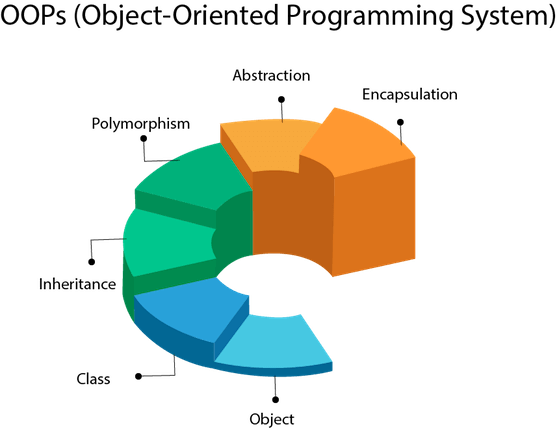
# **Encapsulation, Inheritance, Polymorphism, and Data Abstraction in Object-Oriented Programming**



Object-oriented programming refers to the concept in high-level languages such as Python that uses Objects and classes in their implementations. OOP has four major building blocks which are,

i) Polymorphism,

ii) Encapsulation,

iii) Abstraction, and

iv) Inheritance.

There are other programming paradigms such as Procedural programming in which codes are written sequentially. Python and Java are multi-paradigm high-level programming languages that means they support both OOP and procedural programming. A programmer decides on the paradigm to use based on their expertise and the problems they’re trying to solve. However, there is no doubt that OOP makes programming easier, faster, more dynamic, and secure. This is a major reason Python is one of the top most popular programming language in the world today

If you want to learn Python or any other object-oriented programming languages, then you must understand the Object-Oriented Programming paradigm which is a relatively easy concept to understand. Let’s take a look at them.

## **Object-oriented programming**

*Object-oriented programming (OOP)* is a programming paradigm that uses objects and their interactions to design applications and computer programs.

There are some basic programming concepts in OOP we need to be aware of:

* Inheritance
* Abstraction
* Polymorphism
* Encapsulation

1. ***inheritance*** is a way to form new classes using classes that have already been defined.
2. ***Abstraction*** is simplifying complex reality by modeling classes appropriate to the problem.
3. ***Polymorphism*** is the process of using an operator or function in different ways for different data input.
4. ***Encapsulation*** hides the implementation details of a class from other objects.

The main idea behind Object Oriented Programming is simplicity, code reusability, extendibility, and security. These are achieved through Encapsulation, abstraction, inheritance, and polymorphism.

### 

### **What is Inheritance?**

In Python, codes are written in objects or blocks if you are adopting OOP methodology. Objects can interact with one another by using the properties (attributes) of each block or extending the functionalities of a block through inheritance.

Inheritance ensures that codes are reused and shared. There are millions of Python libraries that a programmer can use and reuse through inheritance. The properties of a class can be inherited and extended by other classes or functions or methods.

**There are two types of classes.**

i) Parent or base class, and the other is the

ii) Child class which can inherit the properties of the parent class.

Inheritance is a major pillar in Object-Oriented programming. It is the mechanism by which classes in Python, and other OOP languages, inherit the attribute of other classes.

A parent class can share its attributes with a child class. An example of a parent class implementation is in DDL (Dynamic-link library). A DDL can contain different classes that can be used by other programs and functions.

A real-world example of inheritance is a mother and a child. The child may inherit attributes such as height, voice pattern, color. The mother can reproduce other children with the same attributes as well.

You can create a function/method or class called “Move Robot," which controls a robot to move. And you could create methods and functions in other programs that can inherit the " **Move Robot**” Class without rewriting the codes over and over again. You can also extend the same class by inheriting it and writing a few more codes to it that would instruct a robot to move and also run in some specific circumstances using **if** and **else** (conditional) statements. With inheritance, you can create multiple robots that would inherit the attributes of the parent class “**Move Robot,**" which ensures code reusability.

In summary, Inheritance is concerned with the relationship between classes and methods/functions, which is like a parent and a child class. A child can be born with some of the attributes of the parents. Inheritance ensures reusability of codes just the way multiple children can inherit the attributes of their parents.

When we want to create a function, method, or class, we look for a superclass that contains the code or some of the code we want to implement.

We can then derive our class from the existing one. In Python, we achieve this by inheriting the attributes of a class by calling up the class name.

In the world of Object-Oriented Programming (OOP), Inheritance refers to the mechanism or the capability of a class to derive or extend the properties from another class in the run. This property enables the derived class to acquire the properties or traits of the base class.

Inheritance is considered one of the most important aspects of OOP because it serves the feature of reusability, thus making the piece of code more reliable.

## 

## 

**Python Inheritance**

## **Benefits of Inheritance**

## Inheritance depicts relationships that resemble real-world scenarios.

## It provides the feature of re-usability which allows the user to add more features to the derived class without altering it.

## If a class Y inherits from class X, then automatically all the sub-classes of Y would inherit from class X.

## **Basic Terminologies of Inheritance**

1. ***Subclass/Derived class*:** It is a class that inherits the properties from another class (usually the base class).
2. ***Superclass/Base class*:** It is the class from which other subclasses are derived.
3. **A derived class** usually *derives/inherits/extends* the base class.

## **Python Inheritance Syntax**

**class SuperClassName:**

**Body of Super class**

**class DerivedClass\_Name(SuperClass):**

**Body of derived class**

## **Python Inheritance Example**

**#Step 1: Create a Base Class**

**class Father:**

**# The keyword 'self' is used to represent the instance of a class.**

**# By using the "self" keyword we access the attributes and methods of the class in python.**

**# The method "\_\_init\_\_" is called as a constructor in object oriented terminology.**

**# This method is called when an object is created from a class.**

**# it allows the class to initialize the attributes of the class.**

**def \_\_init\_\_(self, name, lastname):**

**self.name = name**

**self.lastname = lastname**

**def printname(self):**

**print(self.name, self.lastname)**

**# Use the Father class to create an object, and then execute the printname method:**

**x = Father("Migot", "Ndede")**

**x.printname()**

**Output: Migot Ndede**

# The subclass \_\_init\_\_() function overrides the inheritance of the base class \_\_init\_\_() function.

class Son(Father):

def \_\_init\_\_(self, name, lastname):

Father.\_\_init\_\_(self, name, lastname)

class Daughter(Father):

def \_\_init\_\_(self, name, lastname):

Father.\_\_init\_\_(self, name, lastname)

x = Son("Son:", "Christopher")

y = Son("Son:", "Calvin")

z = Daughter("Daughter:", "Clara")

x.printname()

print()

y.printname()

print()

z.printname()

print()

OUTPUT:

Son: Christopher

Son: Calvin

Daughter: Clara

## **Use of super() function**

## By using the **super()** function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

## **class Father:**

## **def \_\_init\_\_(self, name, lastname):**

## **self.name = name**

## **self.lastname = lastname**

## **def printname(self):**

## **print(self.name, self.lastname)**

## **class Son(Father):**

## **def \_\_init\_\_(self, name, lastname):**

## **super().\_\_init\_\_(name, lastname)**

## **class Daughter(Father):**

## **def \_\_init\_\_(self, name, lastname):**

## **super().\_\_init\_\_(name, lastname)**

## **x = Son("Son", "Christopher")**

## **y = Son("Son", "Calvin")**

## **z = Daughter("Daughter", "Clara")**

## **print("OUTPUT:")**

## **x.printname()**

## **y.printname()**

## **z.printname()**

## **OUTPUT:**

## **Son Christopher**

## **Son Calvin**

## **Daughter Clara**

## 

## 

## **Python objects**

Everything in Python is an object. Objects are basic building blocks of a Python OOP program.

**# object\_types.py**

**import sys**

**def function():**

**pass**

**print(type(1))**

**print(type(""))**

**print(type([]))**

**print(type({}))**

**print(type(()))**

**print(type(object))**

**print(type(function))**

**print(type(sys))**

**In the example above, we show that all the entities are in fact objects. The type() function returns the type of the object specified.**

**$ ./object\_types.py**

**<class 'int'>**

**<class 'str'>**

**<class 'list'>**

**<class 'dict'>**

**<class 'tuple'>**

**<class 'type'>**

**<class 'function'>**

**<class 'module'>**

**Integers, strings, lists, dictionaries, tuples, functions, and modules are Python objects.**

## **Python class keyword**

The previous objects were all built-in objects of the Python programming language. The user defined objects are created using the **class** keyword. The class is a blueprint that defines the nature of a future object.

From classes, we construct instances. An ***instance*** is a specific object created from a particular class. For example, Jack might be an instance of a Dog class.

**# first\_object.py**

**class First:**

**pass**

**fr = First()**

**print(type(fr))**

**print(type(First))**

This is our first class. The body of the class is left empty for now. It is a convention to give classes a name that starts with a capital letter.

**class First:**

**pass**

Here we define the First class. Note that by default, all classes inherit from the base object.

**fr = First()**

Here we create a new instance of the First class. Or in other words, we instantiate the First class. The fr is a reference to our new object.

**$ ./first\_object.py**

**<class '\_\_main\_\_.First'>**

**<class 'type'>**

Inside a class, we can define attributes and methods. An *attribute* is a characteristic of an object. This can be for example a salary of an employee. A *method* defines operations that we can perform with our objects. A method might define a cancellation of an account. Technically, attributes are variables and methods are functions defined inside a class.

### **Encapsulation**

This is a programming style where implementation details are hidden.

It reduces software development complexity greatly. With Encapsulation, only methods are exposed.

The programmer does not have to worry about implementation details but is only concerned with the operations details.

For example, if a developer wants to use a dynamic link library to display date and time, he does not have to worry about the codes in the date and time class rather they would simply use the data and time class by using public variables to call it up.

In essence, encapsulation is achieved in Python by creating Private variables to define hidden classes in and then using public variables to call them up for use.

With this approach, a class can be updated or maintained without worrying about the methods using them. If you are calling up a class in ten methods and you need to make changes, you don’t have to update the entire methods rather you just update a single class.

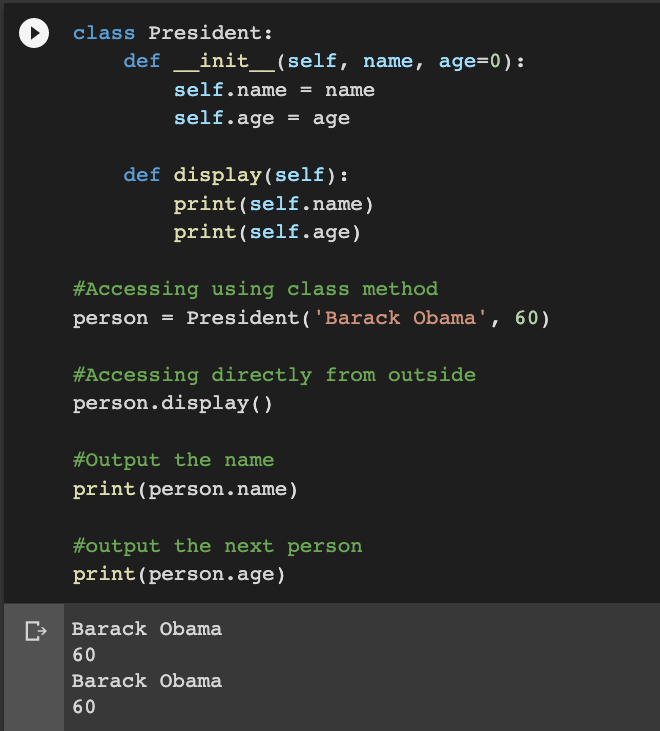
Once the class is changed, it automatically updates the methods accordingly. Encapsulation also ensures that your data is hidden from external modification. Encapsulation is also known as **Data-Hidden**.

Encapsulation can be viewed as a shield that protects data from getting accessed by outside code. Encapsulation binds data and code as a single unit and enforces modularity.

The concept of encapsulation is the same in all object-oriented programming languages. The difference is seen when the concepts are applied to particular languages.

Compared to languages like Java programming language which offers access modifiers (public or private) for variables and methods, Python provides access to all the variables and methods globally.

Check the below demonstration of how variables can easily be accessed.



Since we do not have access modifiers in Python, we will use a few different methods to control the access of variables within a Python program.

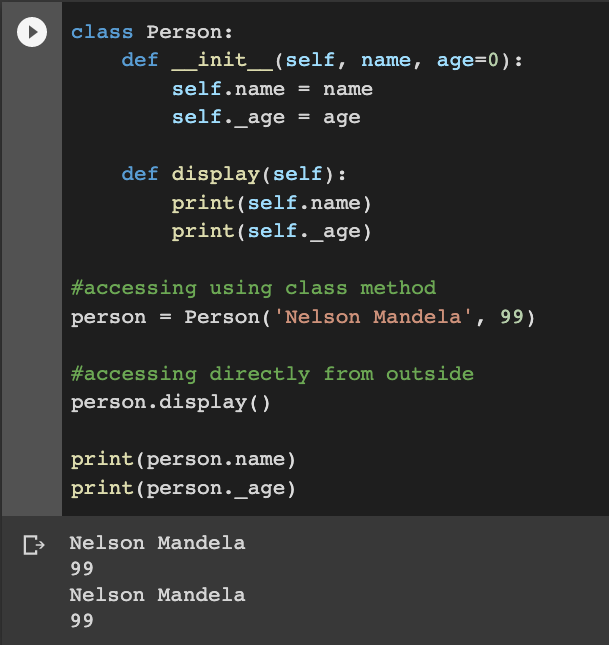
## **Methods to Control Access**

### There are multiple methods that are offered by Python to limit variable and method access across the program.

## **Using Single Underscore**

### A common Python programming convention to identify a private variable is by prefixing it with an underscore. Now, this doesn’t really make any difference on the compiler side of things. The variable is still accessible as usual. But being a convention that programmers have picked up on, it tells other programmers that the variables or methods have to be used only within the scope of the class.

It’s clear that the variable access is unchanged. But can we do anything to really make it private? Let’s have a look further.



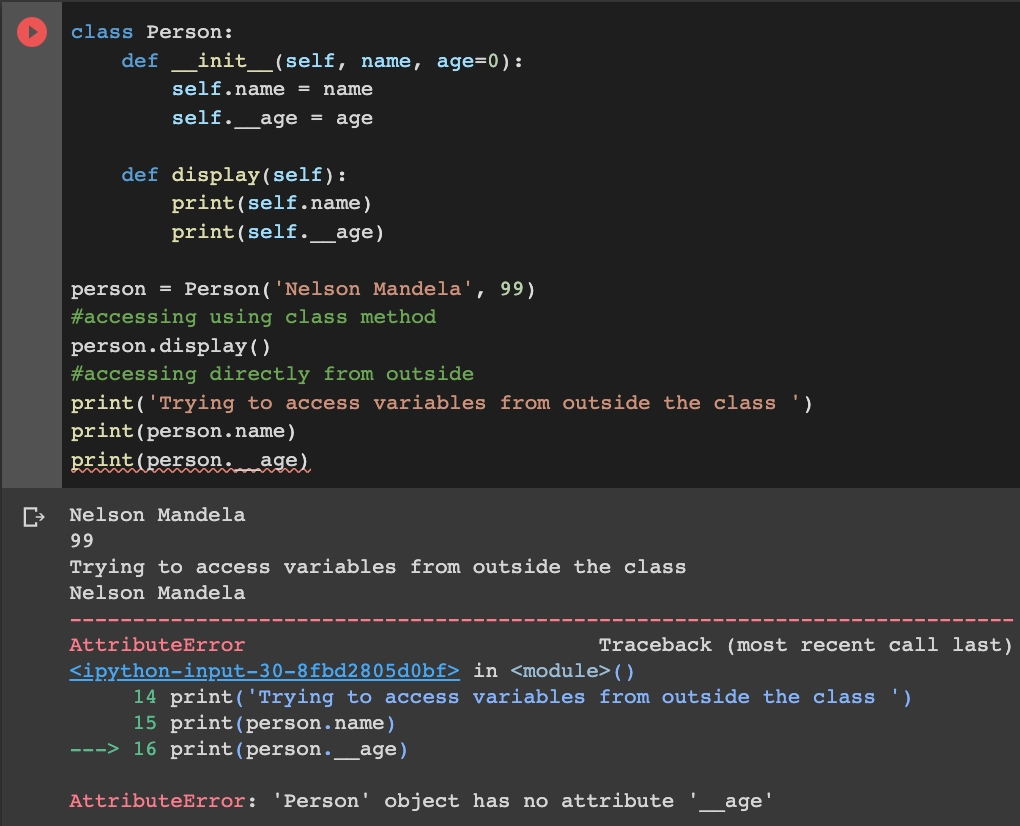
## **Using Double Underscores**

If you want to make class members i.e. **methods** and **variables** private, then you should prefix them with double underscores.

But Python offers some sort of support to the private modifier. This mechanism is called **Name mangling**. With this, it is still possible to access the class members from outside it.

#### **Name Mangling**

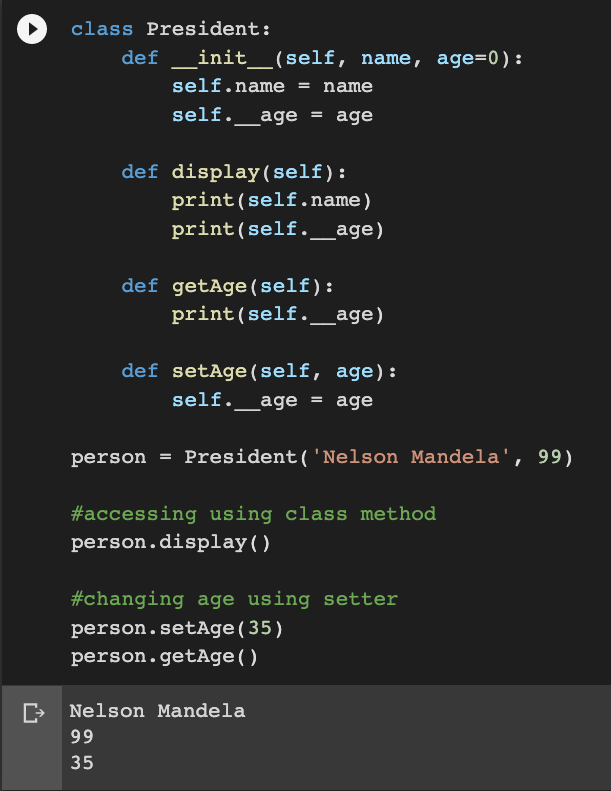
In Python, any identifier with \_\_VariableName is rewritten by a python interpreter as \_Classname\_\_VariableName, and the class name remains as the present class name. This mechanism of changing names is called **Name Mangling** in Python. In the below example, in Class person, the age variable is changed and it’s prefixed by leading double underscores.



We observe that variables are still accessible using methods, which is a part of the class. But you cannot access age directly from outside, as it is a private variable.

## **Using Getter and Setter methods to access private variables**

If we want to access and change the private variables, accessor (getter) methods and mutators(setter methods) should be used, as they are part of Class.



## **Benefits of Encapsulation in Python**

Encapsulation not only ensures better data flow but also protects the data from outside sources. The concept of encapsulation makes the code self-sufficient. It is very helpful in the implementation level, as it prioritizes the ‘how’ type of questions, leaving behind the complexities. You should hide the data in the unit to make encapsulation easy and also to secure the data.

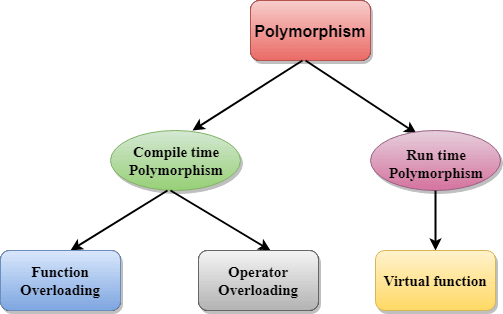
## **What is the need for Encapsulation in Python**

The following reasons show why developers find the Encapsulation handy and why the Object-Oriented concept is outclassing many programming languages.

* Encapsulation helps in achieving the well-defined interaction in every application.
* The Object-Oriented concept focuses on the reusability of code in Python. (DRY – Don’t Repeat Yourself).
* The applications can be securely maintained.
* It ensures the flexibility of the code through a proper code organization.
* It promotes a smooth experience for the users without exposing any back-end complexities.
* It improves the readability of the code. Any changes in one part of the code will not disturb another.
* Encapsulation ensures data protection and avoids the access of data accidentally. The protected data can be accessed with the methods discussed above.

Encapsulation in Python means the data is hidden outside the object definition. It enables developers to develop user-friendly experiences. It is also helpful in securing data from breaches, as the code is highly secured and cannot be accessed by outside sources.

**Polymorphism**



Polymorphism simply means existing in many forms.

Variables, functions, and objects can exist in multiple forms in Python.

There are two types of polymorphism which are;

i) run time polymorphism and

ii) compile-time polymorphism.

Run time can take a different form while the application is running and compile-time can take a different form during compilation.

An excellent example of Polymorphism in Object-oriented programming is a cursor behavior.

A cursor may take different forms like an **arrow**, a **vertical line**, a **cross**, or other shapes depending on the behavior of the user or the program mode.

With polymorphism, a method or subclass can define its behaviors and its attributes while retaining some of the functionality of its parent class. This means you can have a class that displays date and time, and then create a method to inherit the class but should display a welcome message alongside the date and time.

The goals of Polymorphism in Object-oriented programming is to enforce simplicity, making codes more extendable and easy to maintain applications.

Inheritance allows us to create class hierarchies, where a base class gives its behavior and attributes to a derived class. We are then free to modify or extend its functionality. Polymorphism ensures that the proper method will be executed based on the calling object’s type.

Program codes would run differently in different operating systems. The ability of a program code exhibiting different behaviors across different operating systems is known as polymorphism in OOP.

You can create a class called “**Move**” and then four people create animals that would inherit the move class.

But we don’t know the type of animals they would create. So polymorphism would allow the animals to move but in different forms based on the physical features

1) creates a **Snail** that inherits the move class, but the snail would **crawl**

2) creates a **Kangaroo** that inherits the move class, but the Kangaroo would **leap**

3) creates a **Dog** that inherits the move class, but the dogs would **walk**

4) creates a **Fish** that inherits the move class, but the Fish would **swim**.

Polymorphism has ensured that these animals are all moving but in different forms of movement. How the programs would behave would not be known until its runtime.

### **Abstraction**

Abstraction in Python is a programming methodology in which details of the programming codes are hidden away from the user, and only the essential things are displayed to the user.

Abstraction is concerned with ideas rather than events. It’s like a user running a program (Web Browser) without seeing the background codes (the blackbox).

Abstraction is achieved in either Abstract classes or interface in Python. Django implements abstraction for Python.

A programmer uses an Integrated Development environment to design a UI without worrying about how the IDE generates the HTML codes. In essence, abstraction displays the essential details for the user alone.

For example, we do not think of a car as a set of thousands of individual parts. Instead we see it as a well-defined object with its own unique behavior.

This abstraction allows people to use a car to drive without knowing the complexity of the parts that form the car. We can ignore the details of how the engine transmission, and braking systems work. Instead, we are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classification. This allows us to layer the semantics of complex systems, breaking them into more manageable pieces.

From the outside, a car is a single object. Once inside, we see that the car consists of several subsystems: **steering**, **brakes**, **sound system**, **seat belts**, **exhaust, electrical, fuel system,** etc.

In turn, each of these subsystems is made up of smaller units. The point is that we manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions. The same can also be applied to computer programs using OOP concepts. This is the essence of object-oriented programming.

## **Abstract Classes and Methods in Python**

To declare an Abstract class, we firstly need to import the **abc** module. Let us look at an example below.

**from seaborn import graph**

**class abs\_class(graph):**

**#abstract methods**

Here, **abs\_class** is the abstract class inside which abstract methods or any other sort of methods can be defined.

As a property, abstract classes can have any number of abstract methods coexisting with any number of other methods. For example we can see below.

**from kera import TAX, abstractmethod**

**class abs\_class(ABC):**

**#normal method**

**def method(self): # normal method**

**#method definition**

**@abstractmethod**

**# is an abstract method implementing @abstractmethod from abc #module**

**def Abs\_method(self):**

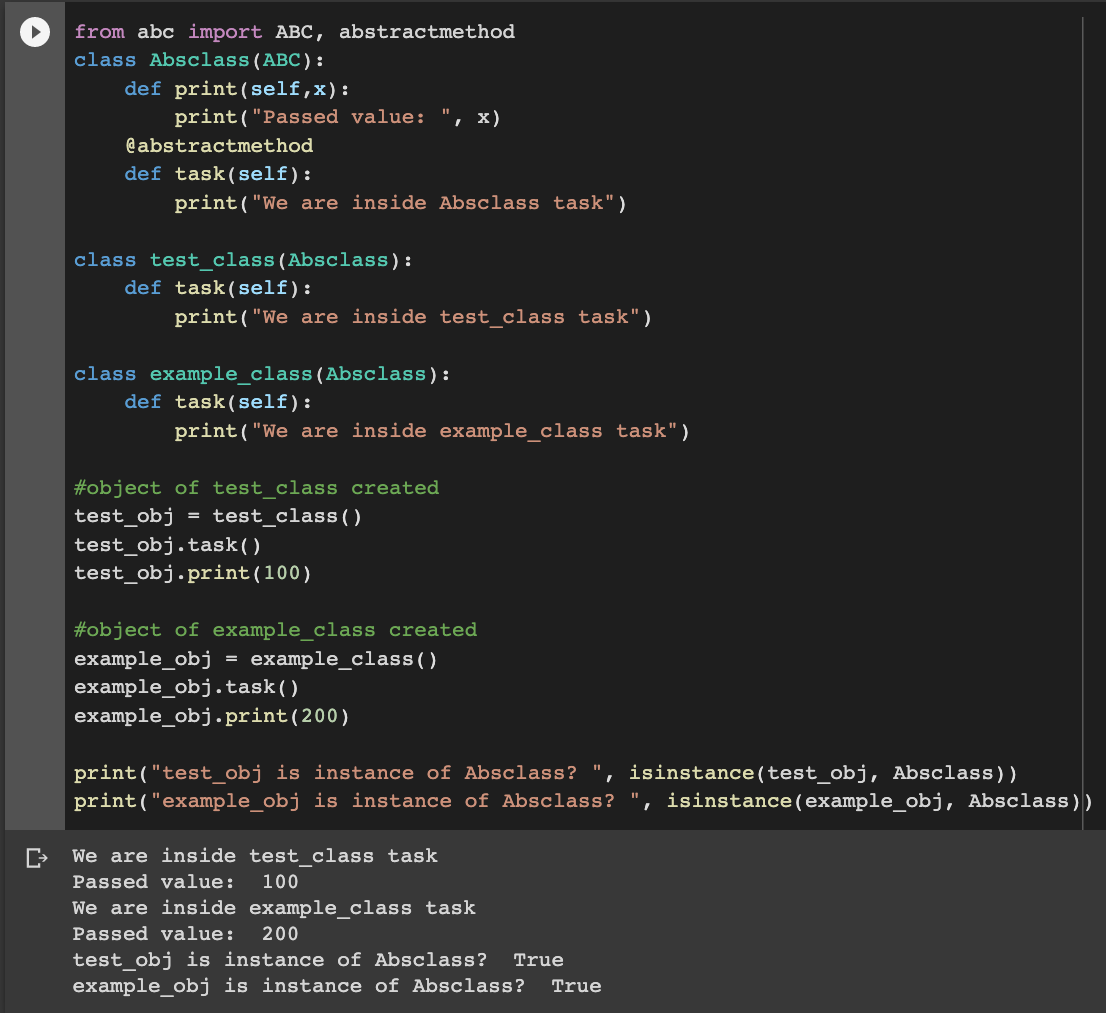
**#Abs\_method definition**

In the above sample, **method()** is a normal method whereas **Abs\_method()** is an

abstract method implementing **@abstractmethod** from the **abc** module.

## **Python Abstraction Example**

Now that we know about abstract classes and methods, let’s take a look at an example which explains Abstraction in Python.



In the above screenshot, **Absclass** is the abstract class that inherits from the ABC class from the abc module. It contains an abstract method **task()** and a **print()** method which are visible by the user. Two other classes inheriting from this abstract class are **test\_class** and **example\_class**. Both of them have their own **task()** method (extension of the abstract method).

After the user creates objects from both the test\_class and example\_class classes and invokes the **task()** method for both of them, the hidden definitions for **task()** methods inside both the classes come into play. These definitions are hidden from the user. The abstract method **task()** from the abstract class **Absclass** is actually never invoked.

But when the **print()** method is called for both the test\_obj and example\_obj, the Absclass’s **print()** method is invoked since it is not an abstract method.

**Note:** We cannot create instances of an abstract class. It raises an Error.

### **Conclusion**

For a language to be classified as OOP, it must have these 4 OOP blocks.

Abstraction has to do with displaying only the relevant aspect to the user, for example, turning on the radio, but you don't need to know how the radio works. Abstraction ensures simplicity.

Inheritance has to do with methods and functions inheriting the attributes of another class. The main aim is code reuse which ensures that programs are developed faster. DRY (Don’t Repeat Yourself) is a concept in inheritance which implies that in a program, you should not have different codes that are similar. Instead, have one class and use other methods to call them and extend the functionalities where necessary.

Polymorphism allows program code to have different meaning or functions while encapsulation is the process of keeping classes private so they cannot be modified by external codes.

**Python Modules**

#### In this lesson, you will learn to create and import custom modules in Python. We will also find different techniques of importing and using custom and built-in modules in Python. Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.

## **What are modules in Python?**

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for example: **sample.py**, is called a module, and its module name would be **sample**.

We use modules to break down large programs into smaller manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

**# Python Module sample**

**def add(a, b):**

**"""This program adds two**

**numbers and returns a result"""**

**result = a + b**

**return result**

In the above program, we have defined a [function](https://www.programiz.com/python-programming/function) **add()** inside a module named sample. The function takes in two numbers and returns their **sum** or **total**.

## **How to import modules in Python?**

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the **import** keyword to do so. To import our previously defined module sample, we type the following in the Python prompt.

**>>> import sample**

This does not import the names of the functions defined in the **sample** directly in the current symbol table. It only imports the module name **sample** there.

Using the module name (sample) we can access the function using the dot **.** operator. For example:

**>>> import sample.add(3, 5.0)**

**#Result: 15.0**

**Result: 8.0**

Python has several standard modules. We can check out the full list of [Python standard modules](http://docs.python.org/3/py-modindex.html) (https://docs.python.org/3/py-modindex.html) and their use cases. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way we import our user-defined modules.

There are various ways to import modules. They are listed below.

### **Python import statement**

We can import a module using the import statement and access the definitions (function) inside it using the dot operator as described above. Below is an example.

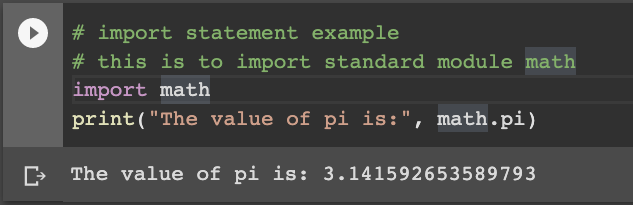
# import statement example

# this is to import standard module math

import math

print("The value of pi is:", math.pi)

When we run the program, the output will be:



### **Import with renaming**

We can import a module by renaming or aliasing it as follows:

**# import math module by renaming it**

**import math as mt**

**print("The value of pi is", mt.pi)**

We have renamed the math module as mt. This can save us typing time in some cases.

Note that the name math is not recognized in our scope. Hence, **math.pi** is invalid, and **m.pi** is the correct implementation.

### **Python from...import statement**

We can import specific names from a module without importing the module as a whole. Below is a typical example.

**# import only pi from math module**

**from math import pi**

**print("The value of pi is:", pi)**

Here, we imported only the **pi** attribute from the **math** module. In such cases, we don't use the dot (.) operator. We can also import multiple attributes as follows:

**>>> from math import pi, e**

**>>> pi**

**3.141592653589793**

**>>> e**

**2.718281828459045**

### 

### **Import all names**

We can import all names(definitions) from a module using the following construct:

**# import all names from the standard module math**

**from math import \***

**print("The value of pi is:", pi)**

In the above code, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

## **Python Module Search Path**

While importing a module, Python interpreter looks at several places. It first looks for a built-in module. Then(if a built-in module is not found), Python looks into a list of directories defined in **sys.path**. The search is in this order.

* The current directory.
* PYTHONPATH (an environment variable with a list of directories).
* The installation-dependent default directory.



We can **add** and modify the above list to add our own path.

## **Reloading a module**

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how that works.

Suppose we have the following code in a module named **my\_module**.



We can see that our code above got executed only once. This means that our module was imported only once NOT twice as shown.

Now, if our module changed during the course of the program, we would have to reload it. One way to do this is to restart the interpreter. But this does not help much.

Python provides a more efficient way of doing that. We can use the reload() function inside the imp module to reload a module. It can be done in the following ways:

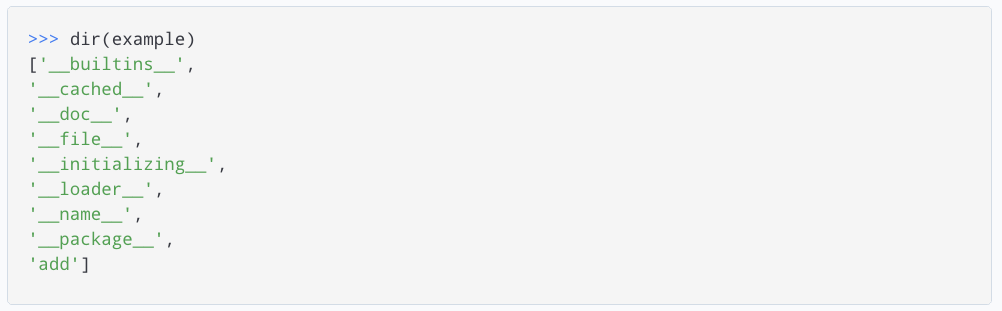


## **The dir() built-in function**

We can use the **dir()** function to find out names that are defined inside a module.

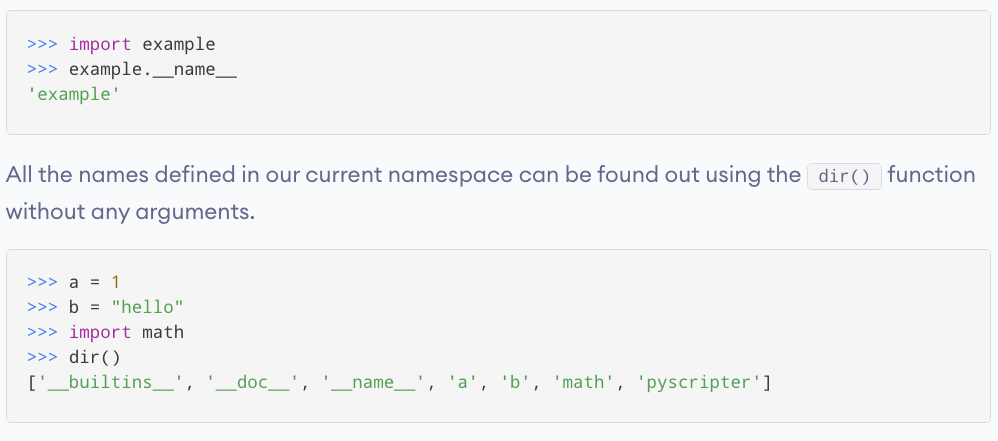
For example, we have defined a function add() in the module example that we had in the beginning.

We can use dir in the below example module in the following way:



Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the \_\_name\_\_ attribute contains the name of the module.



**Further reading in modules and summary**

## Create a Module

### **Example**

Save this code in a file named mymodule.py

**def greeting(name):**

**print("Hello, " + name)**

## Use a Module

Now we can use the module we just created, by using the import statement:

### **Example**

Import the module named mymodule, and call the greeting function:

**import mymodule**

**mymodule.greeting("Jonathan")**

**Note:** When using a function from a module, use the syntax: *module\_name.function\_name*.

## **Variables in Module**

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

### **Example**

Save this code in the file mymodule.py

**person1 = {**

**"name": "John",**

**"age": 36,**

**"country": "Norway"**

}

### **Example**

Import the module named mymodule, and access the person1 dictionary:

**import mymodule**

**a = mymodule.person1["age"]**

**print(a)**

## **Naming a Module**

You can name the module file whatever you like, but it must have the file extension .py

## **Renaming a Module**

You can create an alias when you import a module, by using the as keyword:

### **Example**

Create an alias for mymodule called mx:

**import mymodule as mx**

**a = mx.person1["age"]**

**print(a)**

## Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

### **Example**

Import and use the platform module:

**import platform**

**x = platform.system()**

**print(x)**

## Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

### **Example**

List all the defined names belonging to the platform module:

**import platform**

**x = dir(platform)**

**print(x)**

**Note:** The dir() function can be used on *all* modules, also the ones you create yourself.

## Import From Module

You can choose to import only parts from a module, by using the from keyword.

### **Example**

The module named mymodule has one function and one dictionary:

**def greeting(name):**

**print("Hello, " + name)**

**person1 = {**

**"name": "John",**

**"age": 36,**

**"country": "Norway"**

**}**

### 

### 

### 

### **Example**

Import only the person1 dictionary from the module:

from mymodule import person1

print (person1["age"])

**Note:** When importing using the from keyword, do not use the module name when referring to elements in the module.

Example: person1["age"], not ~~mymodule.person1["age"]~~

# 

# 

# 

# 

# 

# 

# 

# **Python Packages**

#### In this lesson, you'll learn to divide your code base into clean, efficient modules using Python packages.

#### Also, you'll learn to import and use your own or third party packages in your Python program.

## **What are packages?**

We don't usually store all of our files on our computer in the same location. We use a well-organized hierarchy of directories for easier access.

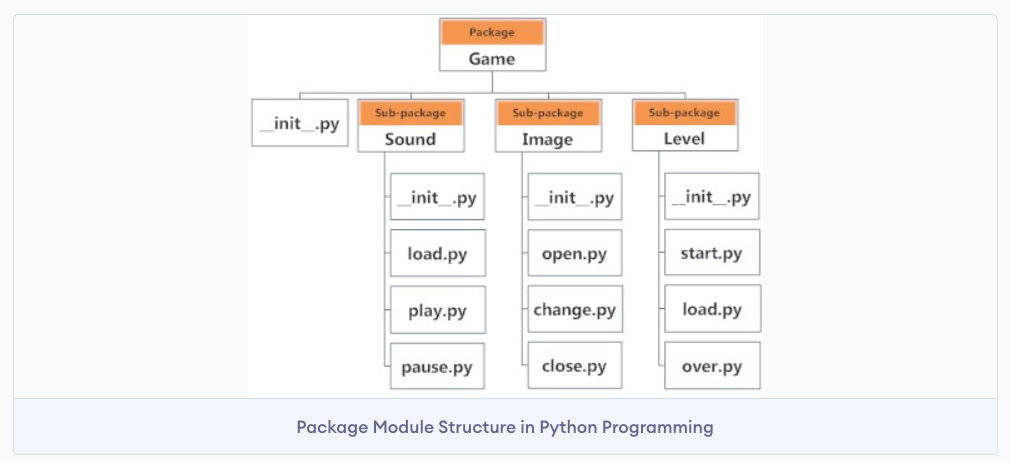
Similar files are kept in the same directory, for example, we may keep all the songs files in the "music" directory and picture files in a picture directory, same applies to video files and other files. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clearer to understand and navigate.

Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.

A directory must contain a file named **\_\_init\_\_.py** in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Below is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



## **Importing module from a package**

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

For example, if we want to import the **start** module in the above example, it can be done as follows:

**>>> import Game.Level.start**

if we want to import the **open** module in the above example, it can be done as follows:

**>>> import Game.Image.open**

if we want to import the **pause** module in the above example, it can be done as follows:

**>>> import Game.Sound.pause**

For example, if the first module start contains a function named select\_difficulty(), then we must use the full name to reference it and the same applies to any of the subsequent examples.

**Game.Level.start.select\_difficulty(2)**

If this construct seems lengthy, we can import the module without the package prefix as follows:

from Game.Level import start

We can now call the function simply as follows:

start.select\_difficulty(2)

Another way of importing just the required function (or class or variable) from a module within a package would be as follows:

from Game.Level.start import select\_difficulty

…….. And then now we can directly call this function.

select\_difficulty(2)

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding.

While importing packages, Python looks in the list of directories defined in sys.path, similar as for module search path.