour, minute and second attributes:  12:55:50
by replacing the hour attribute in time1:  17:00:00
by replacing the minute attribute in time2:  12:00:50
```

**Explanation of Code**

- In the above code we imported the python datetime library.
- We then saved two different type of time object into the variable (one having no attributes and other with some attributes).
- Then we used replace() to change the value of above generated time.
- Then we print the variable to get different times according to the attributes we passed in the time class.

# Datetime

Now, we can use the datetime python class of the datetime module. This class helps to convert the numeric date & time attributes to date-time format in the form of YYYY-MM-DD hr:min:s:ms.

This class accepts 7 attributes Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds in the following order (Year, Month, Day, Hour, Minutes, Seconds, MilliSeconds).

**Syntax**

```
    import datetime

    var1 = datetime.datetime(YYYY, MM, DD, hr, min, s, ms)

    # This will convert the numeral date to the datetime object in the form of
       YYYY-MM-DD hr:min:s:ms.
```

**Example**

```
import datetime

userdatetime = datetime.datetime(2021, 9, 15, 20, 55, 20, 562789)

print('userdatetime: ', userdatetime)
print('type of userdatetime: ', type(userdatetime))
```

**Output**

```
userdatetime:  2021-09-15 20:55:20.562789
type of userdatetime:  <class 'datetime.datetime'>
```