DIGITAL UNIVERSITY KERALA

Curating a responsible digital world

Advanced Programming

Python

OOP Concepts

Following are some fundamental concepts of object-oriented

Programming:

- **Objects** − Objects represent an entity and the basic building block.
- **Class** − Class is the blueprint of an object, it is a way to bind the data and member functions.
- **Abstraction** − Abstraction represents the behavior of a real-world entity.
- **Encapsulation** − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- **Inheritance** − Inheritance is the mechanism of making new classes from existing ones.
- **Polymorphism** − It defines the mechanism to exist in different forms.

```
class Employee:
    def __init__(self, emp_id, name, department, salary):
        """
        Constructor to initialize an Employee object.
        Parameters:
        emp_id (int): Employee ID.
        name (str): Name of the employee.
        department (str): Department where the employee works.
        salary (float): Salary of the employee.
        """
        self.emp_id = emp_id
        self.name = name
        self.department = department
        self.salary = salary

    def display_employee_info(self):
        """Displays the employee's information."""
        print(f"Employee ID: {self.emp_id}")
        print(f"Name: {self.name}")
        print(f"Department: {self.department}")
        print(f"Salary: ${self.salary:.2f}")

    def give_raise(self, amount):
        """
        Increases the employee's salary by a specified amount.
        Parameters:
        amount (float): The amount by which to increase the salary.
        """
        self.salary += amount
        print(f"Salary increased by ${amount:.2f}. New
salary is ${self.salary:.2f}")
```

```
# Creating an instance of the Employee class
employee1 = Employee(emp_id=101, name="John
Doe", department="Engineering", salary=75000)

# Displaying employee information
employee1.display_employee_info()

# Giving the employee a raise
employee1.give_raise(5000)

# Displaying updated employee information
employee1.display_employee_info()
```

OUTPUT

```
Employee ID: 101
Name: John Doe
Department: Engineering
Salary: $75000.00
Salary increased by $5000.00. New salary is
$80000.00
Employee ID: 101
Name: John Doe
Department: Engineering
Salary: $80000.00
```

What is Inheritance?

The method of inheriting the properties of a parent class into a child class is known as inheritance. It is an OOP concept. Following are the benefits of inheritance.

Code reusability- no need to write the same code again and again; we can just inherit the properties we need in a child class.

- If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.
- When we add the __init__() function in a parent class, the child class will no longer be able to inherit the parent class's __init__() function.

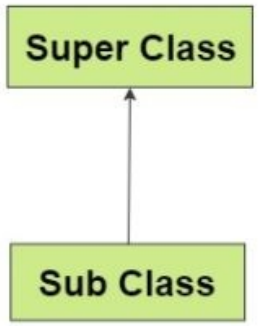
```
1  class Parent():
2          def first(self):
3              print('first function')
4
5  class Child(Parent):
6          def second(self):
7              print('second function')
8
9  ob = Child()
10 ob.first()
11 ob.second()
```

Output:

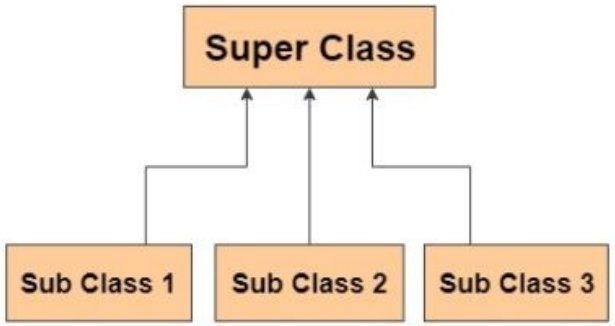
```
first function
second function
```

Type of Inheritance

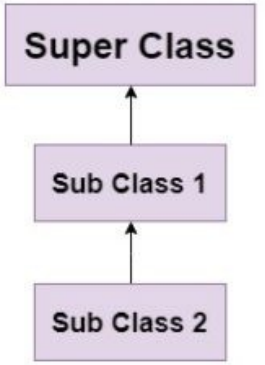
Single Inheritance



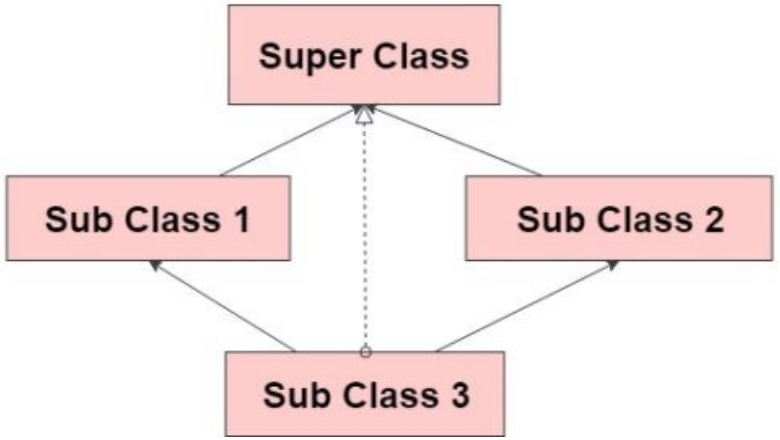
Hierarchial Inheritance



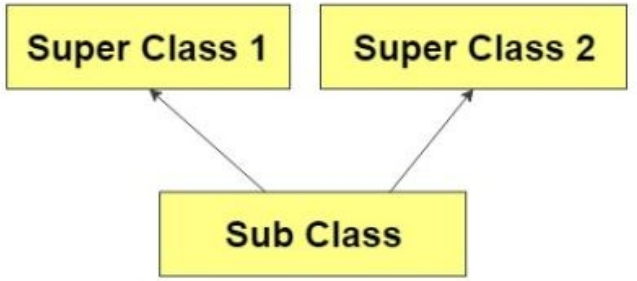
MultiLevel Inheritance



Hybrid Inheritance



Multiple Inhertance



Multiple Inheritance

When a class is derived from more than one base class, it is called multiple inheritance

Example: to override a method of one parent class only.

```
# Defining parent class 1
class Parent1():

    # Parent's show method
    def show(self):
        print("Inside Parent1")

# Defining Parent class 2
class Parent2():

    # Parent's show method
    def display(self):
        print("Inside Parent2")


# Defining child class
class Child(Parent1, Parent2):

    # Child's show method
    def show(self):
        print("Inside Child")


# Driver's code
obj = Child()

obj.show()
obj.display()
```

Output:

```
Inside Child
Inside Parent2
```

Multilevel Inheritance

A child class becomes a parent class for another child class

Example: to override only one method of one of its parent classes.

```
class Parent():

    # Parent's show method
    def display(self):
        print("Inside Parent")


# Inherited or Sub class (Note Parent in bracket)
class Child(Parent):

    # Child's show method
    def show(self):
        print("Inside Child")

# Inherited or Sub class (Note Child in bracket)
class GrandChild(Child):

    # Child's show method
    def show(self):
        print("Inside GrandChild")

# Driver code
g = GrandChild()
g.show()
g.display()
```

Output:

```
Inside GrandChild
Inside Parent
```

Abstraction vs Encapsulation

- Abstraction is the process of hiding unwanted data; encapsulation is the process of hiding data with the goal of protecting that information.
- While abstraction is achieved using abstract classes or interfaces, encapsulation is implemented using an access modifier.

For example, let's say we have a class called **BankAccount** that has attributes such as **balance** and **account_number** and methods such as **deposit()** and **withdraw()**. We want to ensure that the balance can only be modified through the **deposit()** and **withdraw()** methods and not directly.

Exampl
e

```
class BankAccount:

def __init__(self, balance,
account_number):
  self.__balance = balance
  self.__account_number =
account_number

def deposit(self, amount):
  self.__balance += amount

def withdraw(self, amount):
  if self.__balance >= amount:
    self.__balance -= amount
  else:
    print("Insufficient funds")

def get_balance(self):
   return self.__balance
```

In this example, the balance and account_number attributes are prefixed with **two underscores**, which makes them private. This means that they can only be accessed within the BankAccount class and not outside of it. The user can only modify the balance through the deposit() and withdraw() methods, ensuring that the data is secure.

Abstract Class

- An abstract class is a **template definition of methods and variables** in a specific class.
- It can have abstract and non-abstract methods (method with the definition/body).
- It **cannot be instantiated** directly.
- The methods and properties defined (but not implemented) in an abstract class are called **abstract methods** and **abstract properties**. All abstract methods and properties need to be implemented in a child class in order to be able to create objects from it.

Working on Python Abstract classes

- By default, Python does not provide abstract classes.
- Python comes with a module that provides the base for defining Abstract Base classes(ABC), and that module name is ABC.
- ABC works by decorating methods of the base class as an abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the keyword **@abstractmethod**.

Abstract Class - Reference links:

Example in Python: https://www.geeksforgeeks.org/abstract-classes-in-python

Abc – Abstract Base Classes:
https://docs.python.org/3/library/abc.html

Dunder or magic methods in Python

- Dunder or Magic methods in Python are the methods having two prefixes and suffix underscores in the method name.
- Dunder here means “Double Under (Underscores).”
- These are commonly used for operator overloading.

List of Python Magic Methods

To get the list of magic functions in Python, open cmd or terminal, type python3 to go to the Python console, and type:

```
>>> dir(int)
```

Initialization and Construction

- ***__new__:*** *To get called in an object's instantiation.*
- ***__init__:*** *To get called by the __new__ method.*
- ***__del__:*** *It is the destructor.*

Numeric magic methods

- ***__trunc__(self):*** *Implements behavior for math.trunc()*
- ***__ceil__(self):*** *Implements behavior for math.ceil()*
- ***__floor__(self):*** *Implements behavior for math.floor()*
- ***__round__(self, n):*** *Implements behavior for the built-in round()*

String Magic Methods

- ***__str__(self):*** *Defines behavior for when str() is called on an instance of your class.*
- ***__repr__(self):*** ***T****o get called by built-int repr() method to return a machine readable representation of a type.*
- ***__unicode__(self):*** *This method to return an unicode string of a type.*
- ***__dir__(self):*** *This method to return a list of attributes of a class.*
- ***__sizeof__(self):*** *It return the size of the object.*

Arithmetic operators

- ***__add__(self, other):*** *Implements behavior for math.trunc()*
- ***__sub__(self, other):*** *Implements behavior for math.ceil()*
- ***__mul__(self, other):*** *Implements behavior for math.floor()*
- ***__floordiv__(self, other):*** *Implements behavior for the built-in round()*
- ***__div__(self, other):*** *Implements behavior for inversion using the ~ operator.*
- ***….etc***

Comparison magic methods

- ***__eq__(self, other):*** *Defines behavior for the equality operator, ==.*
- ***__ne__(self, other):*** *Defines behavior for the inequality operator, !=.*
- ***__lt__(self, other):*** *Defines behavior for the less-than operator, <.*
- ***__gt__(self, other):*** *Defines behavior for the greater-than operator, >.*

Dunder methods are special functions that Python invokes at certain times during a program's life cycle.

One of the most significant benefits of using Python's magic methods is that they provide a **simple way to make objects behave like built-in types**.

Ref. Note:

https://levelup.gitconnected.com/what-are-python-magic-methods-667567525e9

Examples :

https://builtin.com/data-science/dunder-methods-python

https://www.pythontutorial.net/python-oop/python-__new__

https://www.section.io/engineering-education/dunder-methods-python/

UML (Unified Modeling Language)

- UML is a pictorial language used to make software blueprints
- UML can be described as a general-purpose visual modeling language to visualize, specify, construct, and document software systems.

The building blocks of UML can be defined as:

A. Things

B. Relationships

C. Diagrams

Things

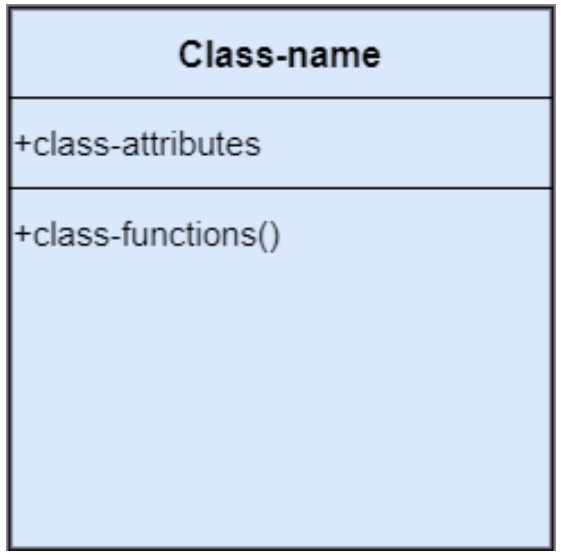
‘Things’ are the most essential building blocks of UML. Things can be:

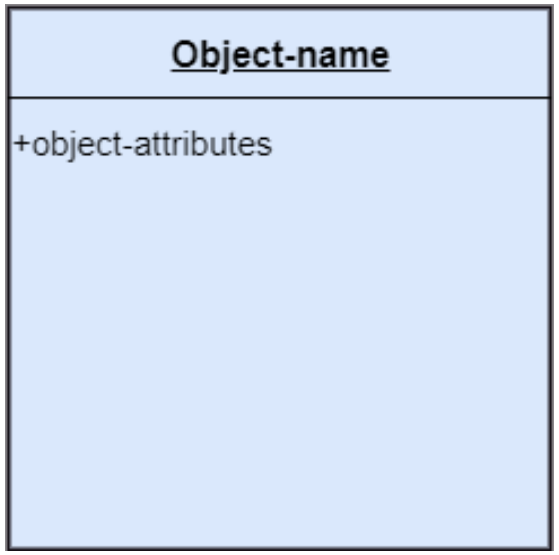
1. Structural
2. Behavioral
3. Grouping
4. Annotational

1. Structural Things

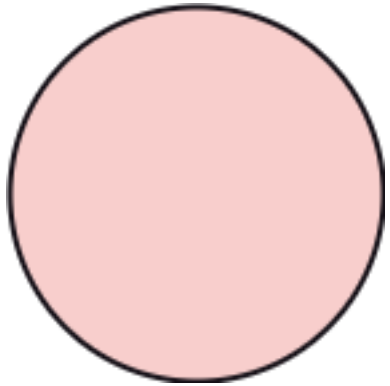
Structural things define the static part of the model. They represent the physical and conceptual elements.

Class - Class represents a set of objects having similar responsibilities.
Object - An individual that describes the behavior and the functions of a system. The notation of the object is similar to that of the class; the only difference is that the object name is always underlined, and its notation is given below
Interface: A set of operations that describes the functionality of a class, which is implemented whenever an interface is implemented.





Interface



Use case

The use case is the core concept of object-oriented modeling. It portrays a set of actions executed by a system to achieve the goal.

Actor

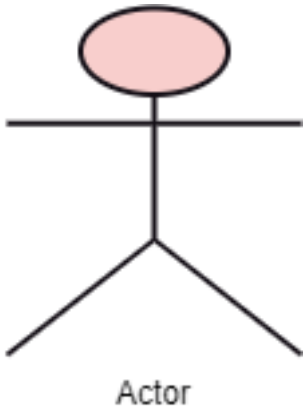
It comes under the use case diagrams. It is an object that interacts with the system, for example, a user.

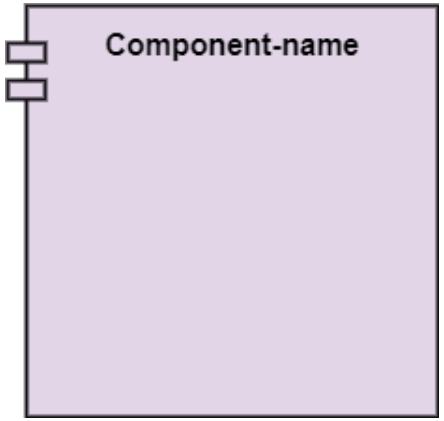
Component

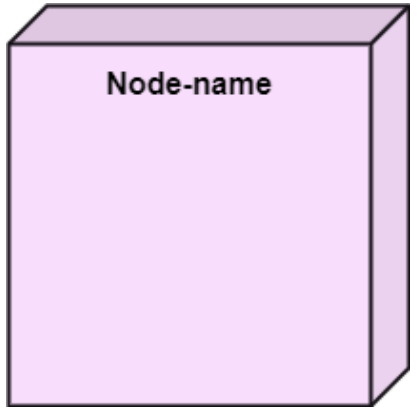
It represents the physical part of the system.

Node

A physical element that exists at run time.



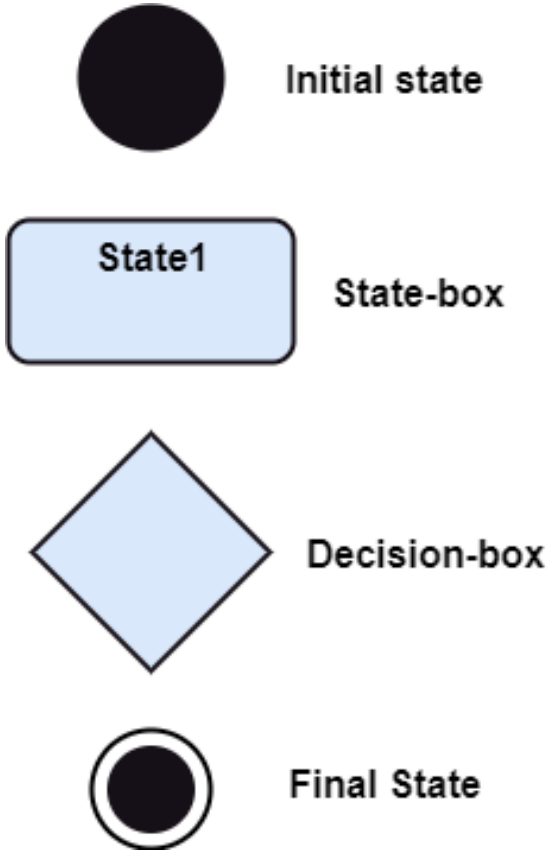


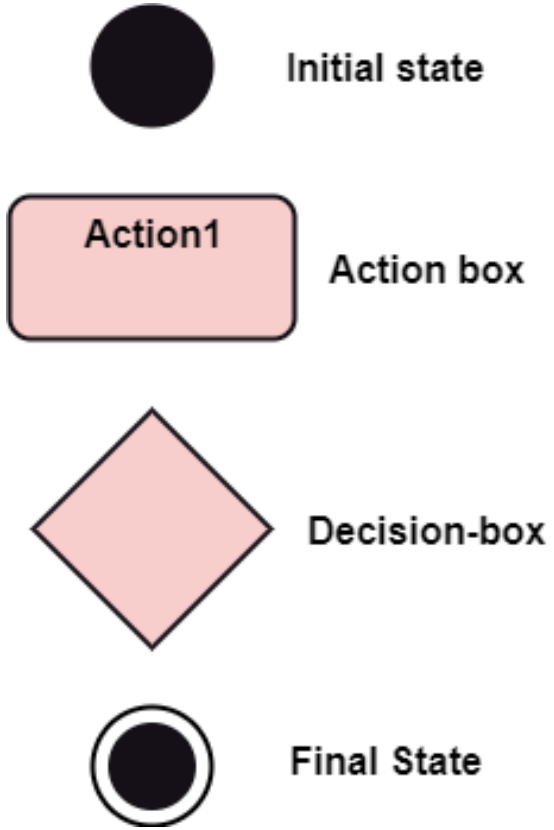


2. Behavioral Things

They are the verbs that encompass the dynamic parts of a model. It depicts the behavior of a system. They involve state machine, activity diagram, interaction diagram, grouping things, and annotation things.

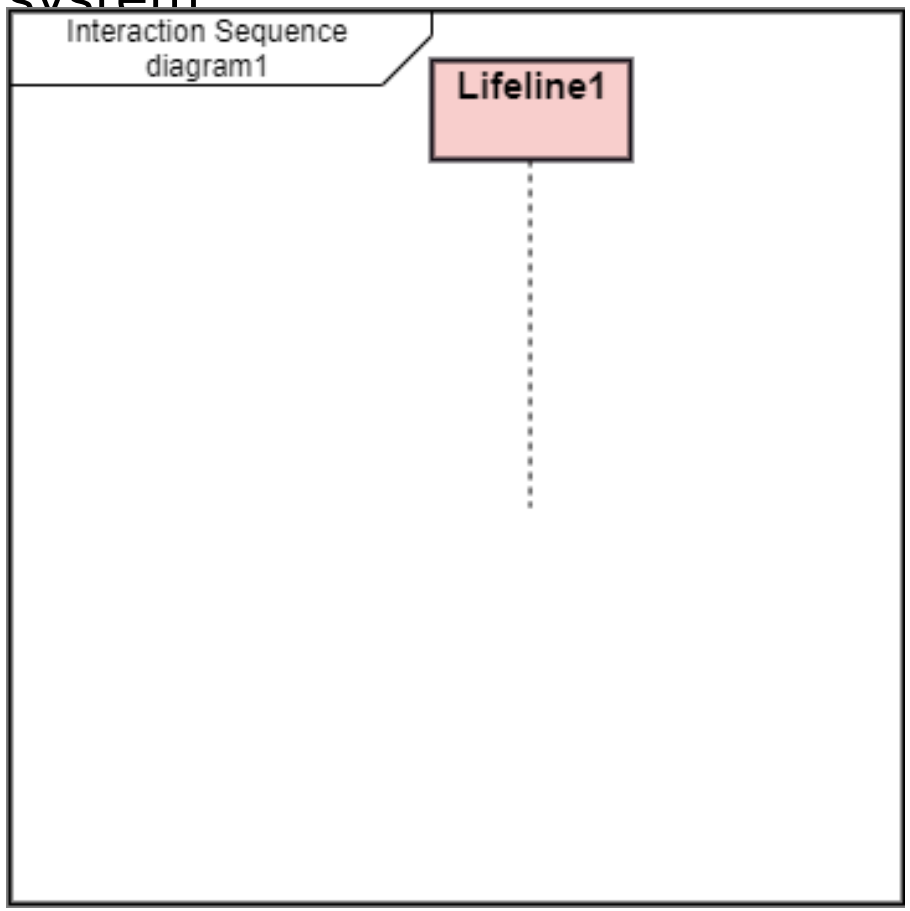
State Machine: It defines a sequence of states that an entity goes through in the software development lifecycle. It keeps a record of several distinct states of a system component.





Activity Diagram: It portrays all the activities accomplished by different entities of a system. It is represented the **same as that of a state machine diagram**.

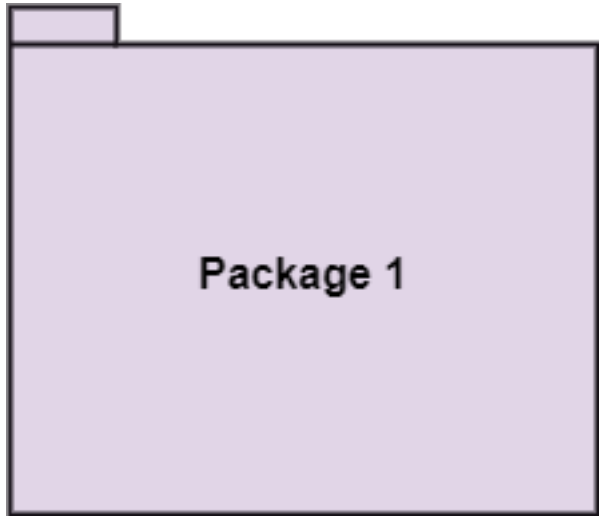
Interaction Diagram: It is used to envision the flow of messages between several components in a system.



Grouping Things

It is a method that together binds the elements of the UML model. In UML, the package is the only thing, which is used for grouping.

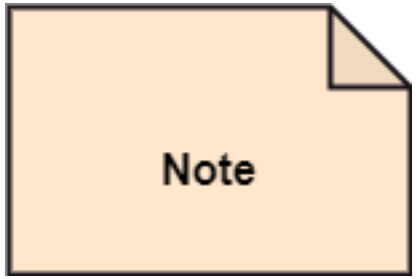
Package: Package is the only thing that is available for grouping behavioral and structural things.



3. Annotation Things

It is a mechanism that captures the remarks, descriptions, and comments of UML model elements. In UML, a note is the only Annotational thing.

Note: It is used to attach the constraints, comments, and rules to the elements of the model. It is a kind of yellow sticky note.



B. Relationships

It illustrates the meaningful connections between things. It shows the association between the entities and defines the functionality of an application.

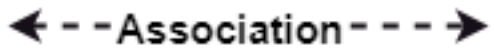
There are four types of relationships given below:

Dependency: Dependency is a kind of relationship in which a change in target element affects the source element, or simply we can say the source element is dependent on the target element.

It is denoted by a dotted line followed by an arrow at one side as shown below,

Association: A set of links that associates the entities to the UML model. It tells how many elements are actually taking part in forming that relationship.

- It is denoted by a dotted line with arrowheads on both sides to describe the relationship with the element on both sides.



Generalization: It portrays the relationship between a general thing (a parent class or superclass) and a specific kind of that thing (a child class or subclass). It is used to describe the concept of inheritance.

It is denoted by a straight line followed by an empty arrowhead at one side.

Realization: It is a semantic relationship between two things, where one defines the behavior to be carried out, and the other implements the mentioned behavior. It exists in interfaces.
It is denoted by a dotted line with an empty arrowhead at one side.

C. Diagrams

The diagrams are the graphical implementation of the models that incorporate symbols and text. Each symbol has a different meaning in the context of the UML diagram

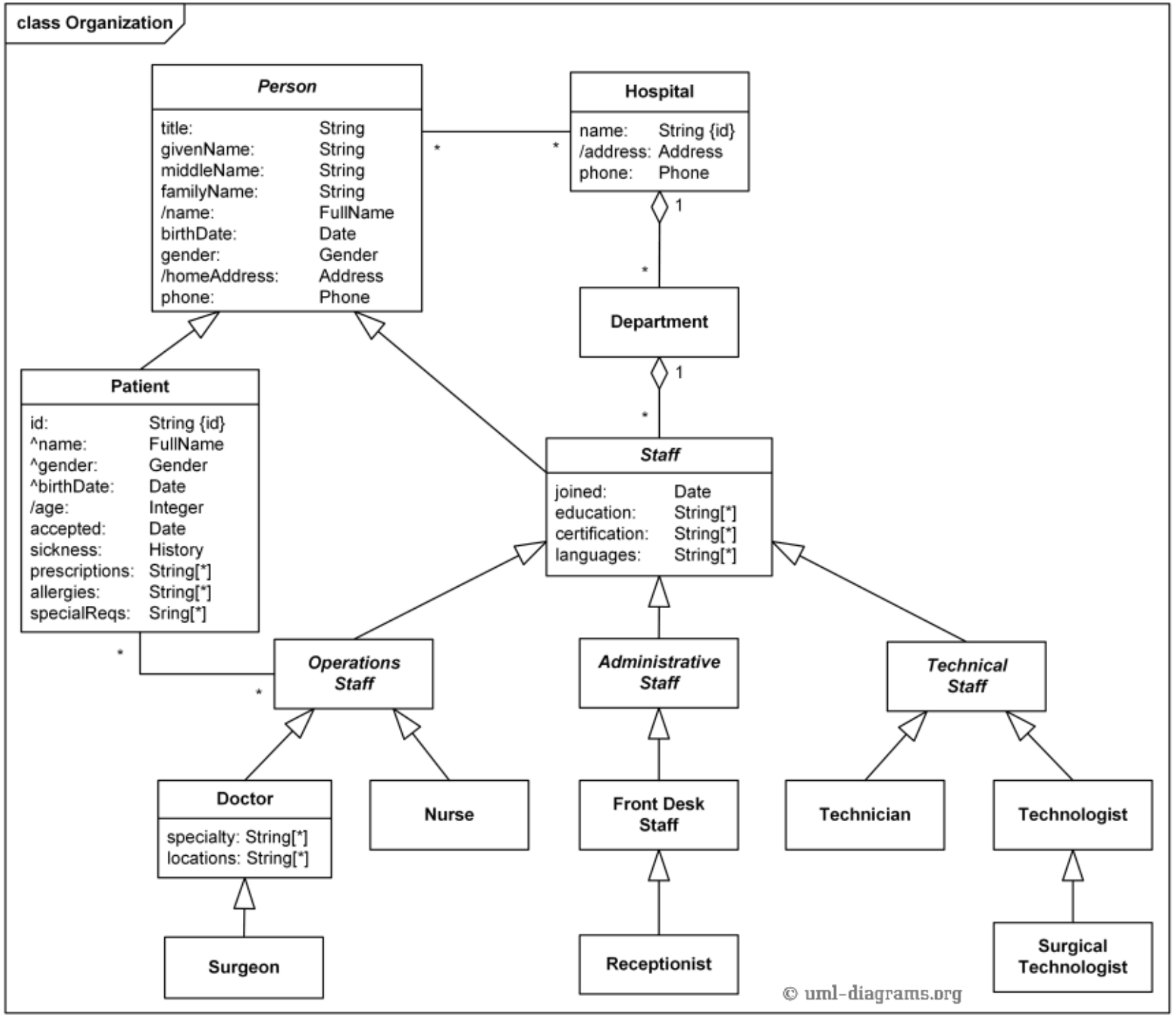
Class diagra

1.It analyses and designs a static view of an application.

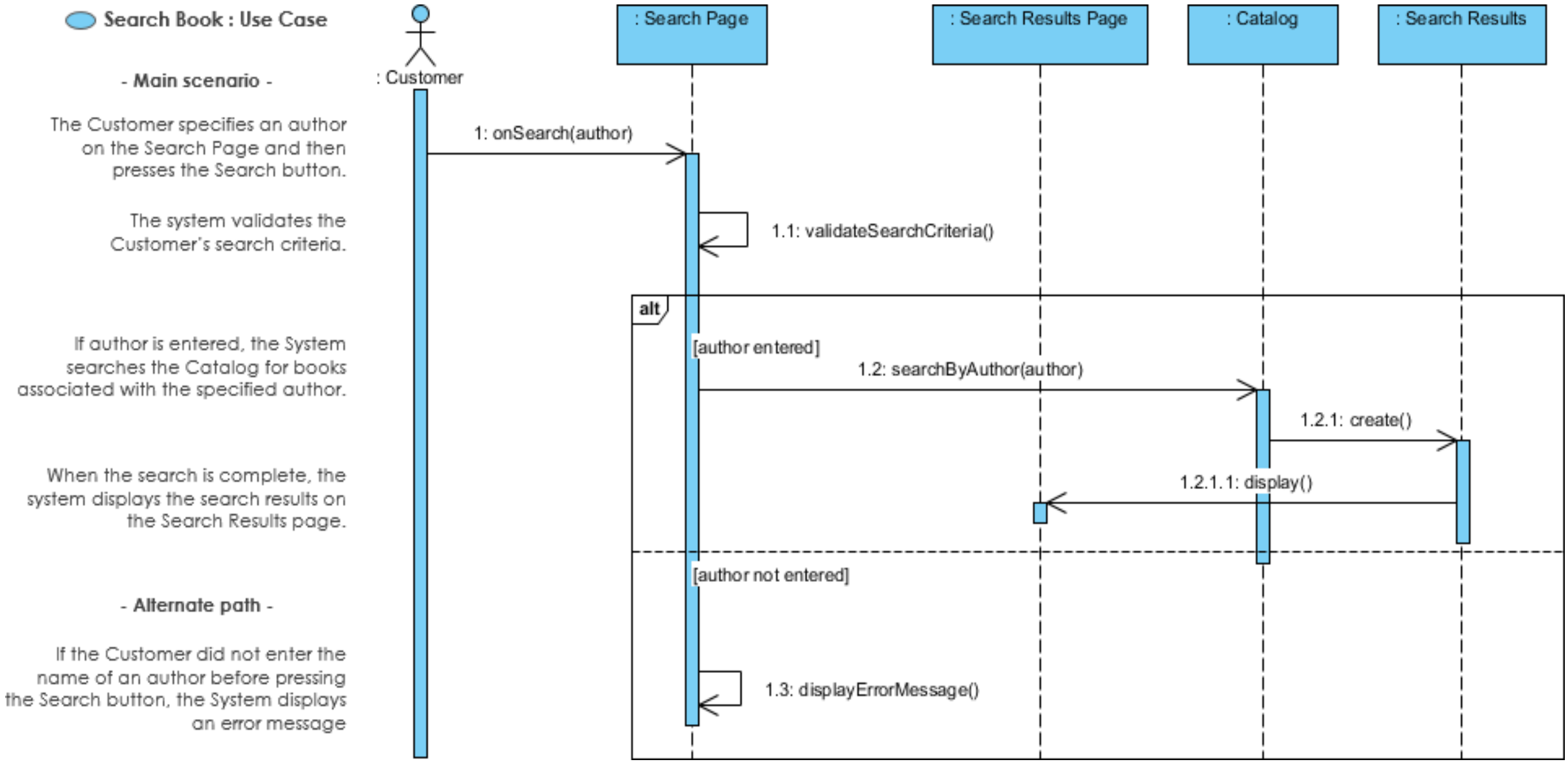
2.It describes the major responsibilities of a system.

3.It is a base for component and deployment diagrams.

4.It incorporates forward and reverse engineering.



Sequence diagram



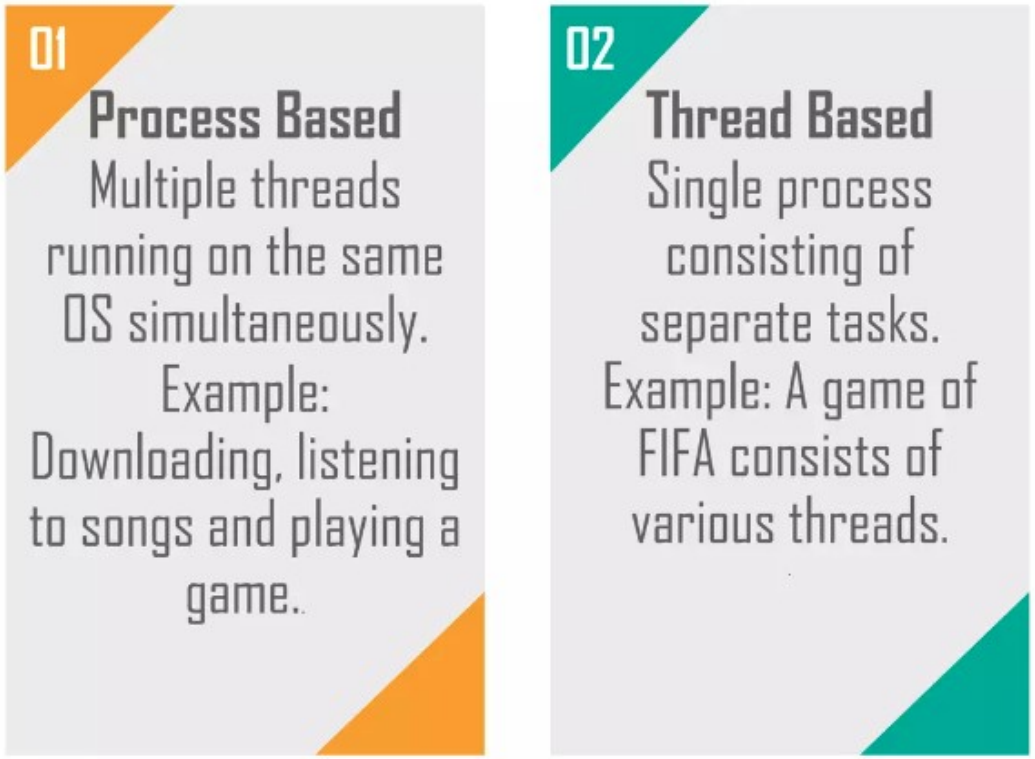
ttps://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

Multitasking in Python

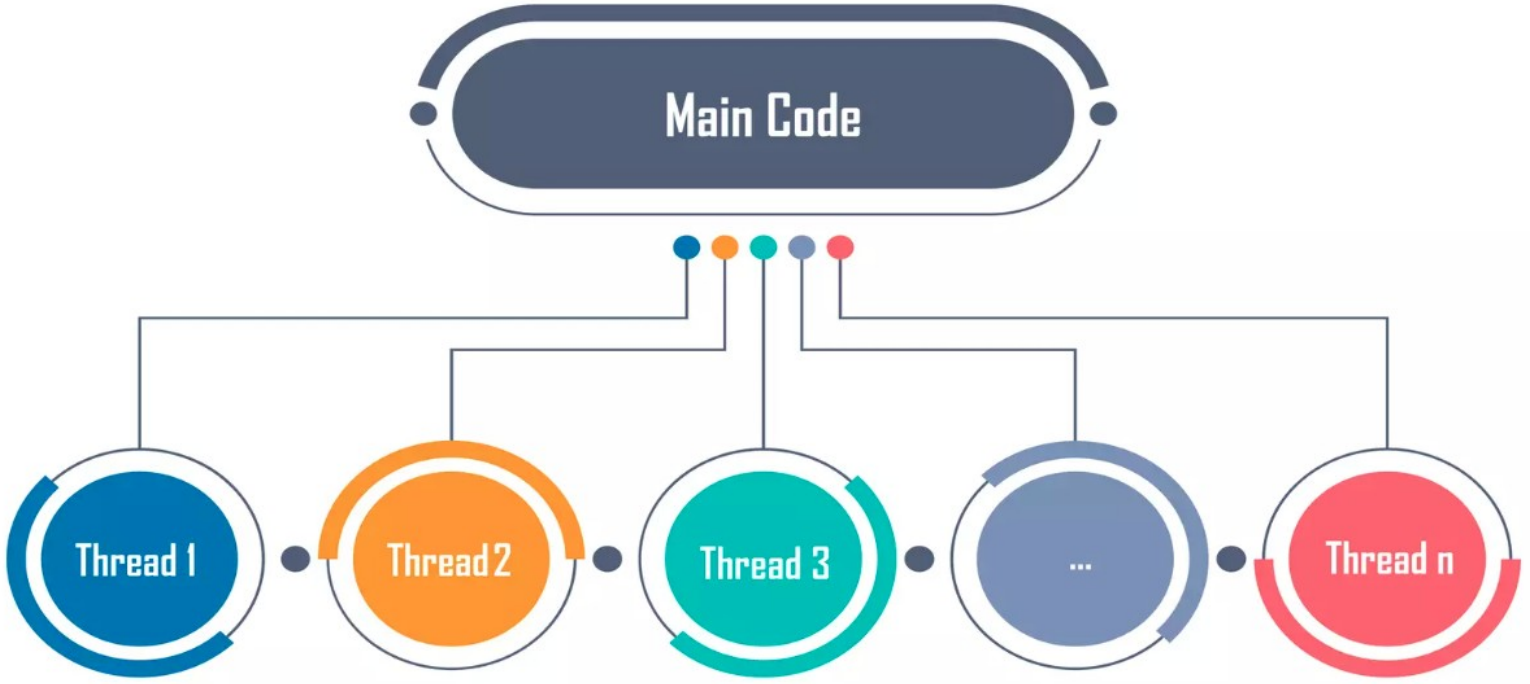
Multitasking refers to the ability of an operating system to perform different tasks at the same time

Type of Multitasking

There are two types of multitasking in an OS:



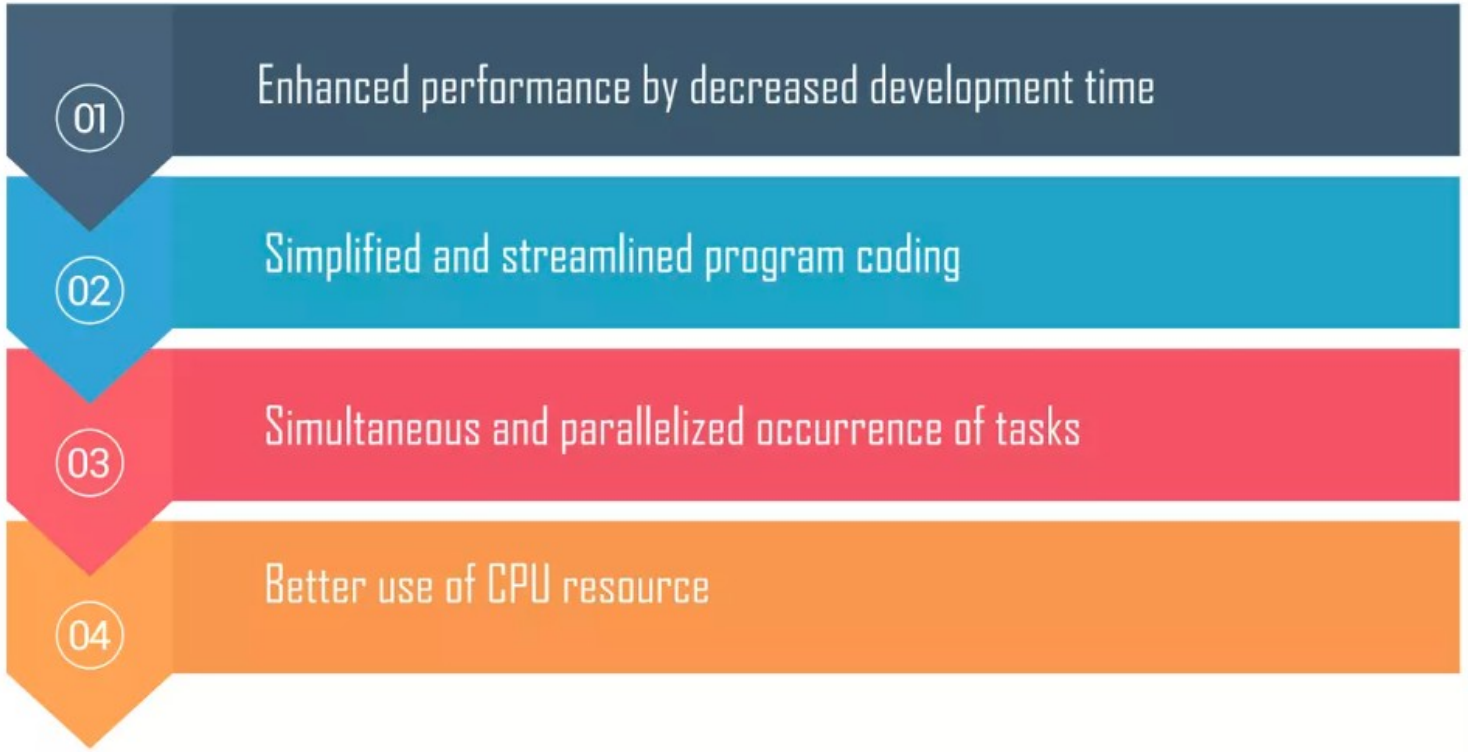
What is Multithreading



How to achieve Multithreading in Python?

By importing the ***threading*** module :
import threading
*From threading import **

Advantages of Multithreading



Thread class methods

Methods	Description
start()	A start() method is used to initiate the activity of a thread. And it calls only once for each thread so that the execution of the thread can begin.
run()	A run() method is used to define a thread's activity and can be overridden by a class that extends the threads class.
join()	A join() method is used to block the execution of another code until the thread terminates.

Declaration of the thread parameters: It contains the target function, argument, and **kwargs** as the parameter in the **Thread()** class.

Target: It defines the function name that is executed by the thread.

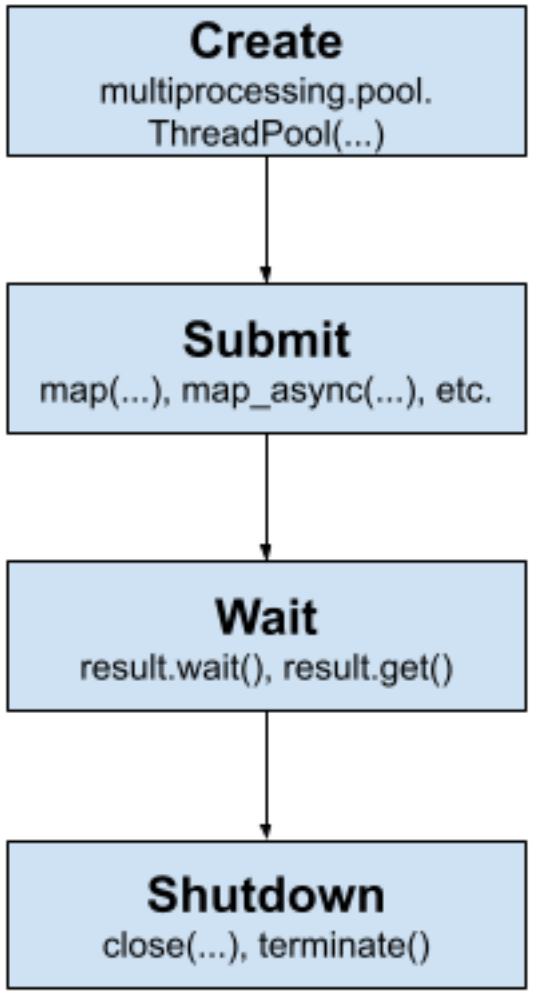
Args: It defines the arguments passed to the target function name.

Thread Pools

- It is a programming pattern for automatically managing a pool of worker threads.
- The pool is responsible for a fixed number of threads.
  - ✓ It controls when the threads are created, such as just-in-time when they are needed.
  - ✓ It also controls what threads should do when they are not being used, such as making them wait without consuming computational resources.
- Each thread in the pool is called a worker or a worker thread

- Worker threads are designed to be re-used once the task is completed and provide protection against the unexpected failure of the task, such as raising an exception, without impacting the worker thread itself.
- This is unlike a single thread that is configured for the single execution of one specific task.
- The pool may provide some facility to configure the worker threads, such as running an initialization function and naming each worker thread using a specific naming convention.

Life-Cycle of the ThreadPool




```
# example of handling an exception raised within
task
from time import sleep
from multiprocessing.pool import ThreadPool

# task executed in a worker thread
def task():
    # block for a moment
    sleep(1)
    try:
        raise Exception('Something bad happened!')
    except Exception:
        return 'Unable to get the result'
    return 'Never gets here'

# protect the entry point
if __name__ == '__main__':
    # create a thread pool
    with ThreadPool() as pool:
        # issue a task
        result = pool.apply_async(task)
        # get the result
        value = result.get()
        # report the result
```

```
import concurrent.futures

……

with
concurrent.futures.ThreadPoolExecutor(max_workers=num_threads)
as executor:
# Submit each function to the thread pool
futures = [executor.submit(func) for func in functions]

# Wait for all threads to complete
concurrent.futures.wait(futures)
```

The **ThreadPool** and **ThreadPoolExecutor** classes are very similar. They are both thread pools that provide a collection of workers for executing ad hoc tasks.