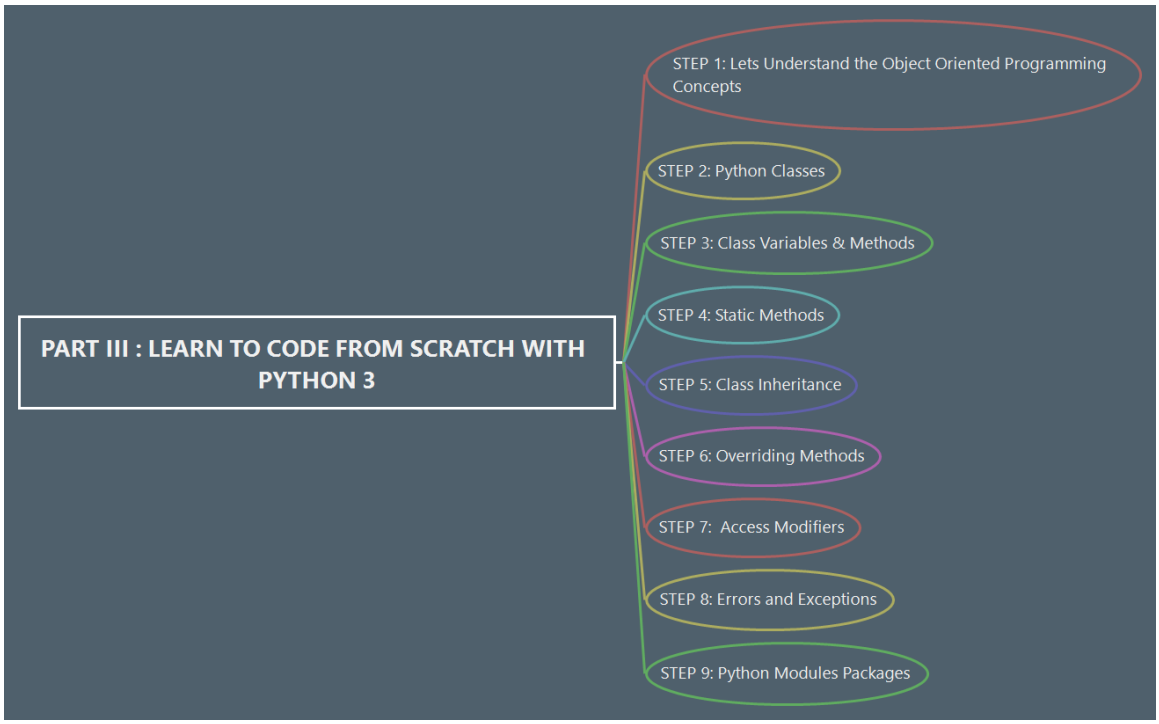
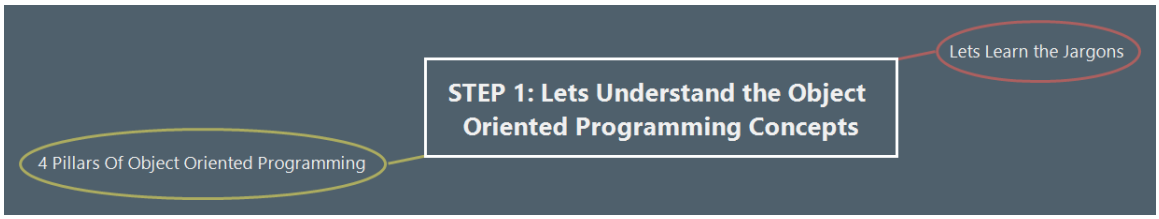
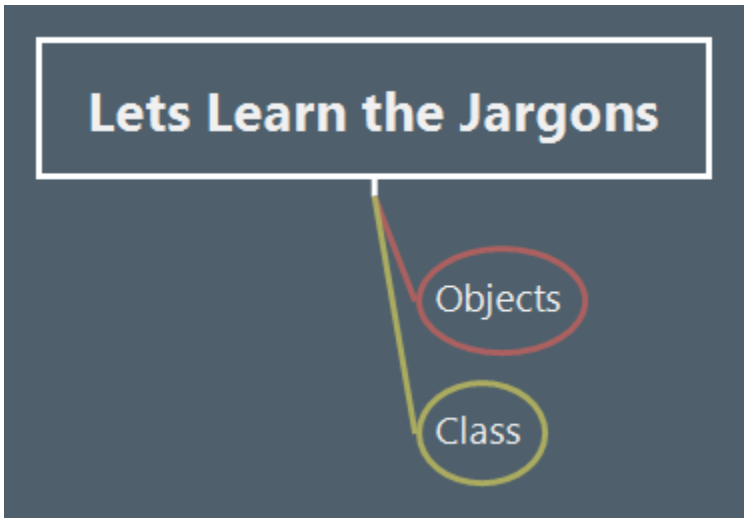
PART III : LEARN TO CODE FROM SCRATCH WITH PYTHON 3



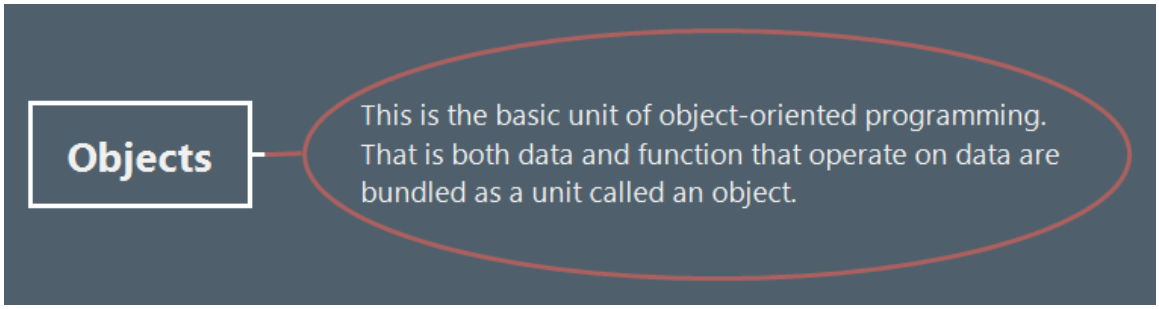
1. STEP 1: Lets Understand the Object Oriented Programming Concepts



1.1. Lets Learn the Jargons

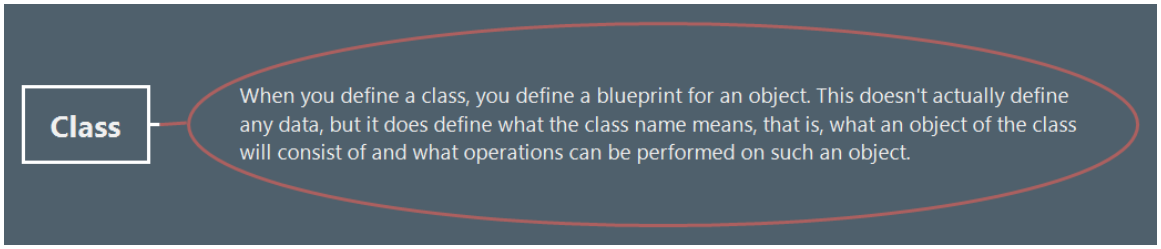


1.1.1. Objects



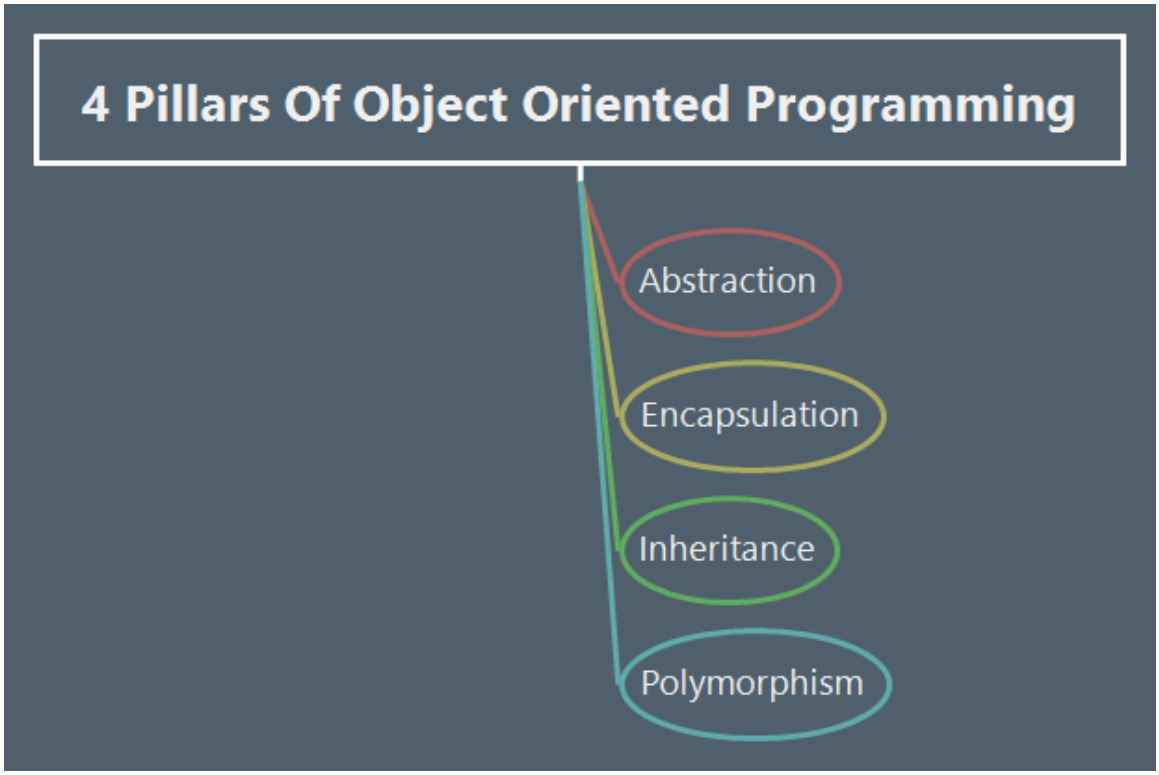
This is the basic unit of object-oriented programming. That is both data and function that operate on data are bundled as a unit called an object.

1.1.2. Class

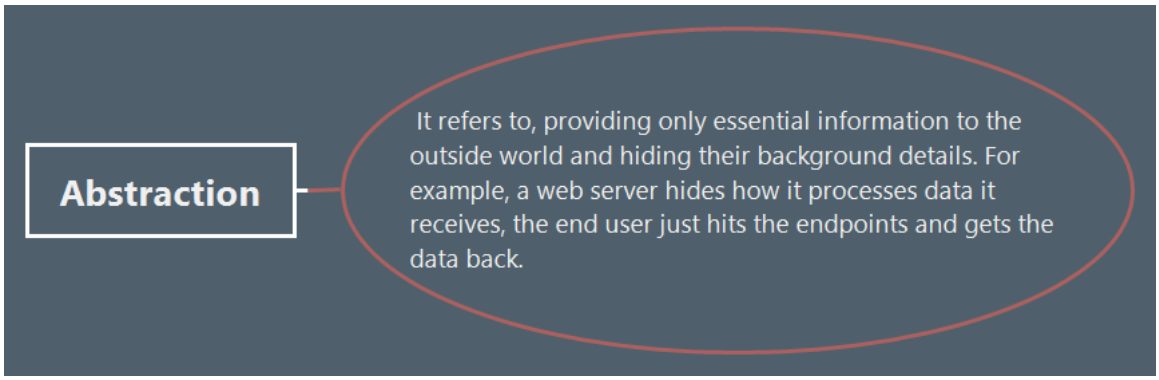


When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

1.2. 4 Pillars Of Object Oriented Programming

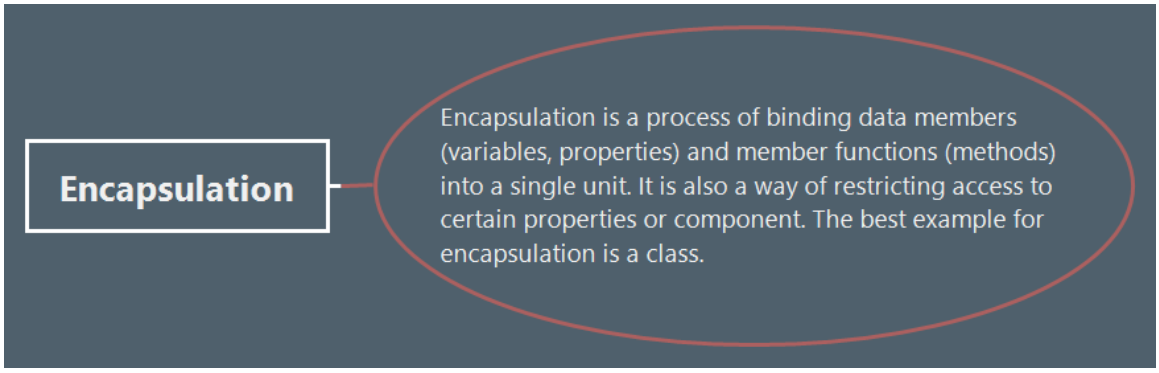


1.2.1. Abstraction



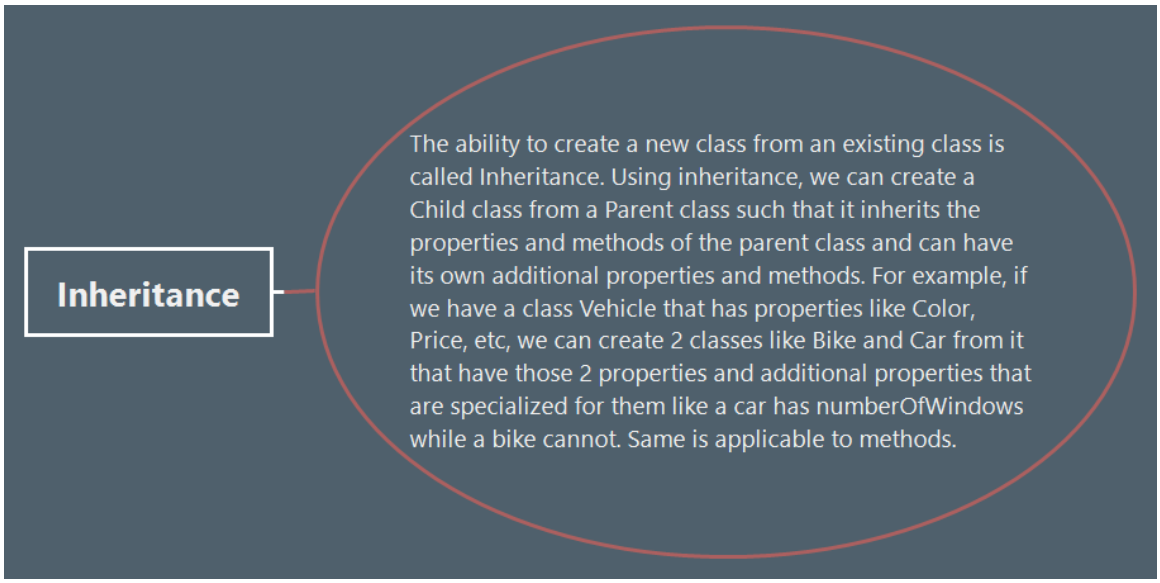
It refers to, providing only essential information to the outside world and hiding their background details. For example, a web server hides how it processes data it receives, the end user just hits the endpoints and gets the data back.

1.2.2. Encapsulation



Encapsulation is a process of binding data members (variables, properties) and member functions (methods) into a single unit. It is also a way of restricting access to certain properties or component. The best example for encapsulation is a class.

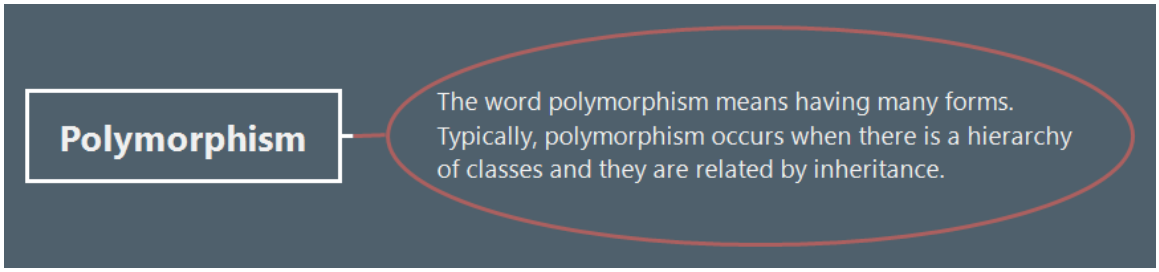
1.2.3. Inheritance



The ability to create a new class from an existing class is called Inheritance. Using inheritance, we can create a Child class from a Parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and methods. For example, if we have a class Vehicle that has properties like Color, Price, etc, we can create 2 classes like Bike and Car from it that have those 2 properties and additional properties that are

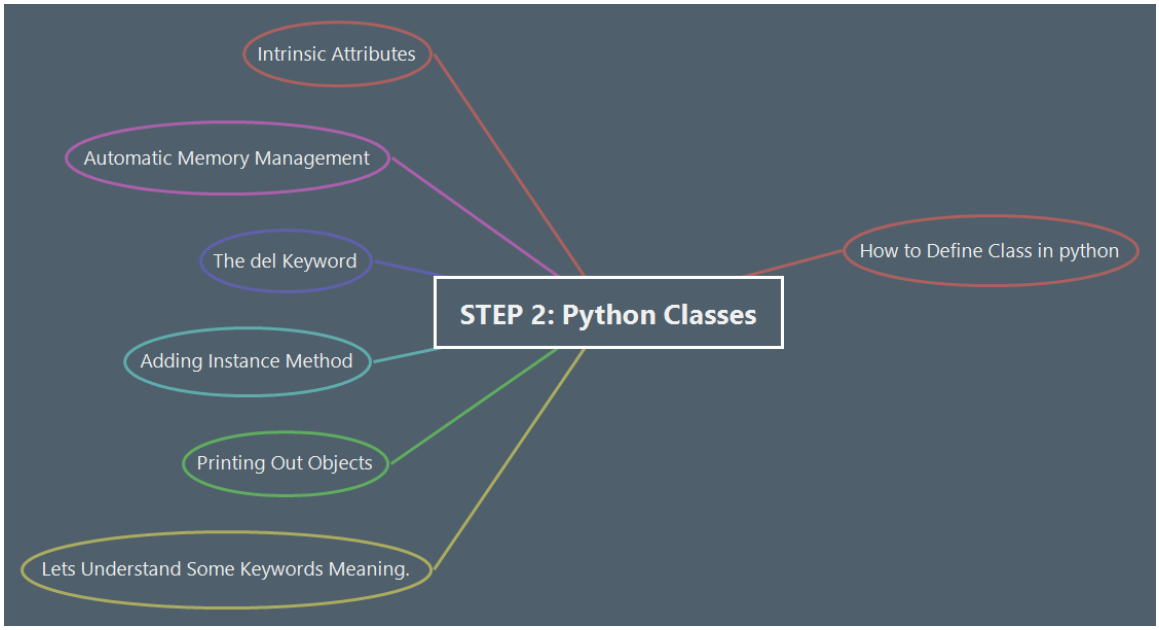
specialized for them like a car has numberOfWindows while a bike cannot. Same is applicable to methods.

1.2.4. Polymorphism

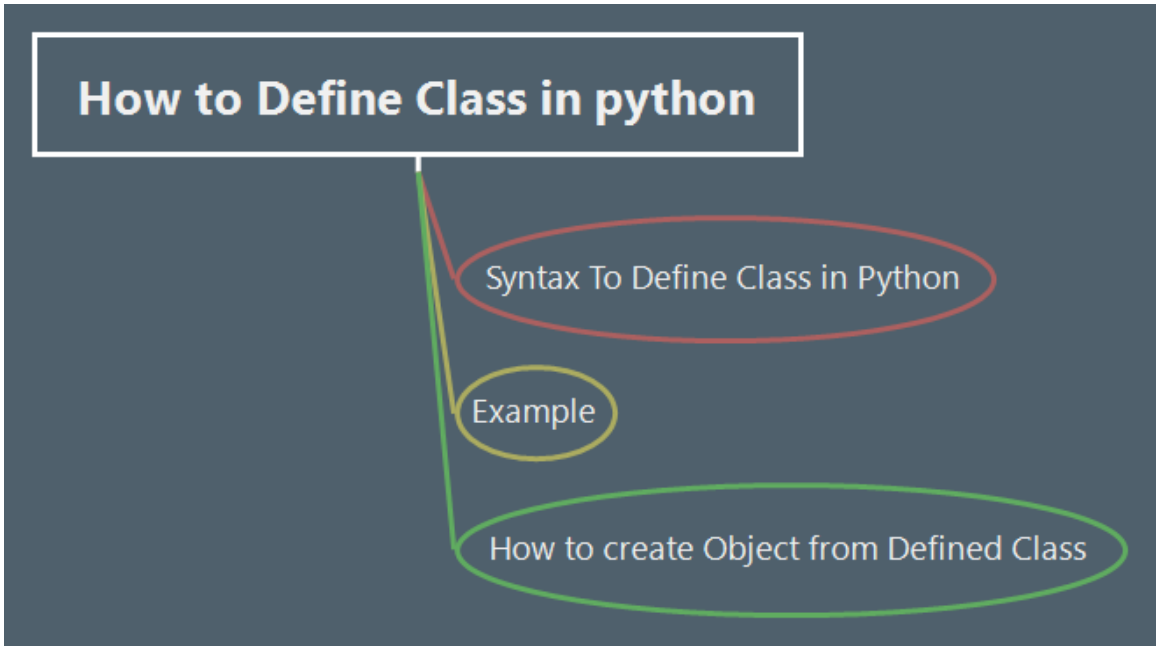


The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

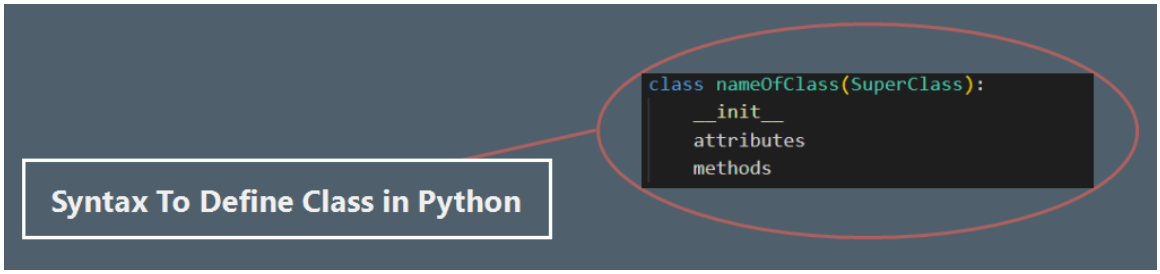
2. STEP 2: Python Classes



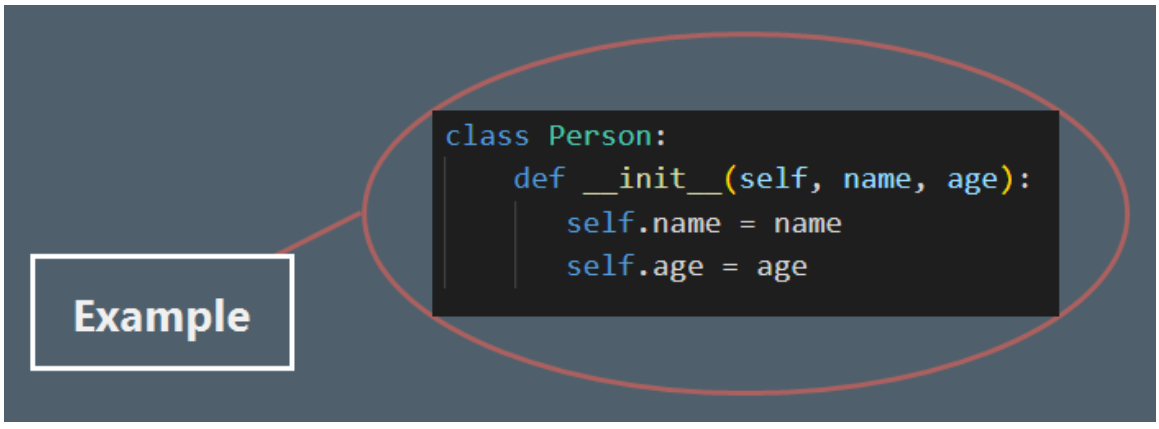
2.1. How to Define Class in python



2.1.1. Syntax To Define Class in Python

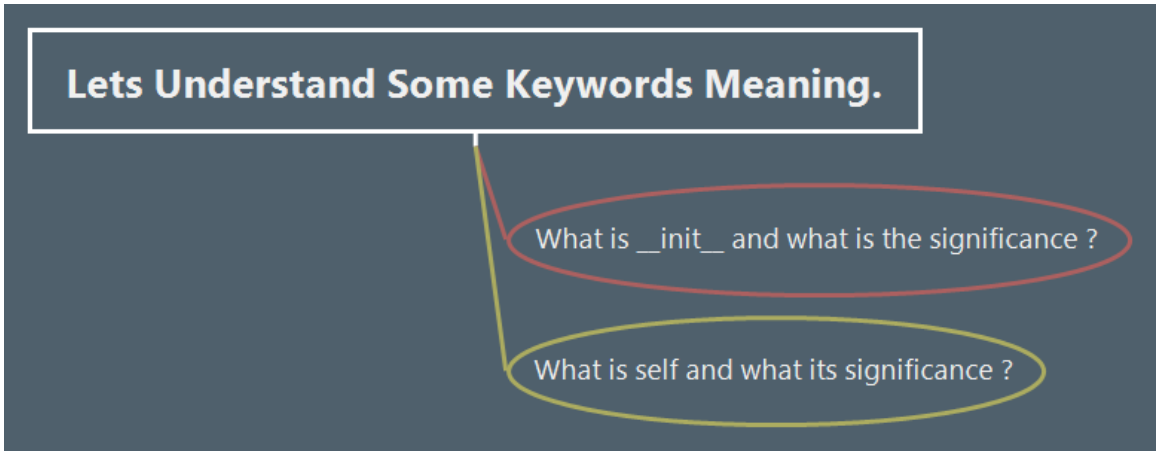


2.1.2. Example

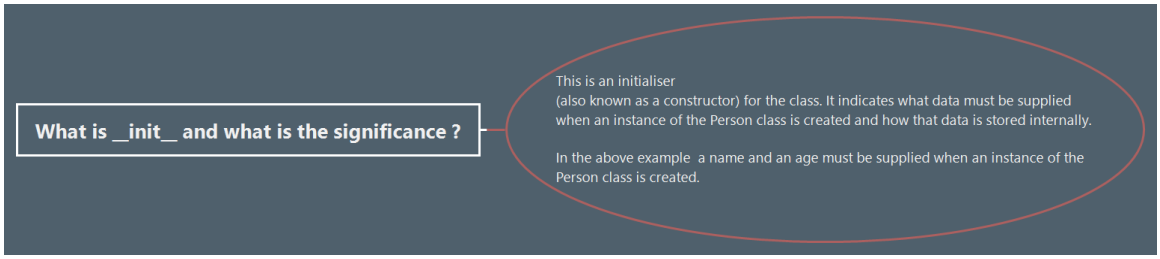


2.1.3. How to create Object from Defined Class

2.2. Lets Understand Some Keywords Meaning.



2.2.1. What is __init__ and what is the significance ?



This is an initialiser

(also known as a constructor) for the class. It indicates what data must be supplied

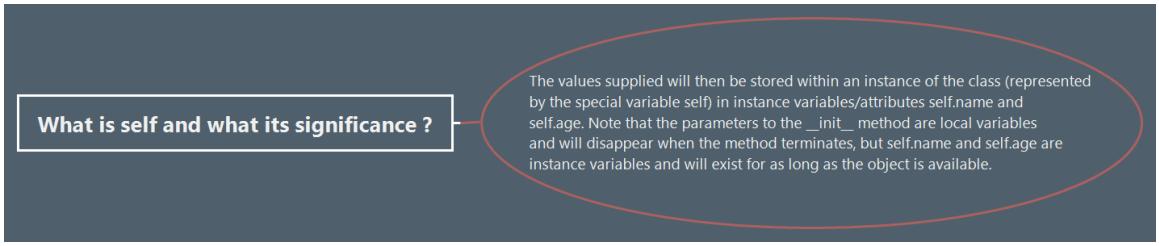
when an instance of the Person class is created and how that data is stored internally.

In the above example a name and an age must be supplied when an instance of

the

Person class is created.

2.2.2. What is self and what its significance ?

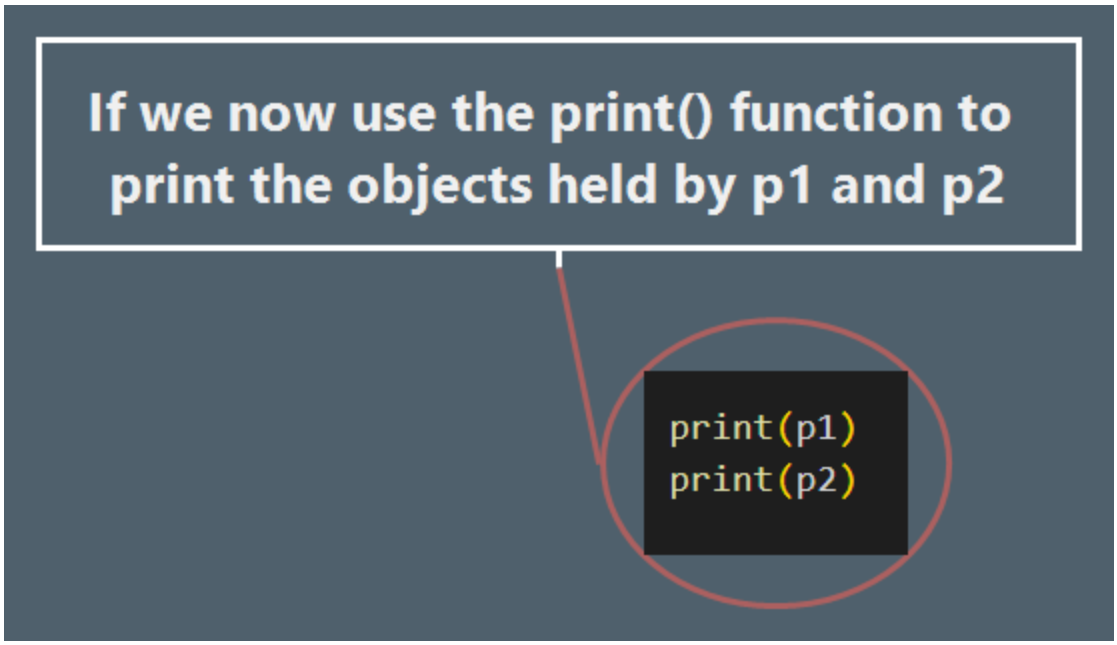


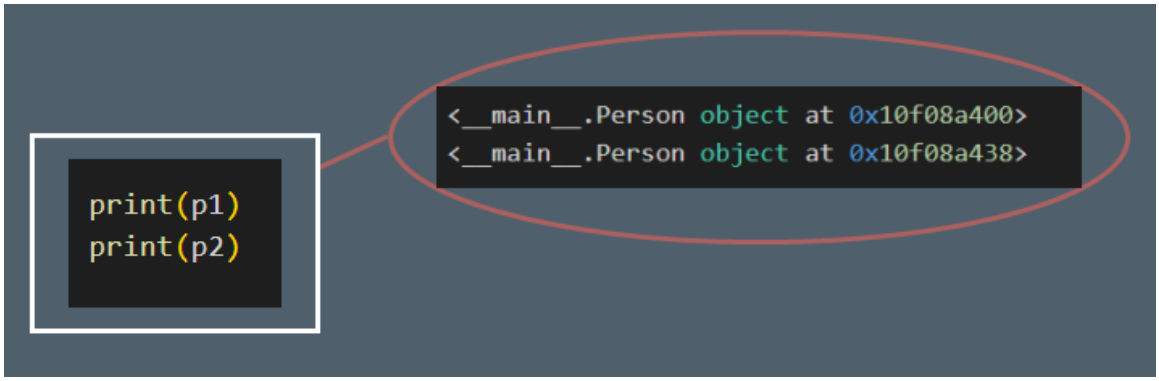
The values supplied will then be stored within an instance of the class (represented

by the special variable self) in instance variables/attributes self.name and self.age. Note that the parameters to the __init__ method are local variables and will disappear when the method terminates, but self.name and self.age are instance variables and will exist for as long as the object is available.

2.3. Printing Out Objects

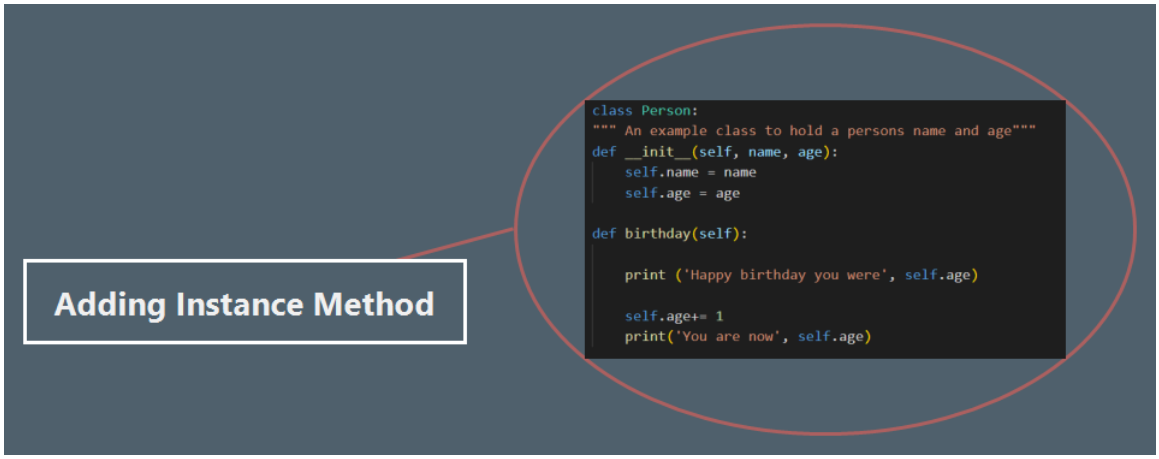
2.3.1. If we now use the print() function to print the objects held by p1 and p2



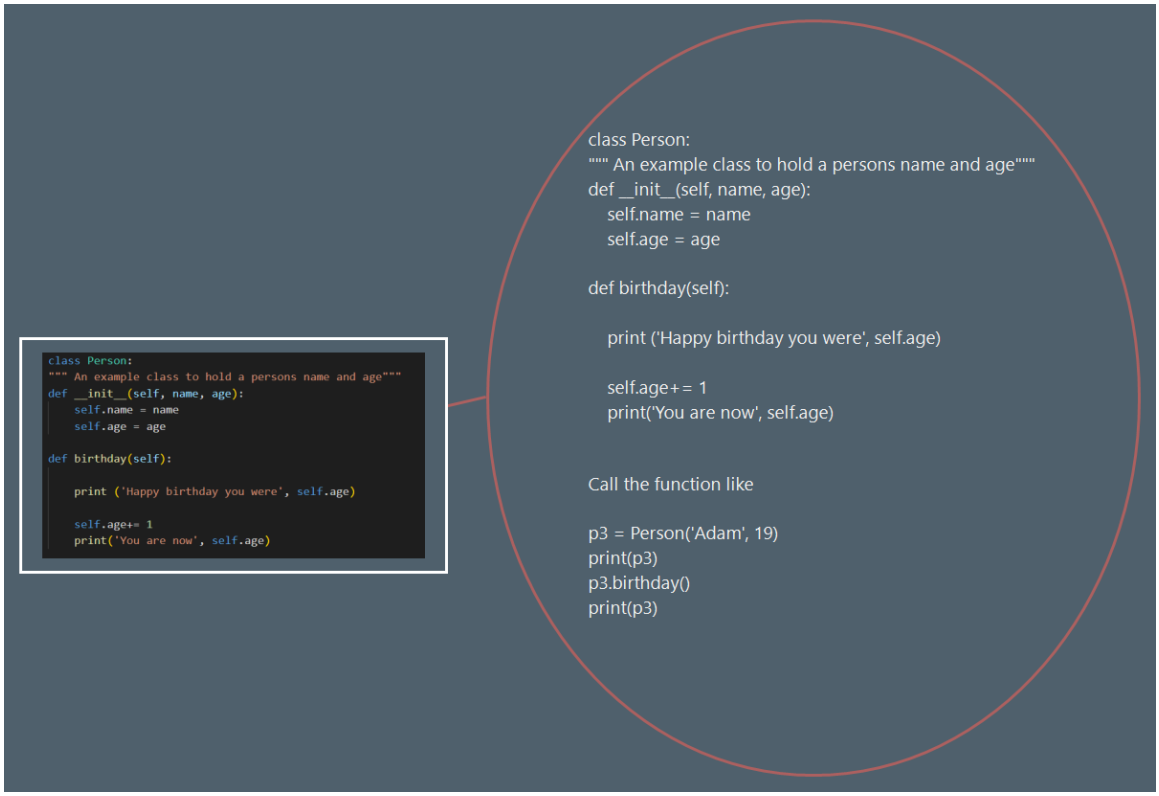


2.3.2. Accessing Object Attributes

2.4. Adding Instance Method



2.4.1.



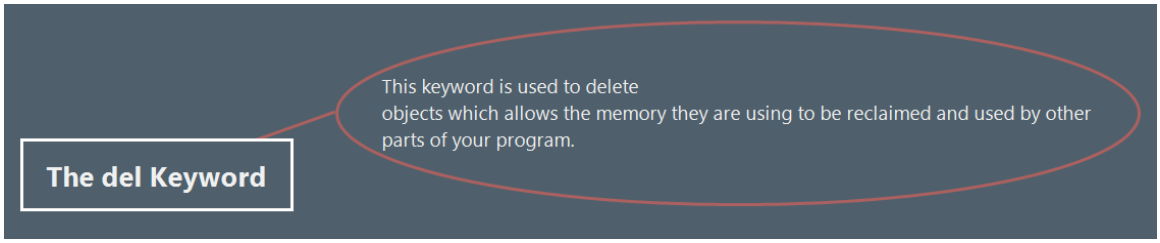

```
class Person:
""" An example class to hold a persons name and age"""
def __init__(self, name, age):
   self.name = name
   self.age = age
```

```
def birthday(self):

    print ('Happy birthday you were', self.age)

    self.age+= 1
    print('You are now', self.age)
```

Call the function like

```
p3 = Person('Adam', 19)
print(p3)
p3.birthday()
print(p3)
```

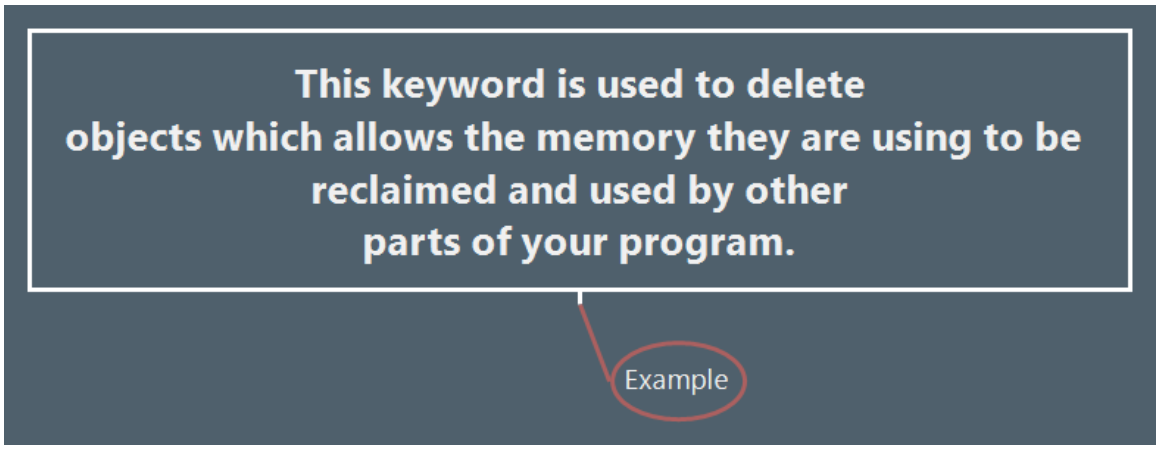
2.5. The del Keyword



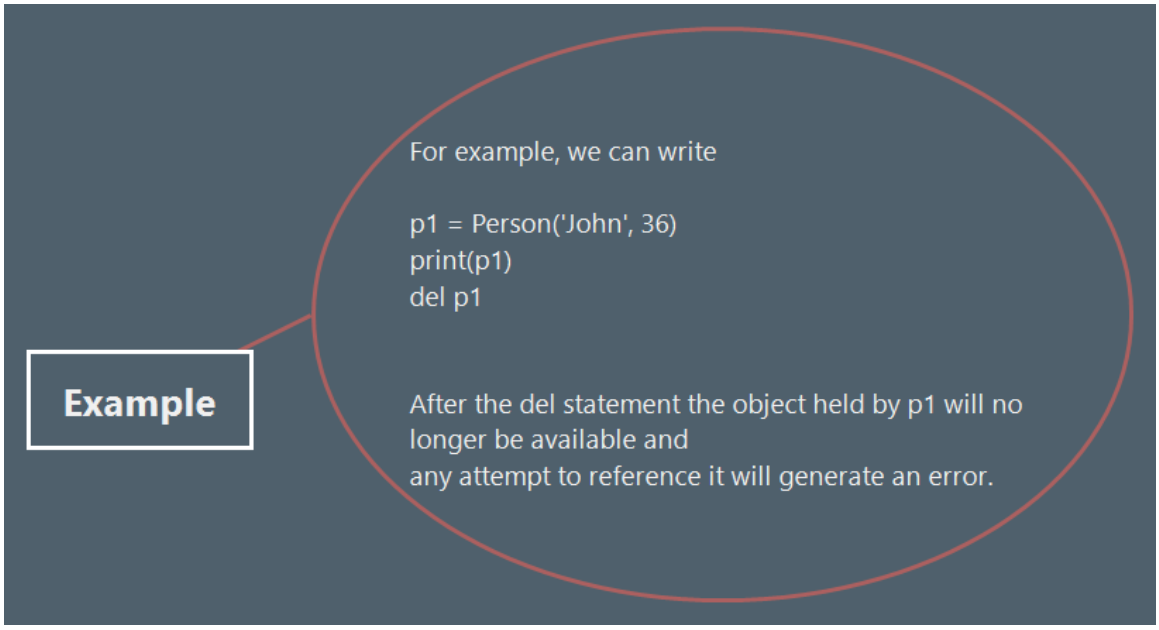
2.5.1. This keyword is used to delete

objects which allows the memory they are using to be reclaimed and used by other

parts of your program.



Example

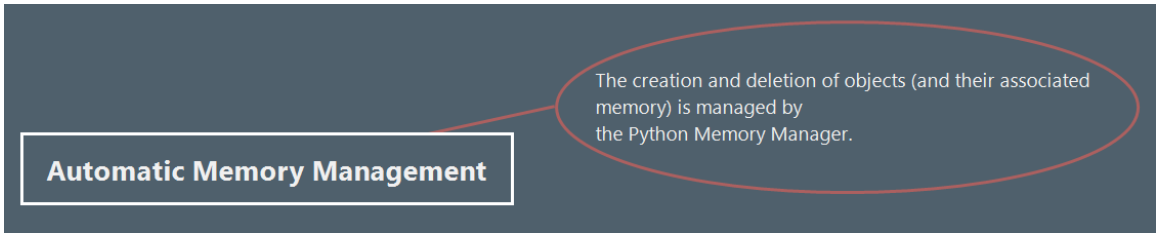


For example, we can write

```
p1 = Person('John', 36)
print(p1)
del p1
```

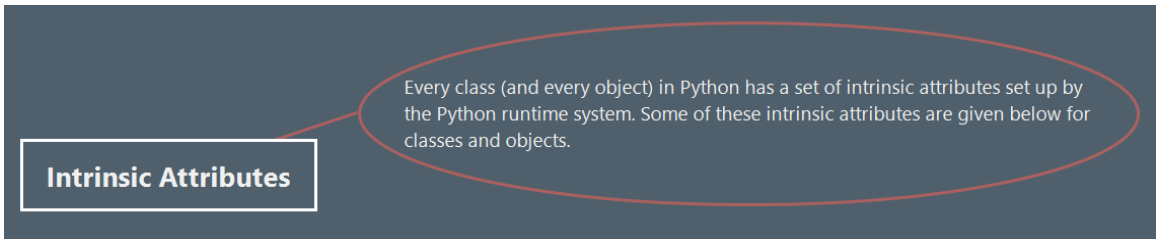
After the del statement the object held by p1 will no longer be available and any attempt to reference it will generate an error.

2.6. Automatic Memory Management

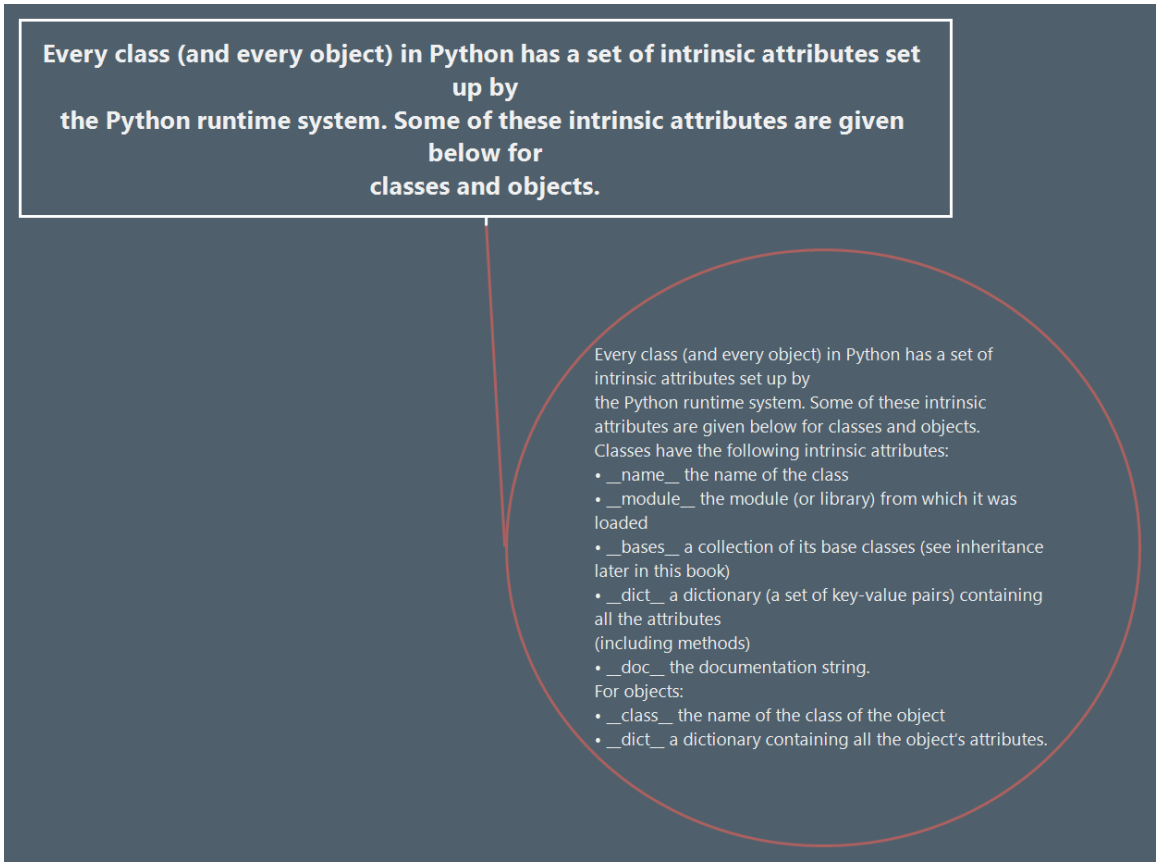


2.6.1. The creation and deletion of objects (and their associated memory) is managed by
the Python Memory Manager.

2.7. Intrinsic Attributes



2.7.1. Every class (and every object) in Python has a set of intrinsic attributes set up by
the Python runtime system. Some of these intrinsic attributes are given below for classes and objects.



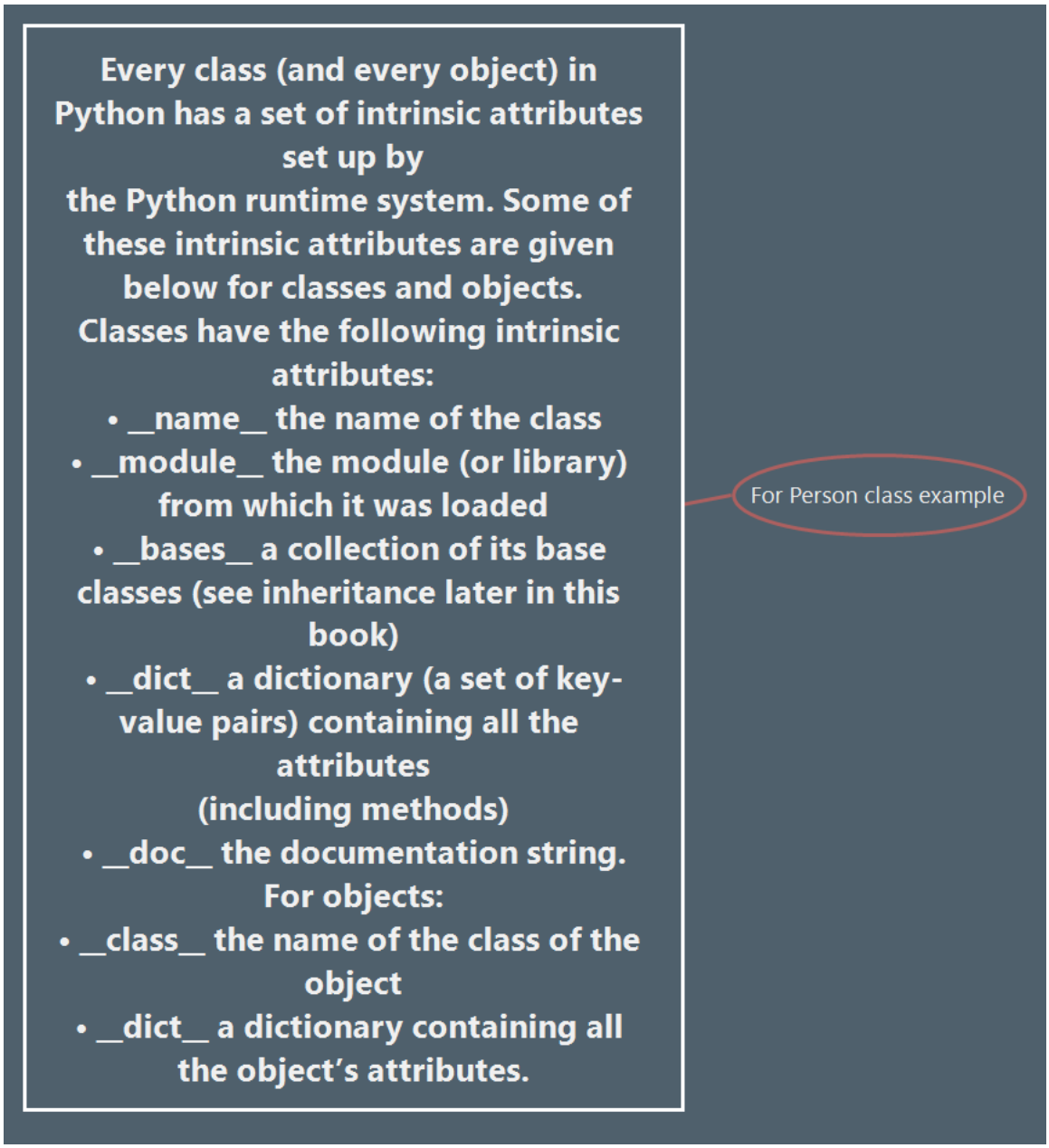
Every class (and every object) in Python has a set of intrinsic attributes set up by the Python runtime system. Some of these intrinsic attributes are given below for classes and objects.

Classes have the following intrinsic attributes:

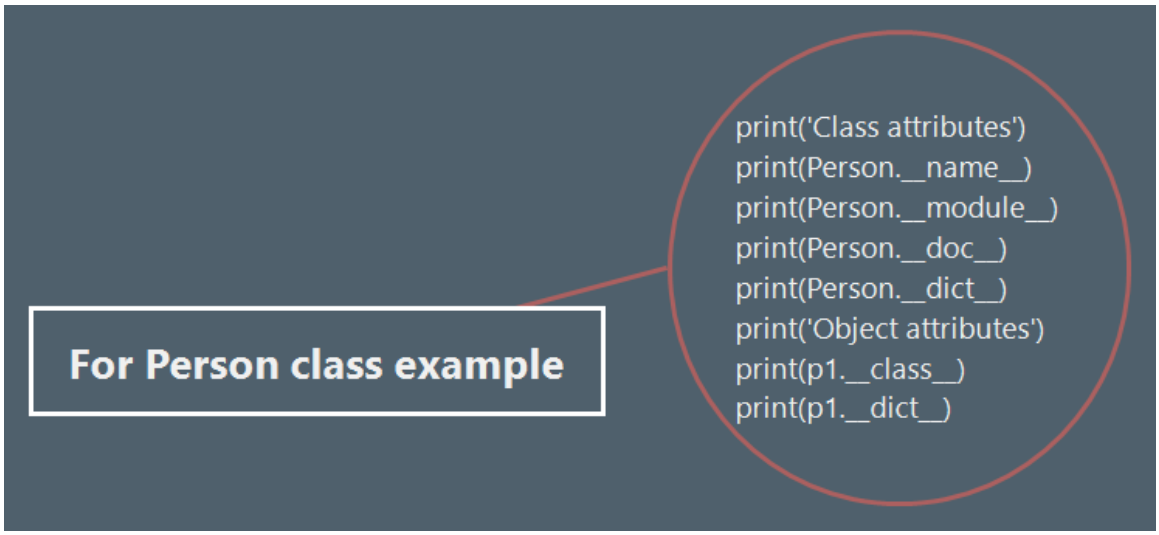
- **__name__ the name of the class**
- **__module__ the module (or library) from which it was loaded**
- **__bases__ a collection of its base classes (see inheritance later in this book)**
- **__dict__ a dictionary (a set of key-value pairs) containing all the attributes (including methods)**
- **__doc__ the documentation string.**

For objects:

- **__class__ the name of the class of the object**
- **__dict__ a dictionary containing all the object's attributes.**



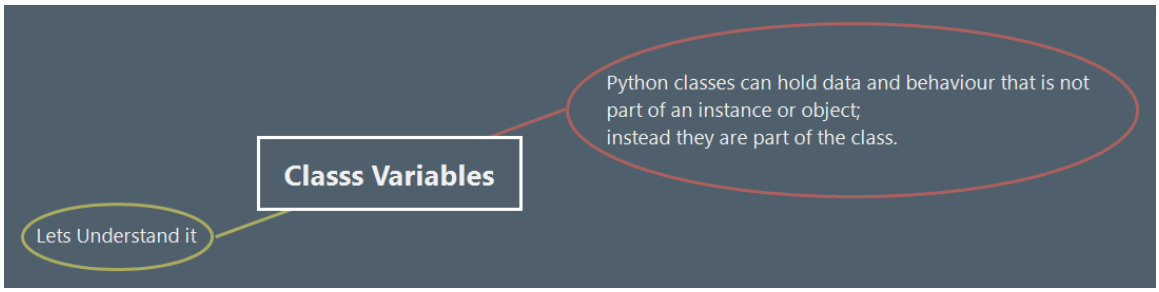
For Person class example




```
print('Class attributes')
print(Person.__name__)
print(Person.__module__)
print(Person.__doc__)
print(Person.__dict__)
print('Object attributes')
print(p1.__class__)
print(p1.__dict__)
```

3. STEP 3: Class Variables & Methods

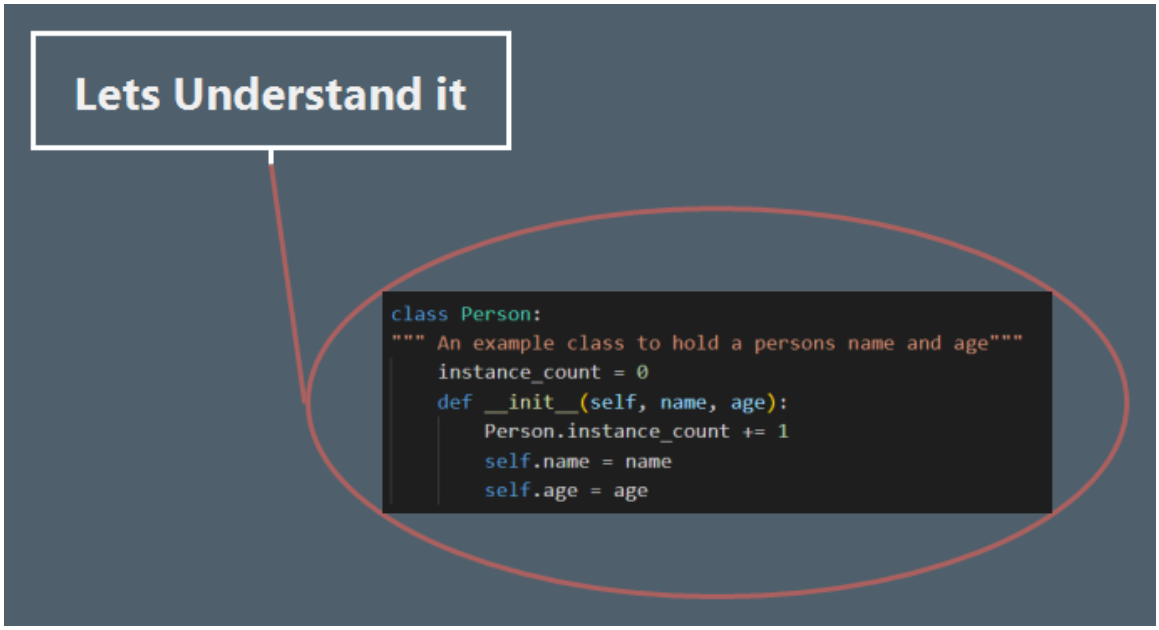
3.1. Classs Variables

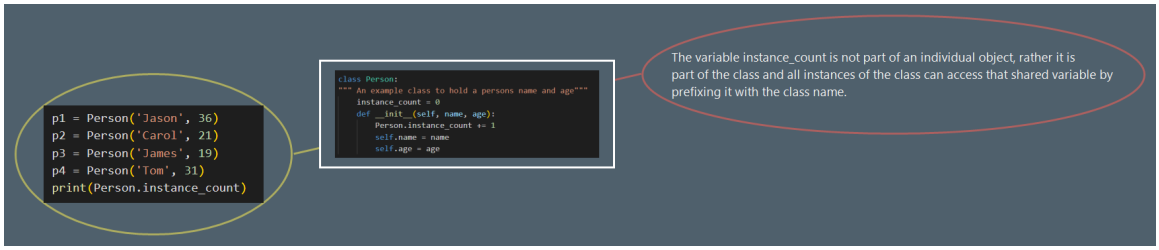


3.1.1. Python classes can hold data and behaviour that is not part of an instance or object;

instead they are part of the class.

3.1.2. Lets Understand it



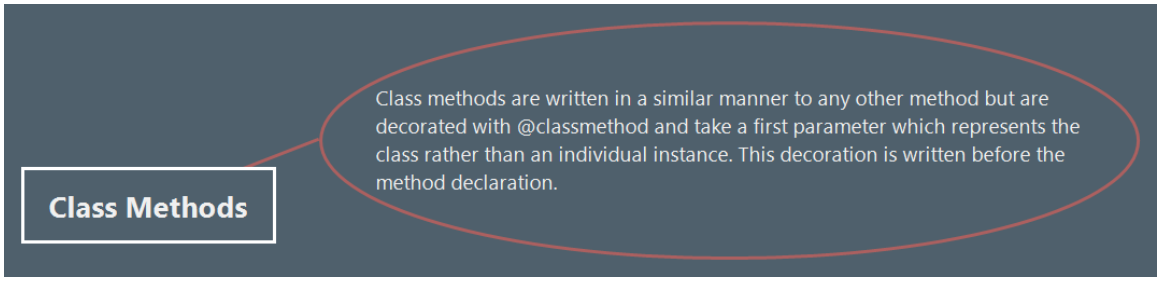


The variable instance_count is not part of an individual object, rather it is part of the class and all instances of the class can access that shared variable

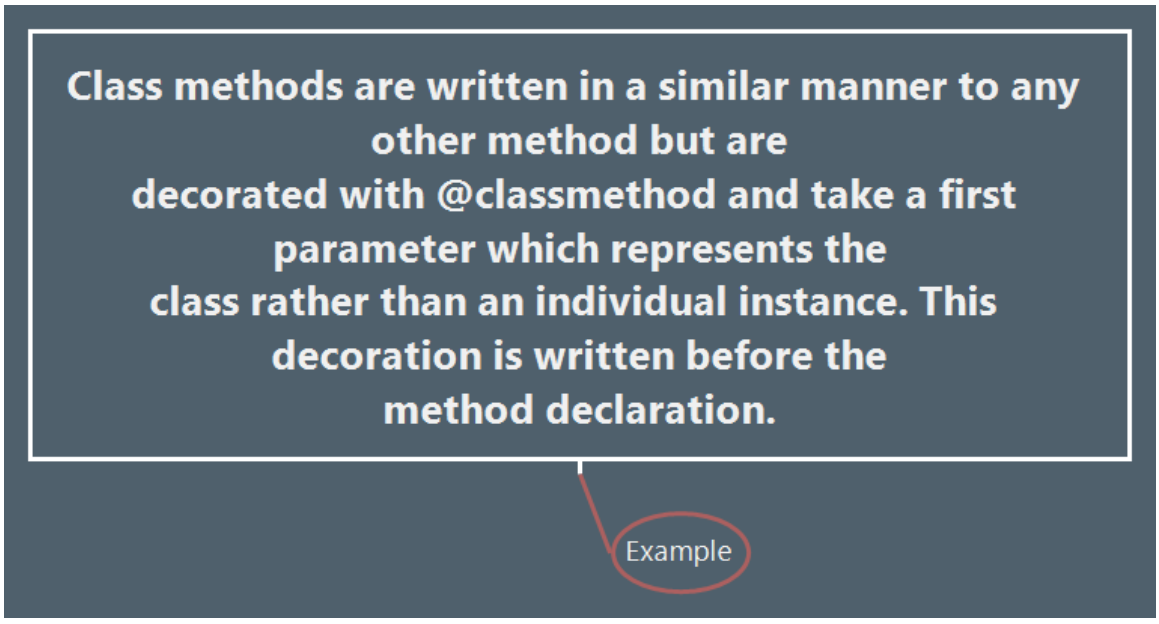
by

prefixing it with the class name.

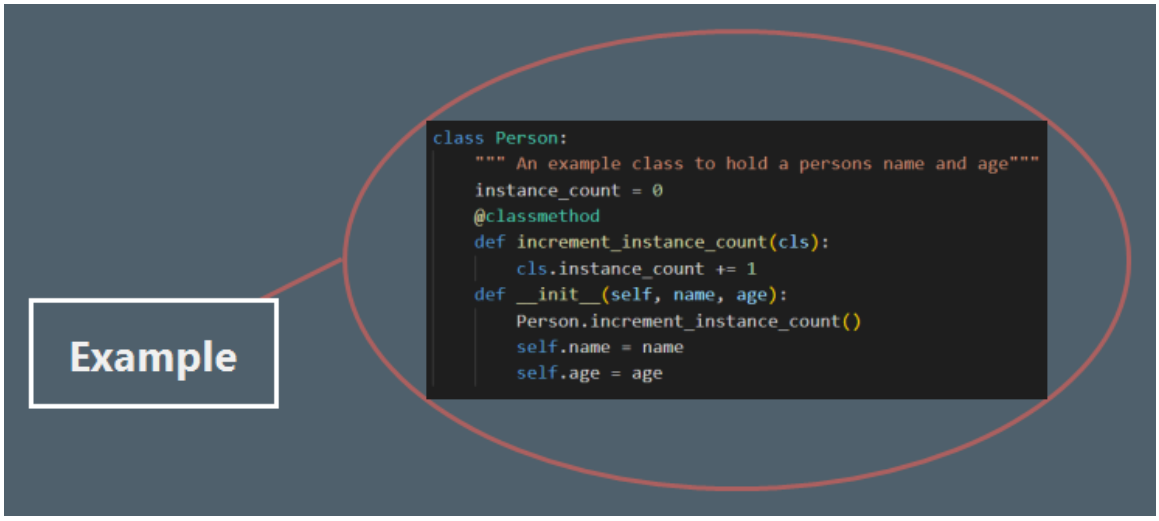
3.2. Class Methods

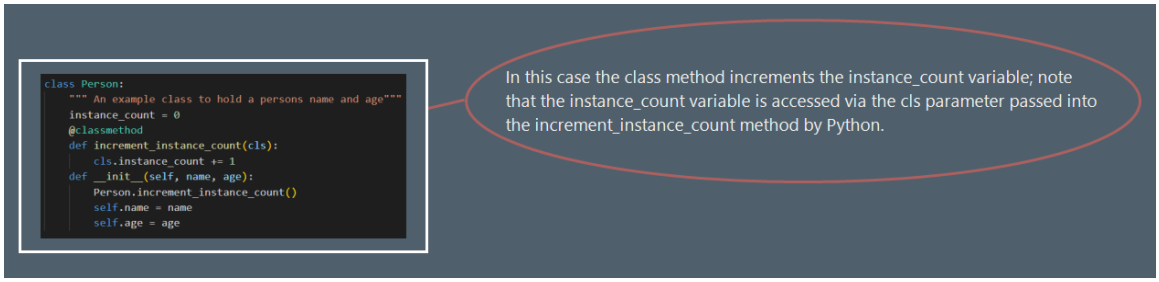


3.2.1. Class methods are written in a similar manner to any other method but are decorated with @classmethod and take a first parameter which represents the class rather than an individual instance. This decoration is written before the method declaration.



Example



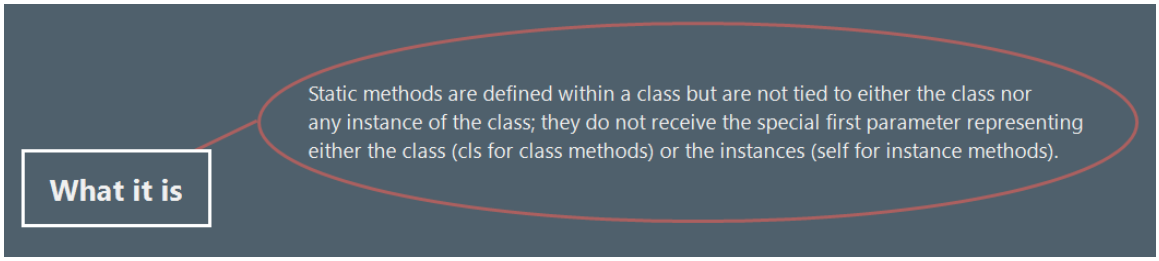


In this case the class method increments the instance_count variable; note that the instance_count variable is accessed via the cls parameter passed into

the increment_instance_count method by Python.

4. STEP 4: Static Methods

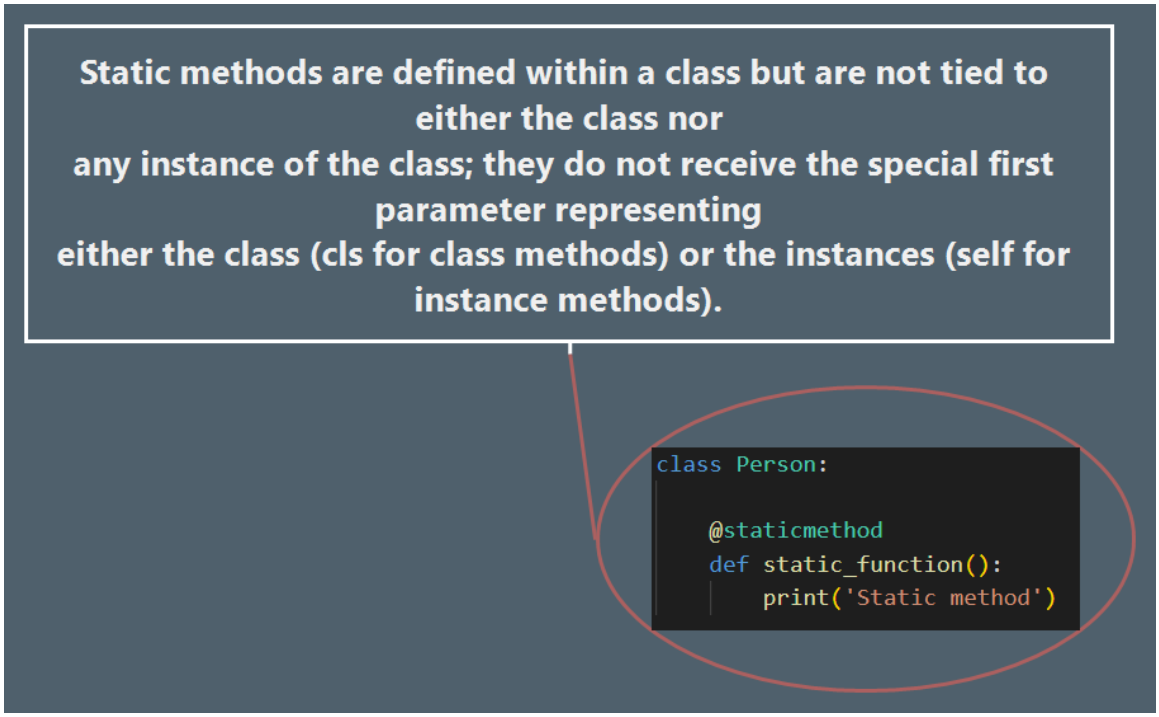
4.1. What it is



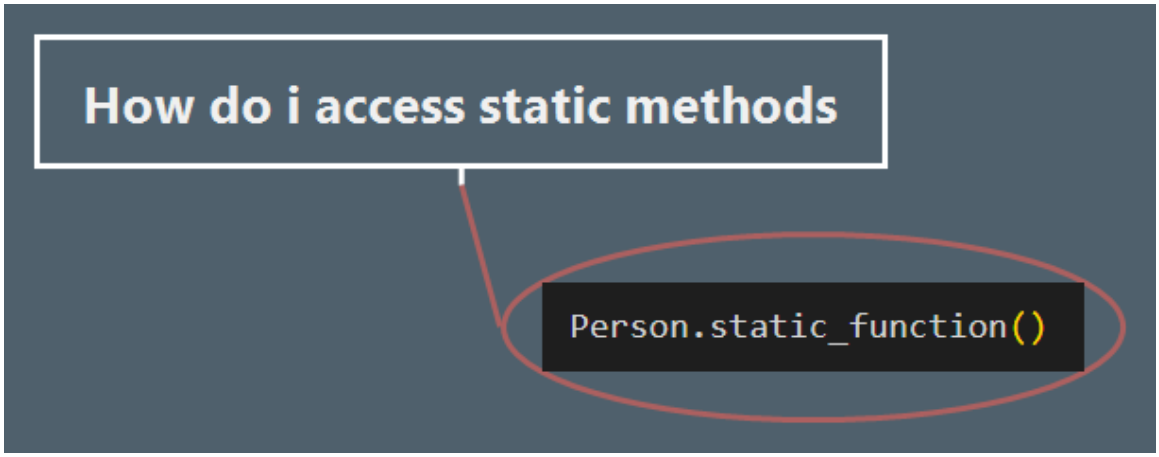
4.1.1. Static methods are defined within a class but are not tied to either the class nor

any instance of the class; they do not receive the special first parameter representing

either the class (cls for class methods) or the instances (self for instance methods).

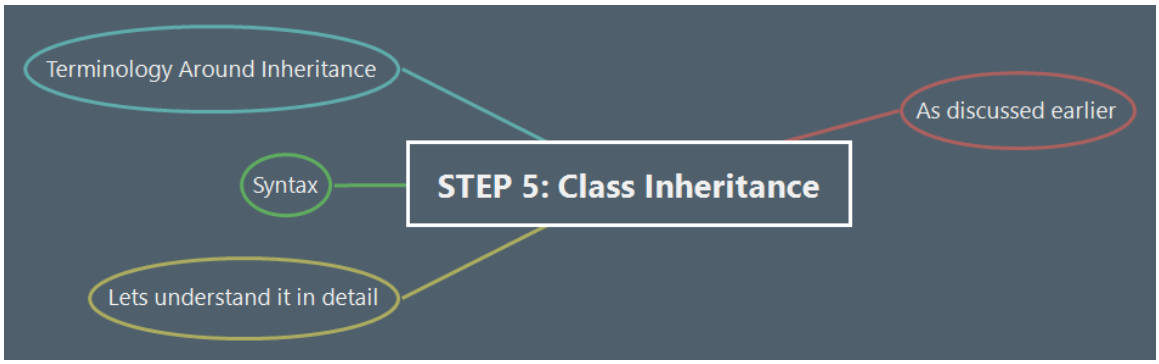


4.2. How do i access static methods

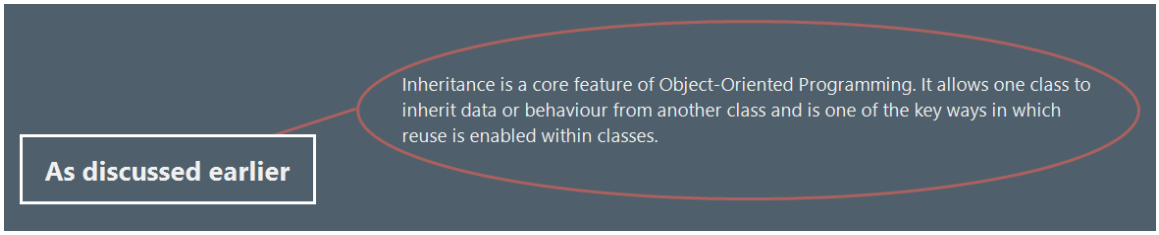


4.2.1.

5. STEP 5: Class Inheritance



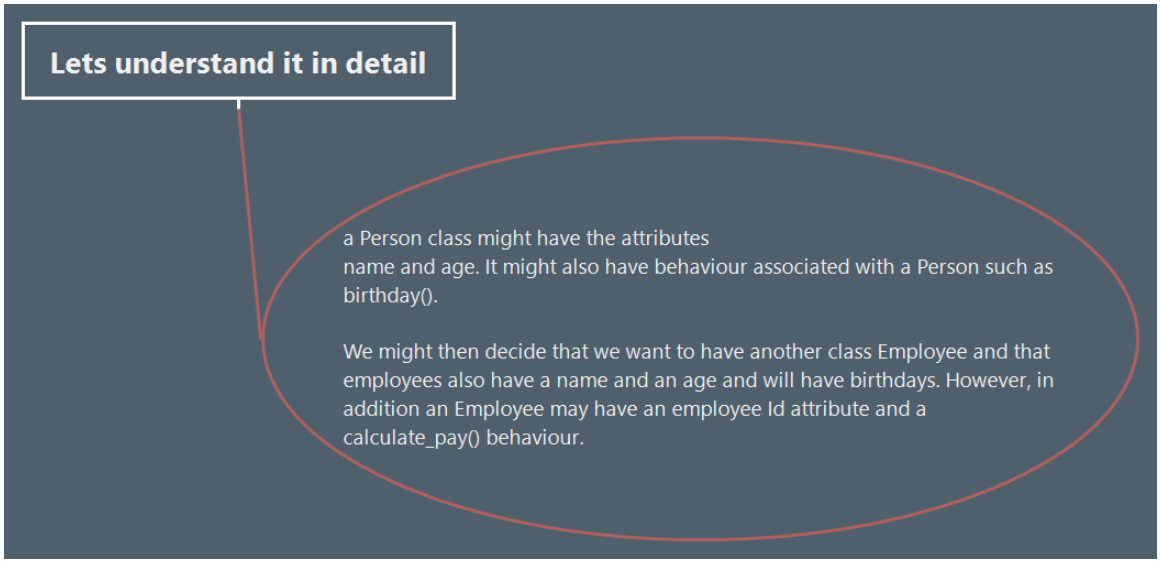
5.1. As discussed earlier



5.1.1. Inheritance is a core feature of Object-Oriented Programming. It allows one class to

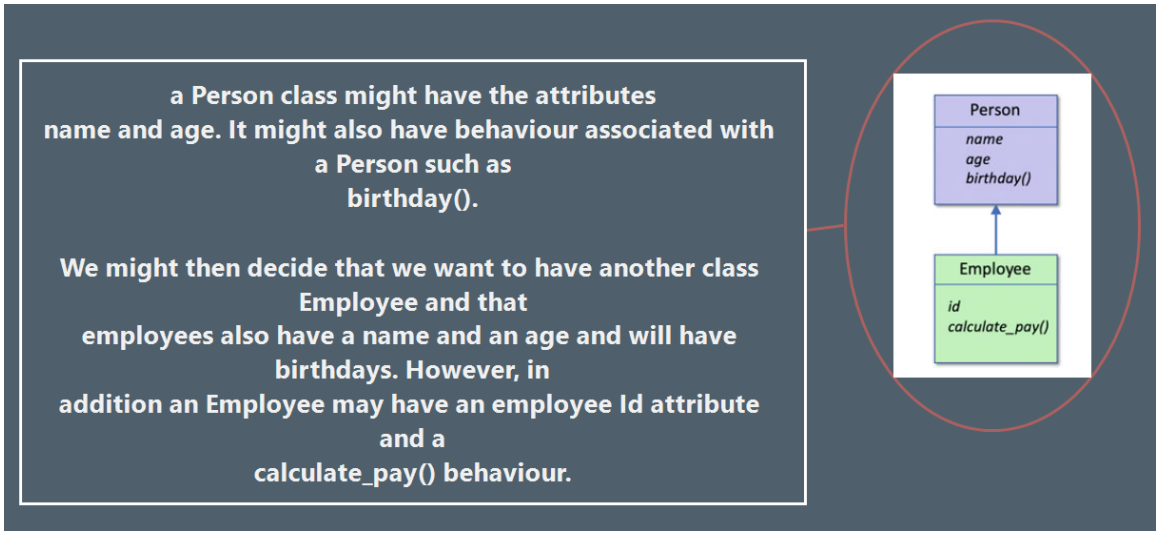
inherit data or behaviour from another class and is one of the key ways in which reuse is enabled within classes.

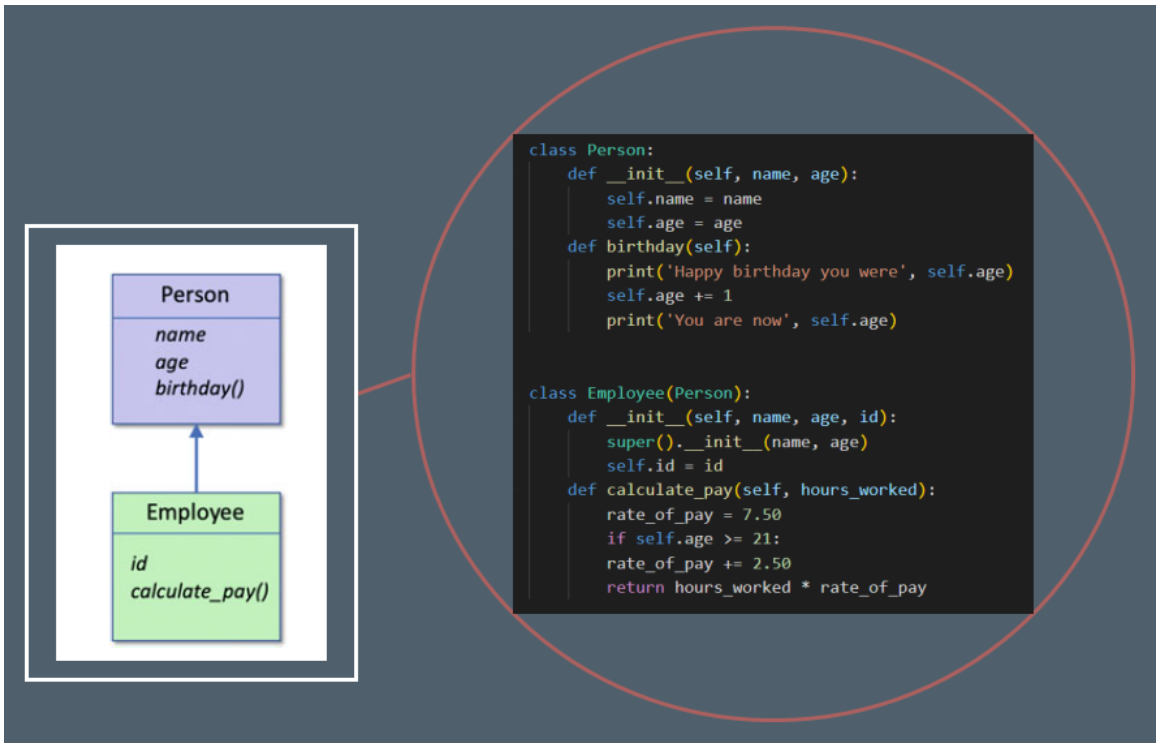
5.2. Lets understand it in detail

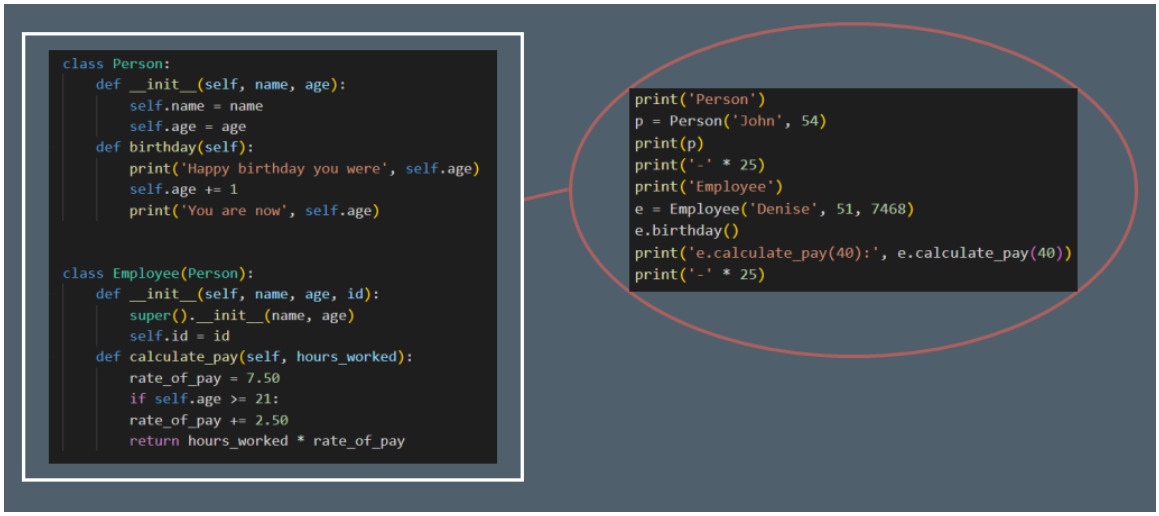


5.2.1. a Person class might have the attributes
name and age. It might also have behaviour associated with a Person such as birthday().

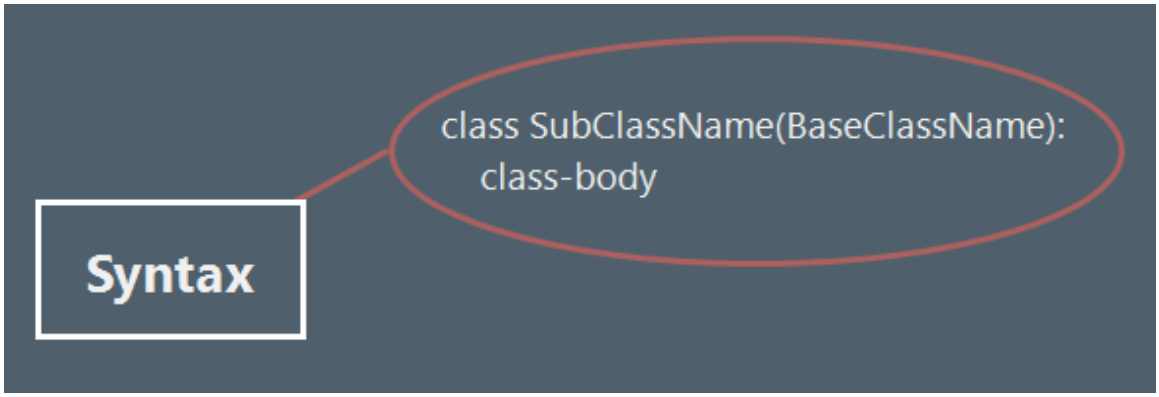
We might then decide that we want to have another class Employee and that employees also have a name and an age and will have birthdays. However, in addition an Employee may have an employee Id attribute and a calculate_pay() behaviour.




Person
name
age
birthday()
Employee
id
calculate_pay()
class Person:
def __init__(self, name, age):
self.name = name
self.age = age
def birthday(self):
print('Happy birthday you were', self.age)
self.age += 1
print('You are now', self.age)
class Employee(Person):
def __init__(self, name, age, id):
super().__init__(name, age)
self.id = id
def calculate_pay(self, hours_worked):
rate_of_pay = 7.50
if self.age >= 21:
rate_of_pay += 2.50
return hours_worked * rate_of_pay


class Person:
def __init__(self, name, age):
self.name = name
self.age = age
def birthday(self):
print('Happy birthday you were', self.age)
self.age += 1
print('You are now', self.age)
class Employee(Person):
def __init__(self, name, age, id):
super().__init__(name, age)
self.id = id
def calculate_pay(self, hours_worked):
rate_of_pay = 7.50
if self.age >= 21:
rate_of_pay += 2.50
return hours_worked * rate_of_pay
print('Person')
p = Person('John', 54)
print(p)
print('-' * 25)
print('Employee')
e = Employee('Denise', 51, 7468)
e.birthday()
print('e.calculate_pay(40):', e.calculate_pay(40))
print('-' * 25)

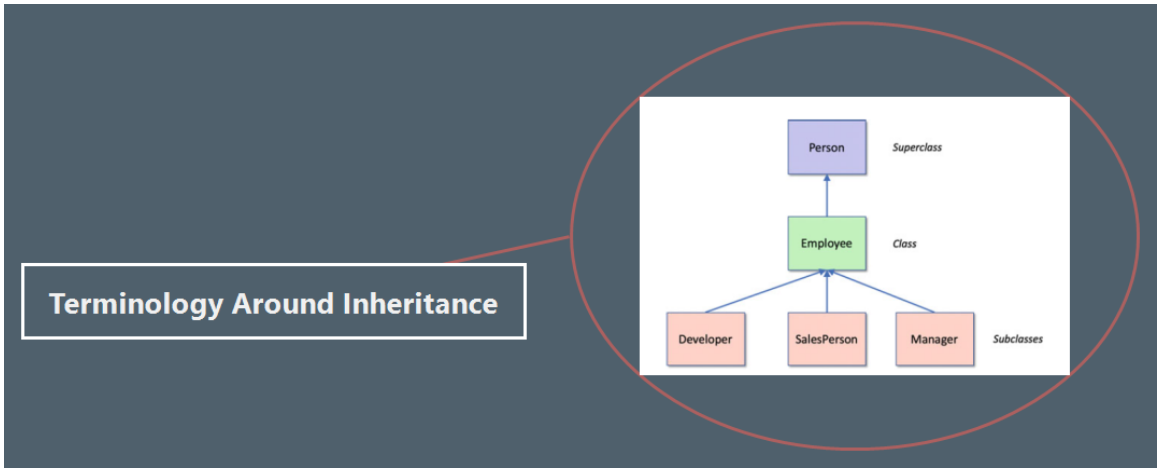
5.3. Syntax



5.3.1. class SubClassName(BaseClassName):

class-body

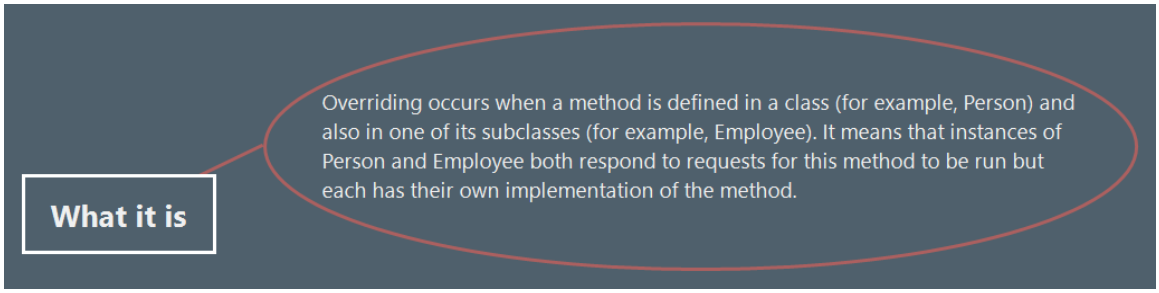
5.4. Terminology Around Inheritance



5.4.1.

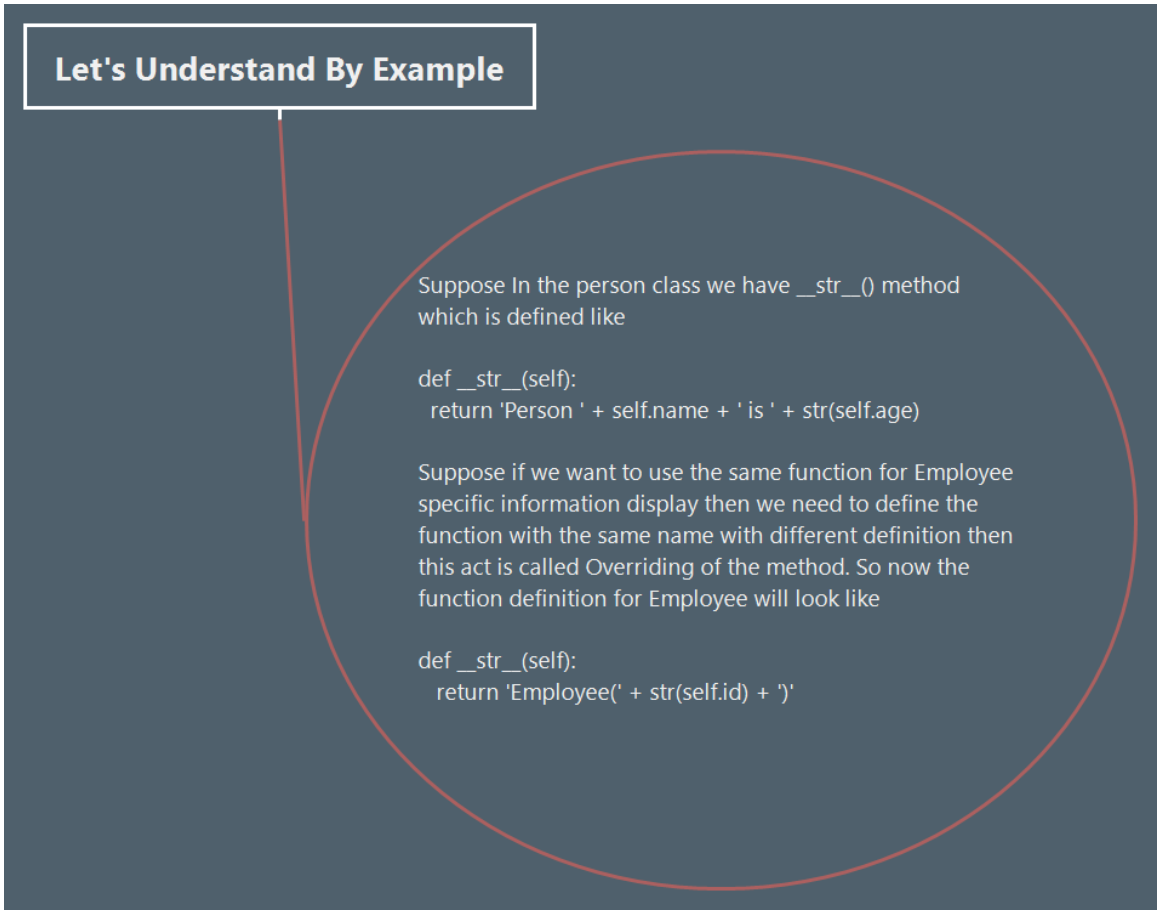
6. STEP 6: Overriding Methods

6.1. What it is



6.1.1. Overriding occurs when a method is defined in a class (for example, Person) and
also in one of its subclasses (for example, Employee). It means that instances of Person and Employee both respond to requests for this method to be run but each has their own implementation of the method.

6.2. Let's Understand By Example

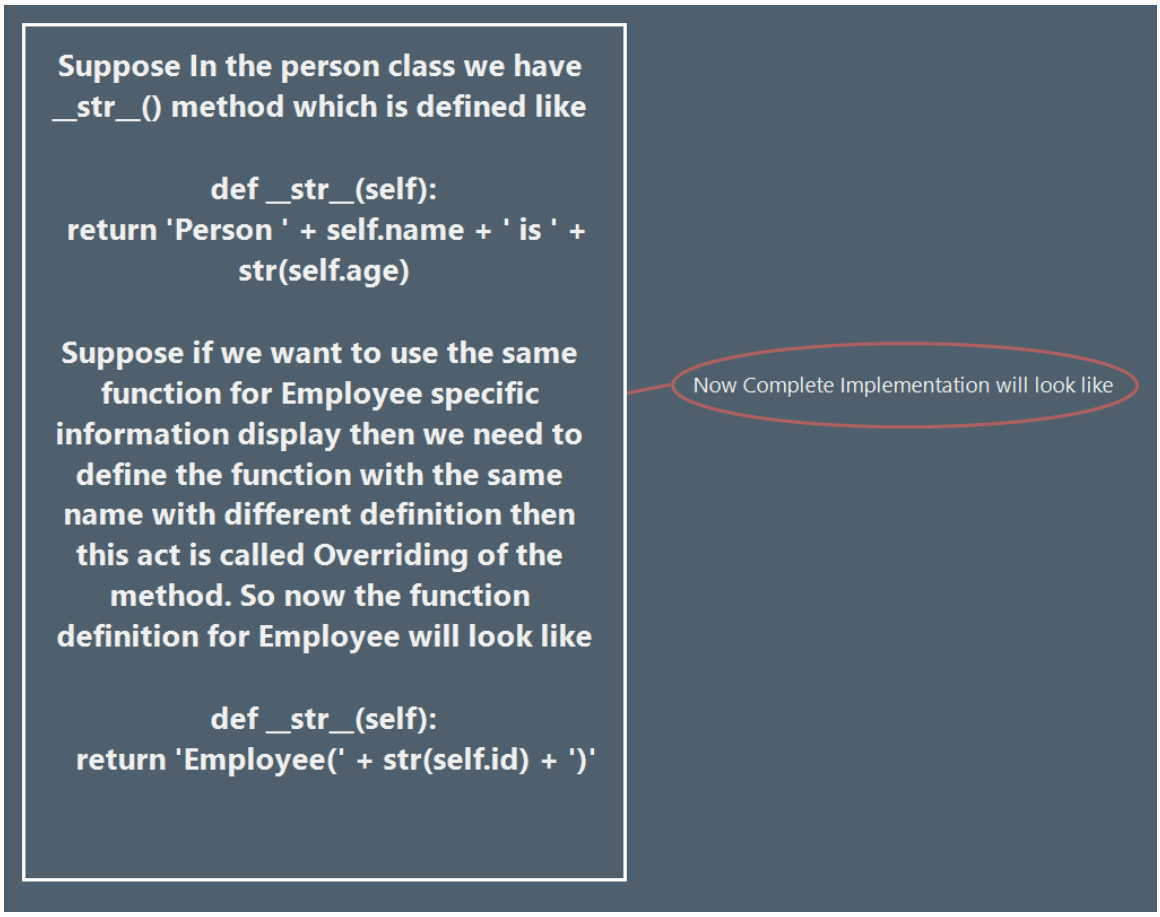


6.2.1. Suppose In the person class we have __str__() method which is defined like

```
def __str__(self):
  return 'Person ' + self.name + ' is ' + str(self.age)
```

Suppose if we want to use the same function for Employee specific information display then we need to define the function with the same name with different definition then this act is called Overriding of the method. So now the function definition for Employee will look like

```
def __str__(self):
   return 'Employee(' + str(self.id) + ')'
```



Now Complete Implementation will look like

Now Complete Implementation will look like

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name + ' is ' + str(self.age)

class Employee(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.id = id
    def __str__(self):
        return self.name + ' is ' + str(self.age) + ' - i
        str(self.id) + ')'
```

```
p = Person('John', 54)
print(p)
e = Employee('Denise', 51, 1234)
print(e)
```

```
p = Person('John', 54)
print(p)
e = Employee('Denise', 51, 1234)
print(e)
```

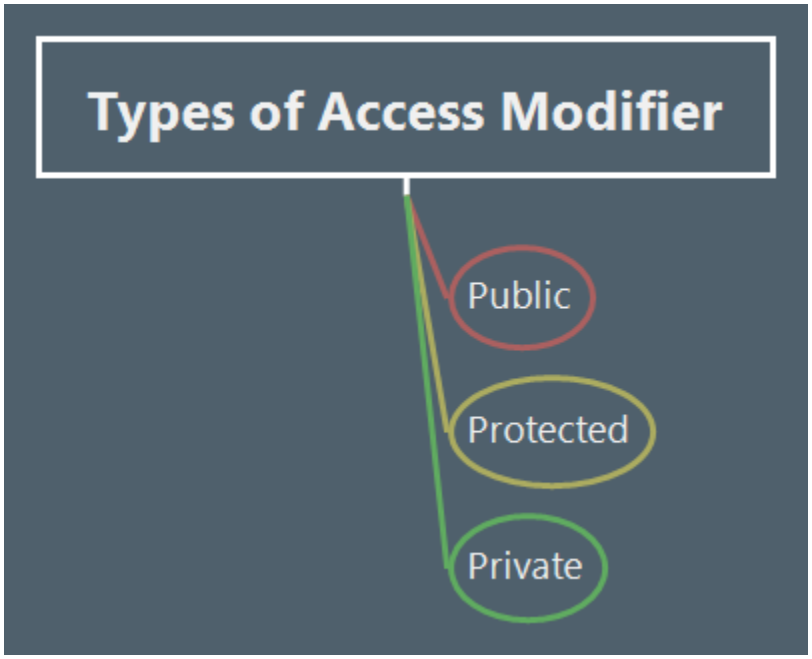
Output will look like

```
John is 54
Denise is 51 - id(1234)
```

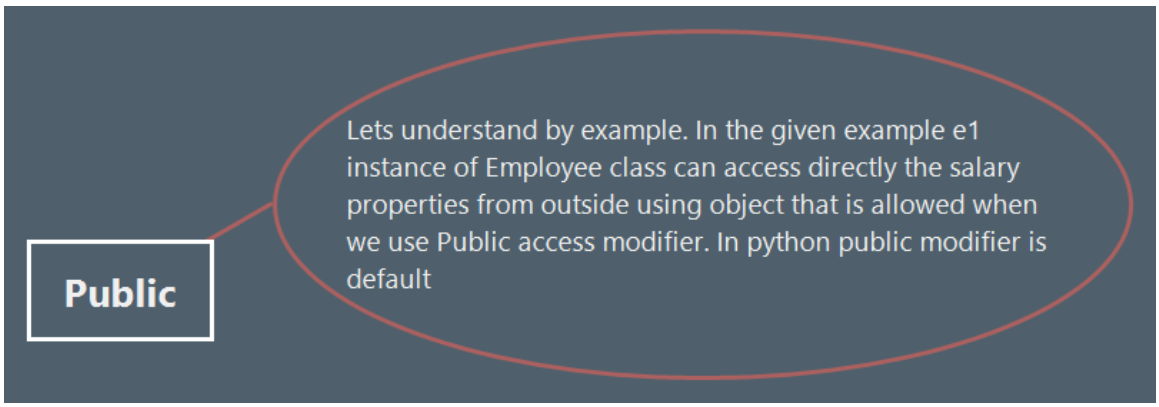
7. STEP 7: Access Modifiers

7.1. What it is

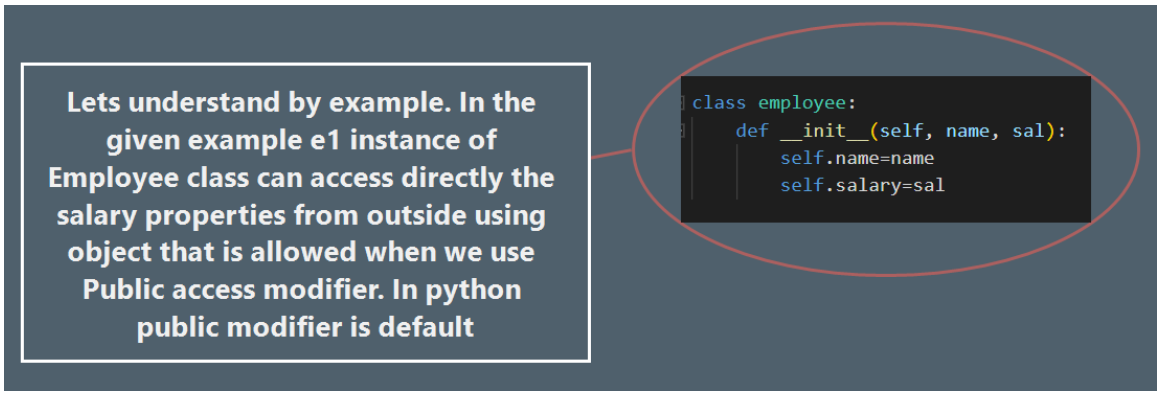
7.2. Types of Access Modifier

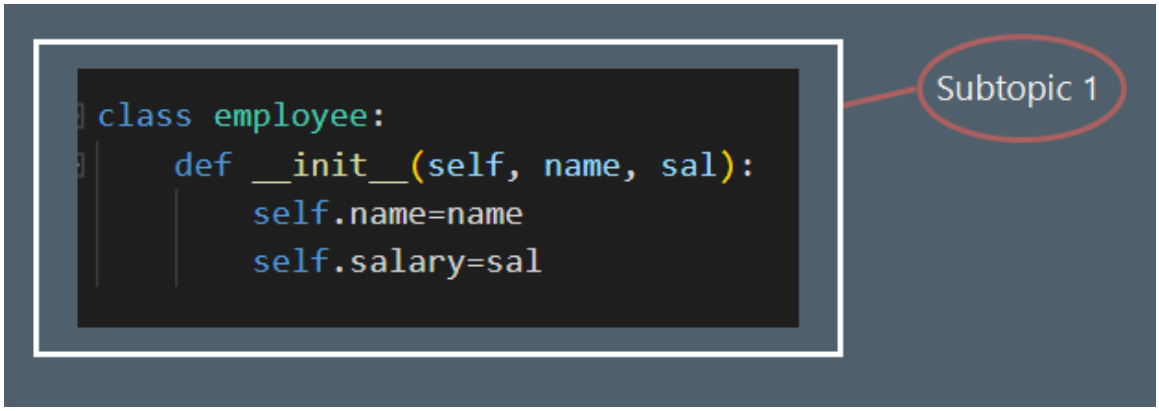


7.2.1. Public

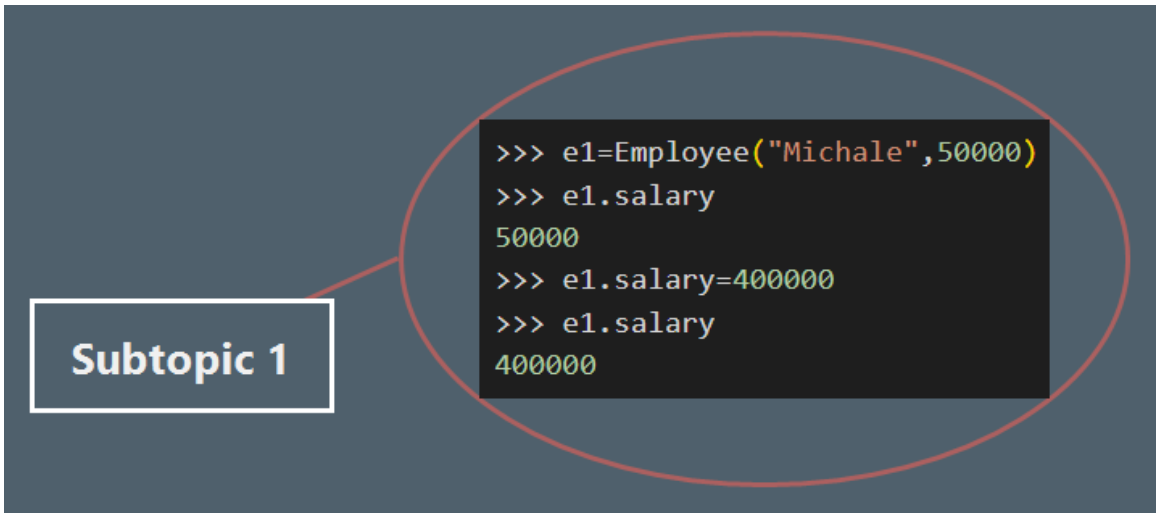


Lets understand by example. In the given example e1 instance of Employee class can access directly the salary properties from outside using object that is allowed when we use Public access modifier. In python public modifier is default

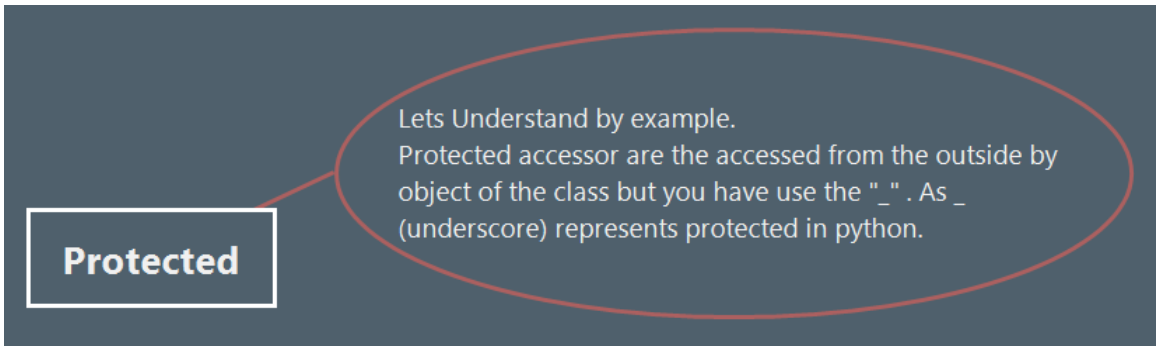




Subtopic 1

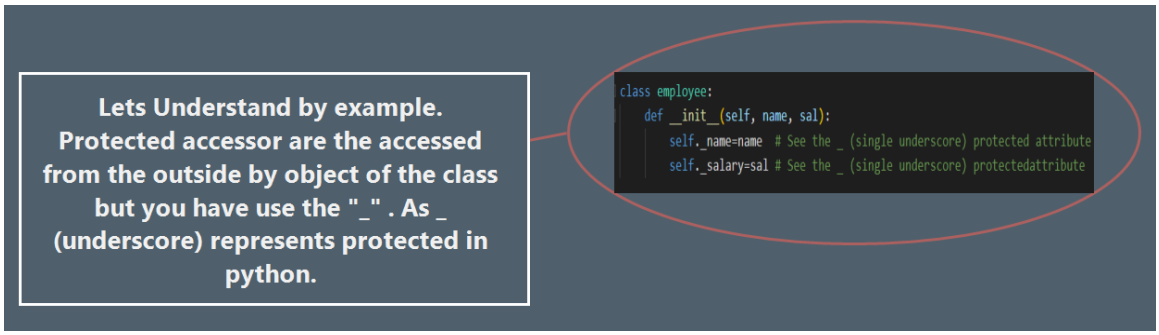


7.2.2. Protected

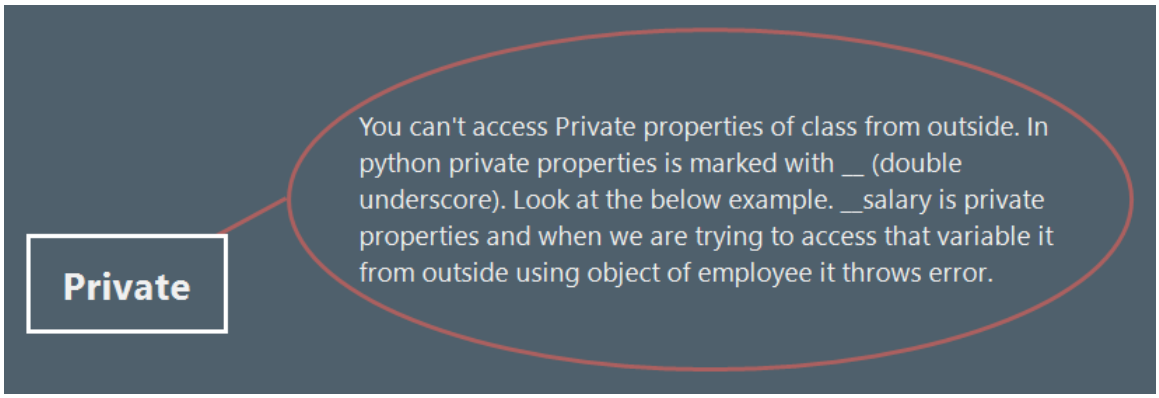


Lets Understand by example.

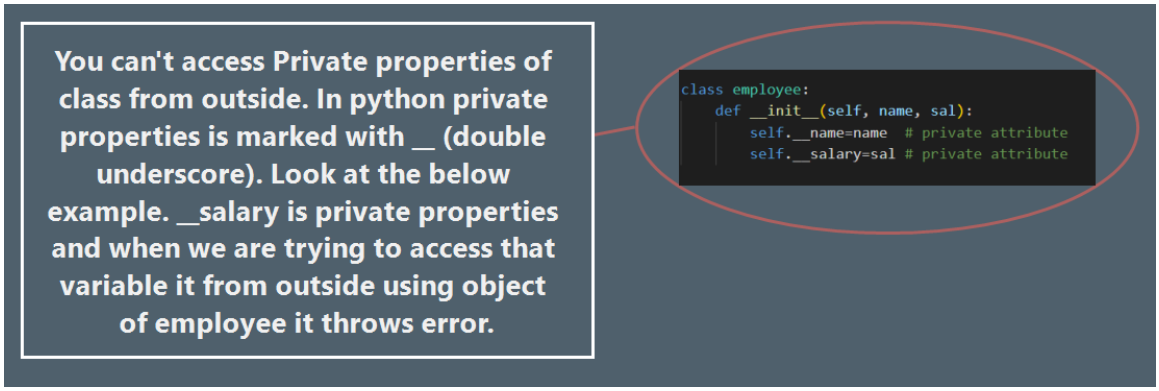
Protected accessor are the accessed from the outside by object of the class but you have use the "_" . As _ (underscore) represents protected in python.

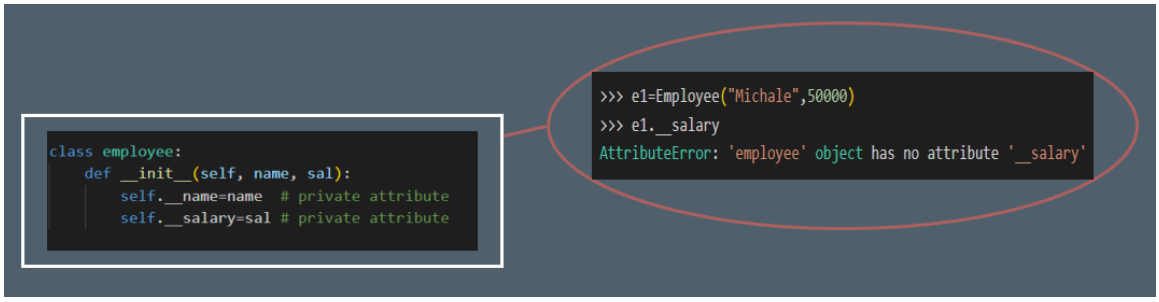


7.2.3. Private

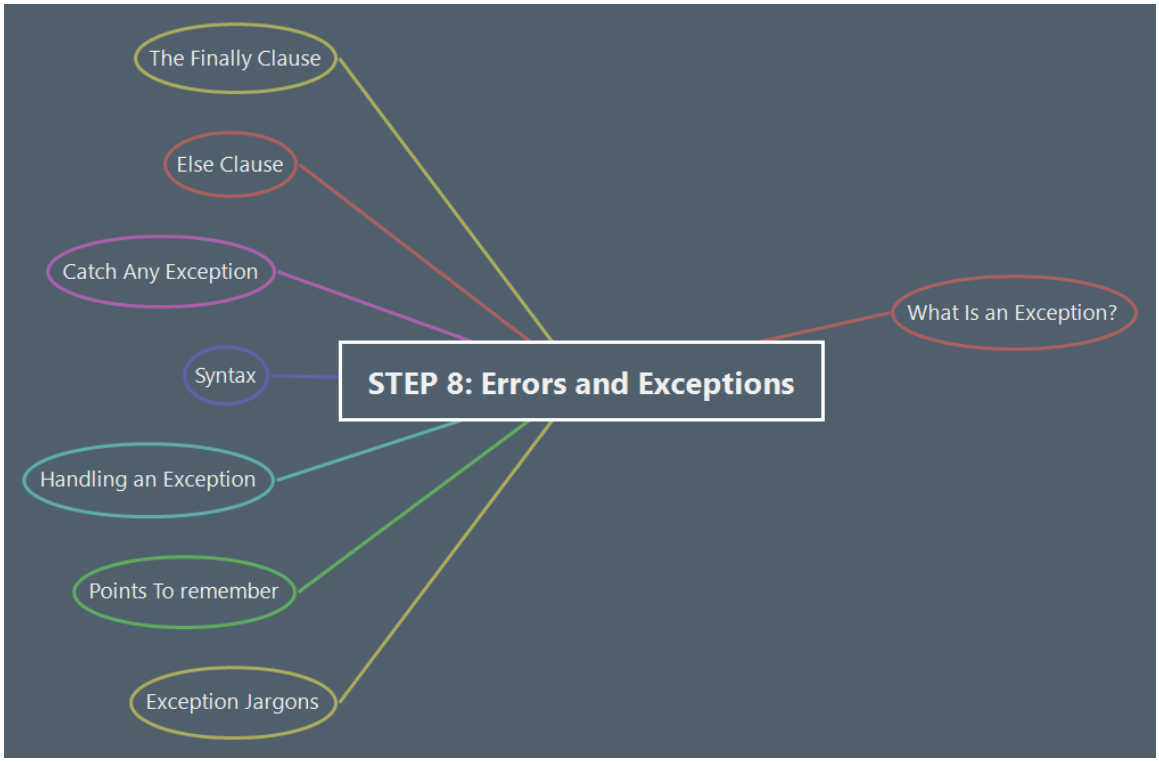


You can't access Private properties of class from outside. In python private properties is marked with __ (double underscore). Look at the below example. __salary is private properties and when we are trying to access that variable it from outside using object of employee it throws error.

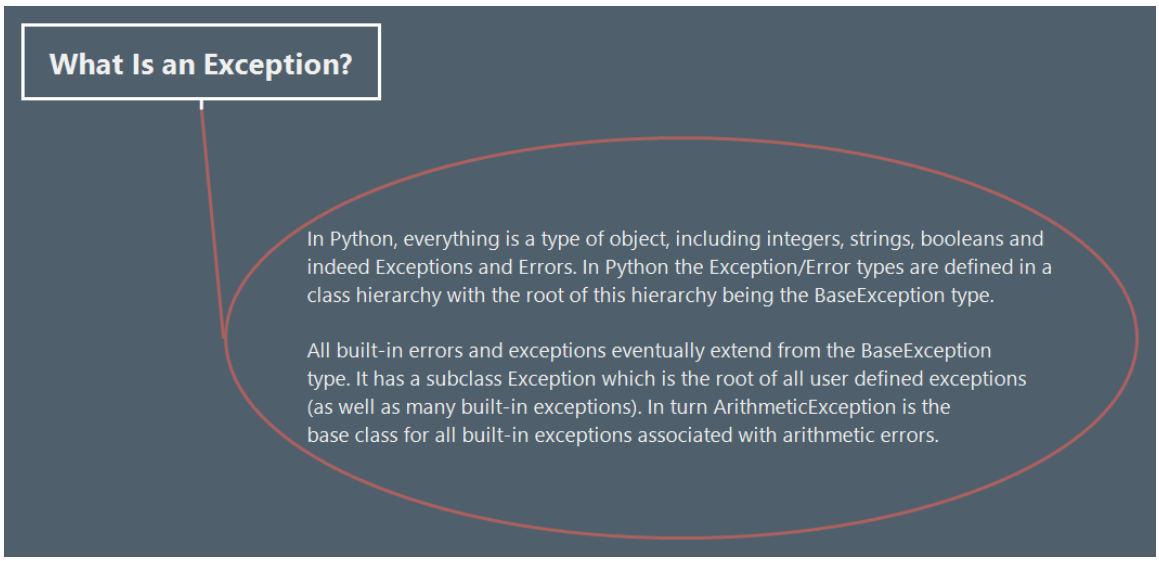




8. STEP 8: Errors and Exceptions



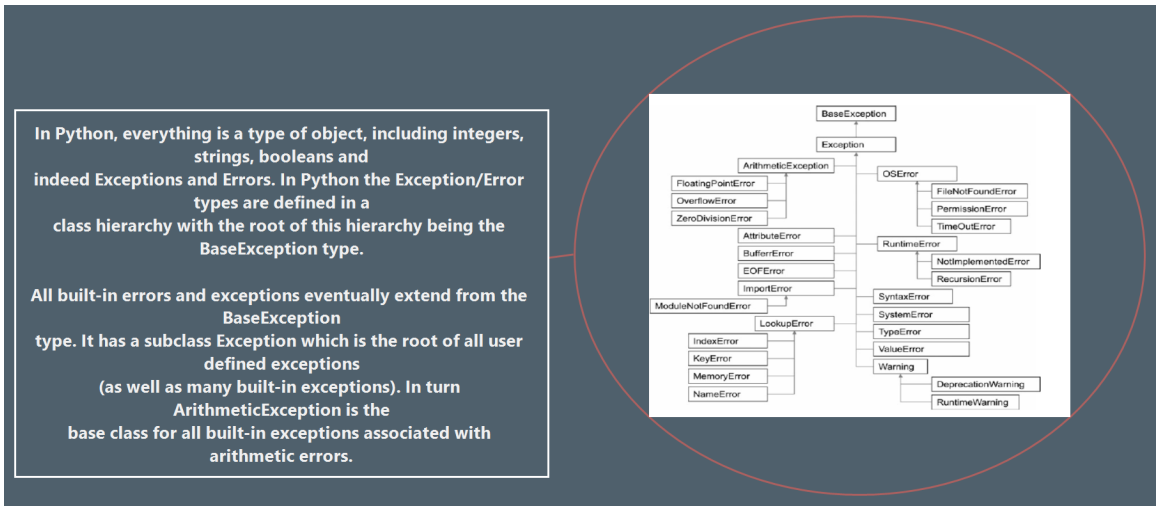
8.1. What Is an Exception?



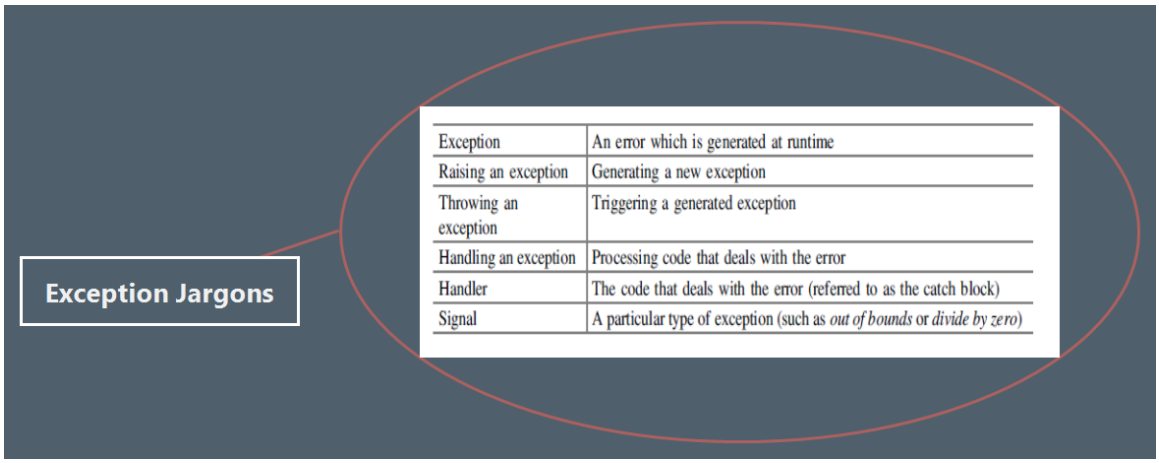
8.1.1. In Python, everything is a type of object, including integers, strings, booleans and

indeed Exceptions and Errors. In Python the Exception/Error types are defined in a class hierarchy with the root of this hierarchy being the BaseException type.

All built-in errors and exceptions eventually extend from the BaseException type. It has a subclass Exception which is the root of all user defined exceptions (as well as many built-in exceptions). In turn ArithmeticException is the base class for all built-in exceptions associated with arithmetic errors.

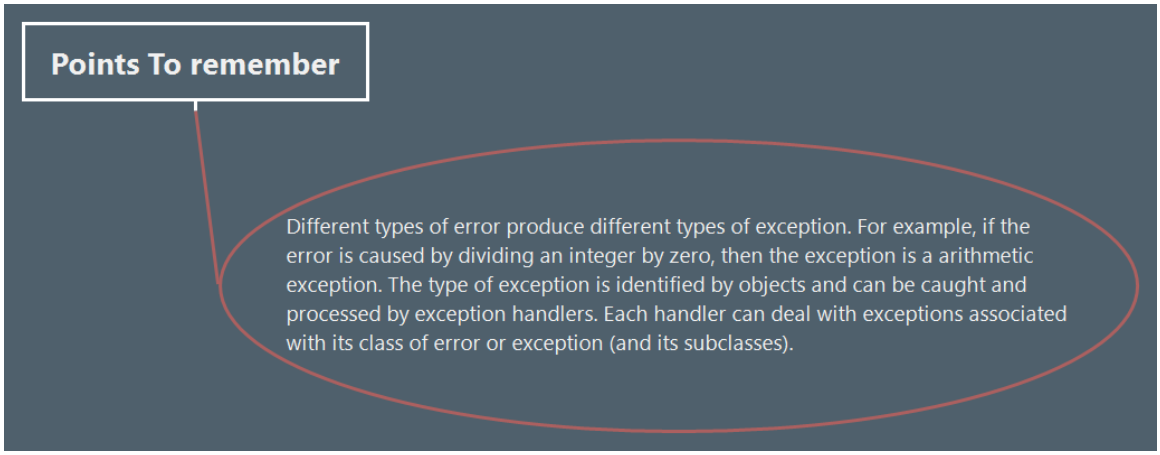


8.2. Exception Jargons

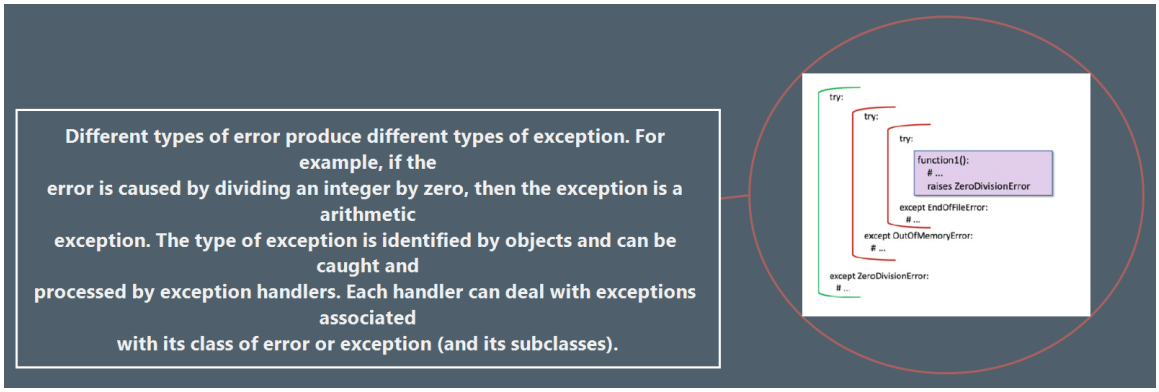


8.2.1.

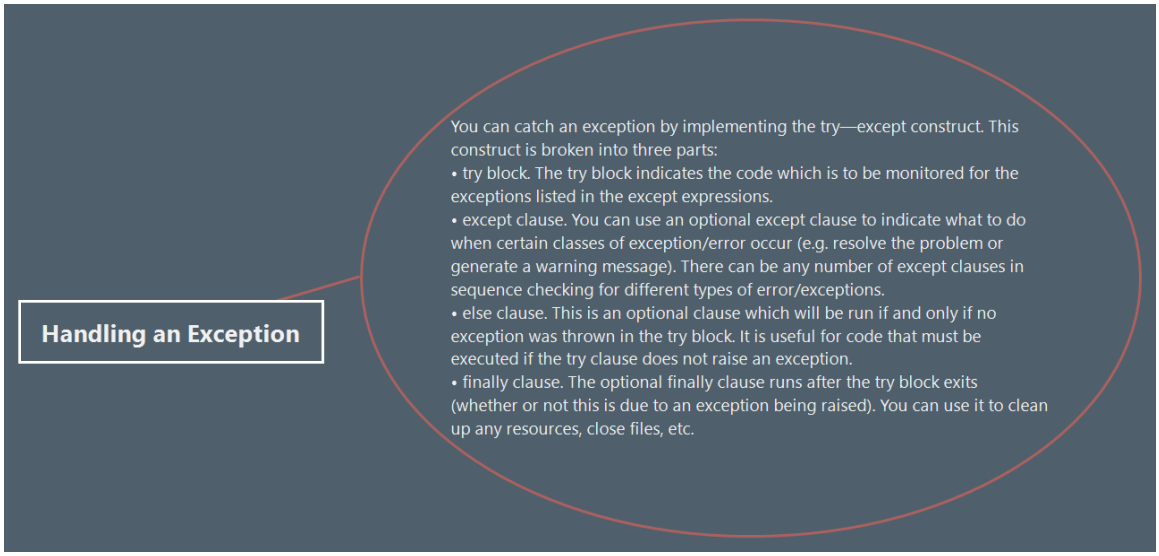
8.3. Points To remember



8.3.1. Different types of error produce different types of exception. For example, if the
error is caused by dividing an integer by zero, then the exception is a arithmetic exception. The type of exception is identified by objects and can be caught and processed by exception handlers. Each handler can deal with exceptions associated
with its class of error or exception (and its subclasses).



8.4. Handling an Exception



8.4.1. You can catch an exception by implementing the try—except construct. This construct is broken into three parts:

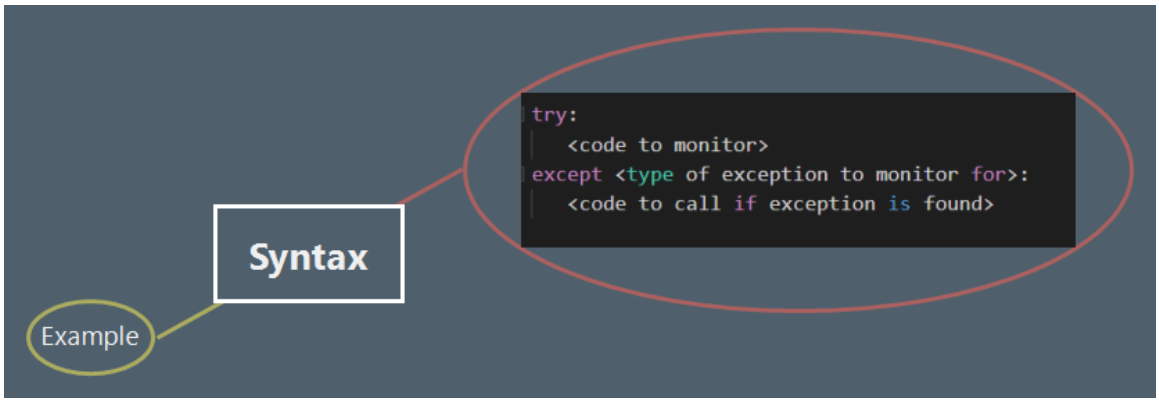
• try block. The try block indicates the code which is to be monitored for the exceptions listed in the except expressions.

• except clause. You can use an optional except clause to indicate what to do when certain classes of exception/error occur (e.g. resolve the problem or generate a warning message). There can be any number of except clauses in sequence checking for different types of error/exceptions.

• else clause. This is an optional clause which will be run if and only if no exception was thrown in the try block. It is useful for code that must be executed if the try clause does not raise an exception.

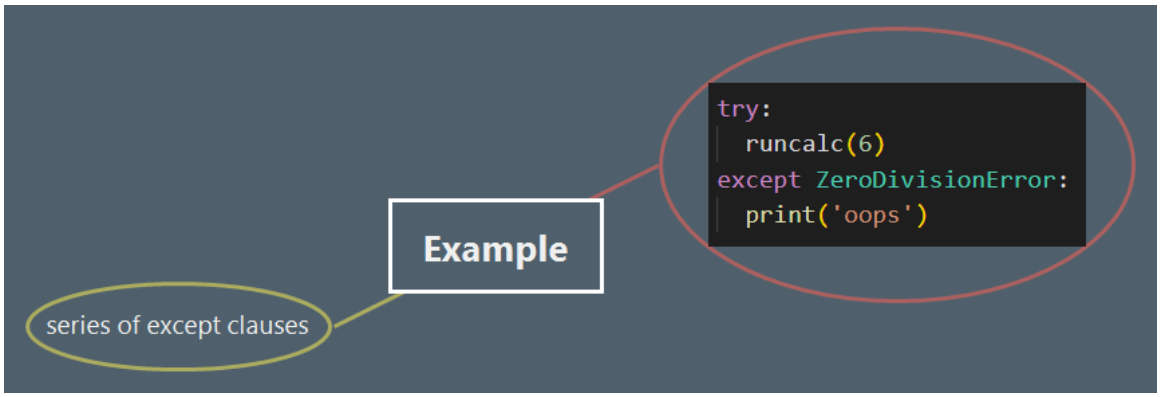
• finally clause. The optional finally clause runs after the try block exits (whether or not this is due to an exception being raised). You can use it to clean up any resources, close files, etc.

8.5. Syntax

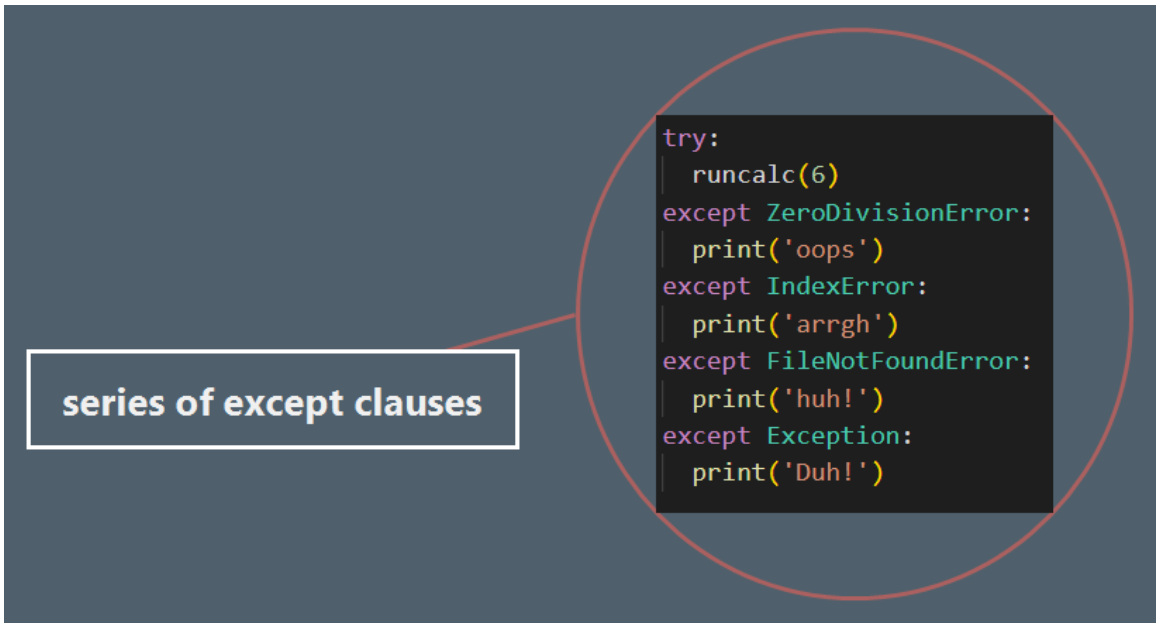


8.5.1.

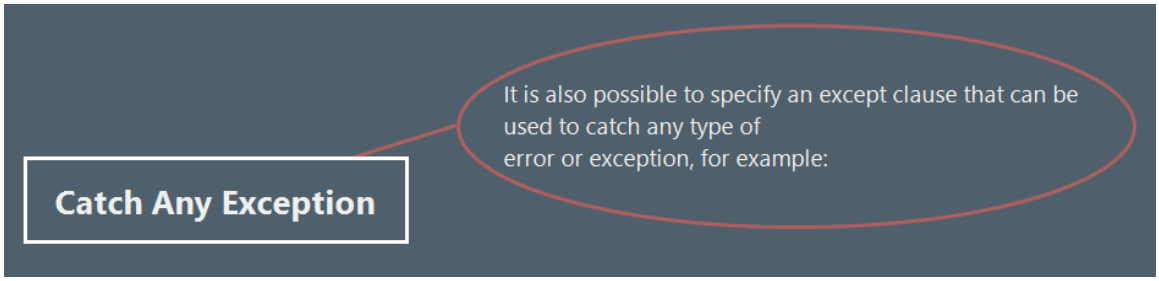
8.5.2. Example



series of except clauses

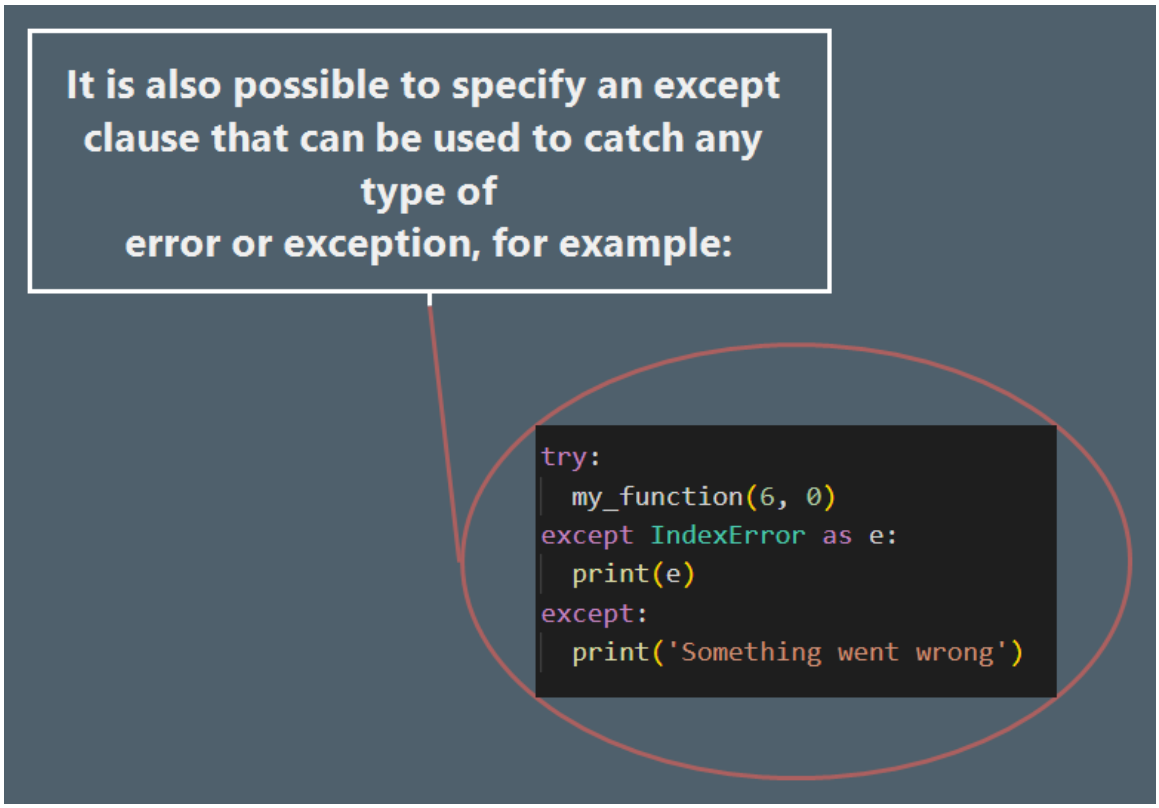


8.6. Catch Any Exception

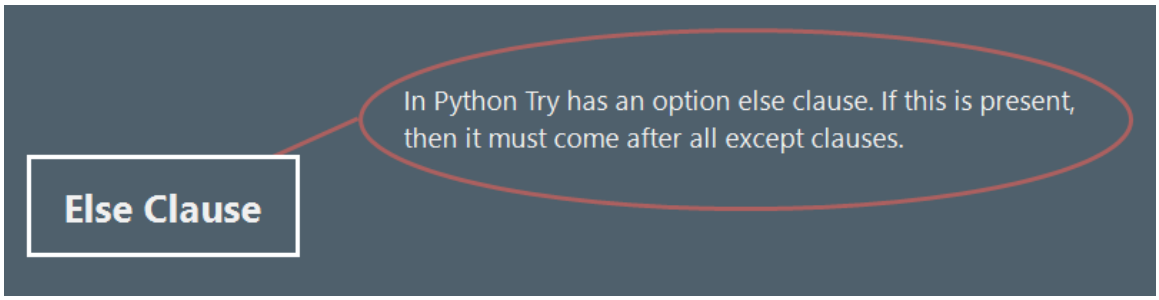


8.6.1. It is also possible to specify an except clause that can be used to catch any type of

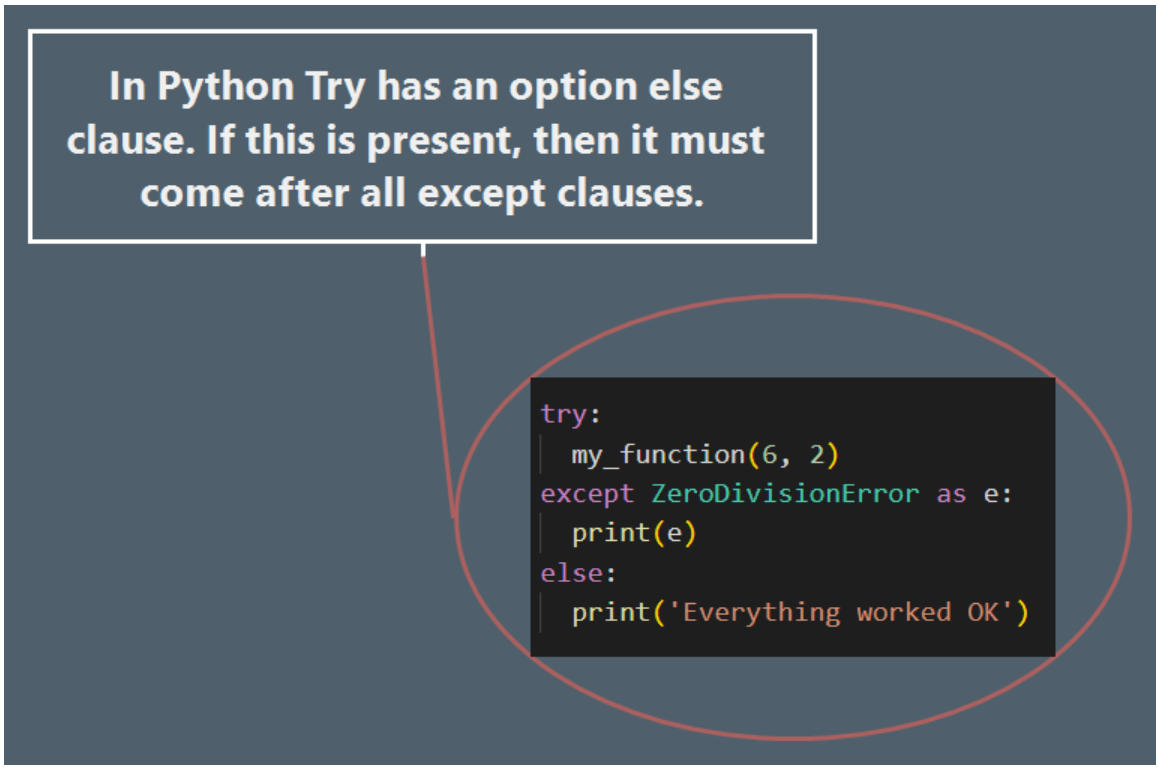
error or exception, for example:



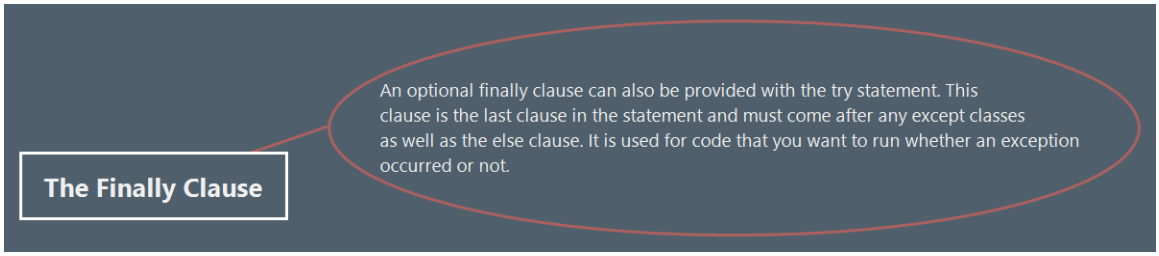
8.7. Else Clause



8.7.1. In Python Try has an option else clause. If this is present, then it must come after all except clauses.

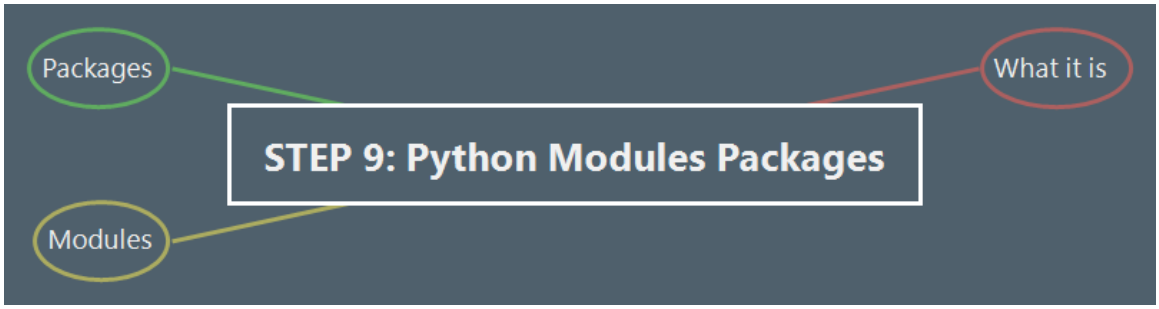


8.8. The Finally Clause

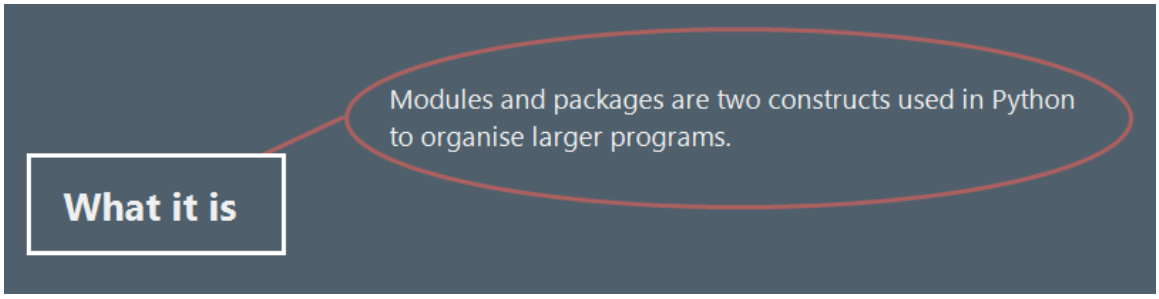


8.8.1. An optional finally clause can also be provided with the try statement. This clause is the last clause in the statement and must come after any except classes as well as the else clause. It is used for code that you want to run whether an exception occurred or not.

9. STEP 9: Python Modules Packages

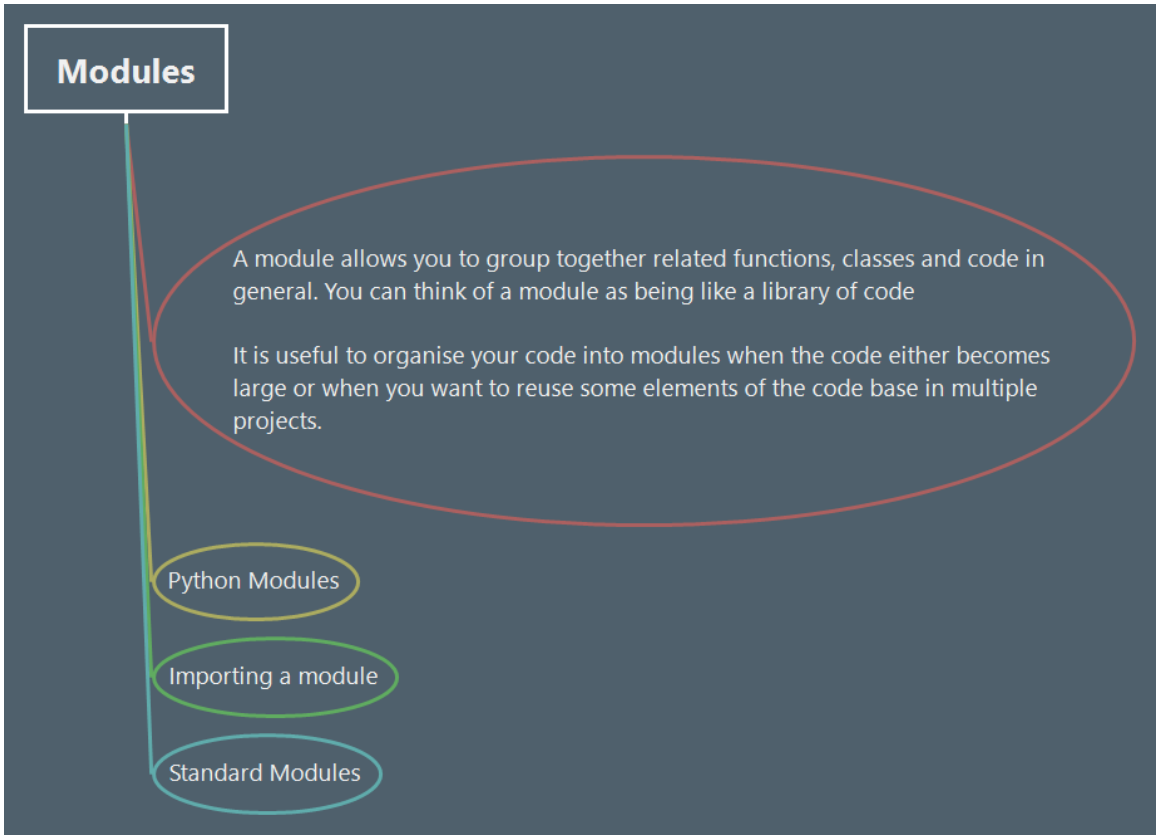


9.1. What it is



9.1.1. Modules and packages are two constructs used in Python to organise larger programs.

9.2. Modules

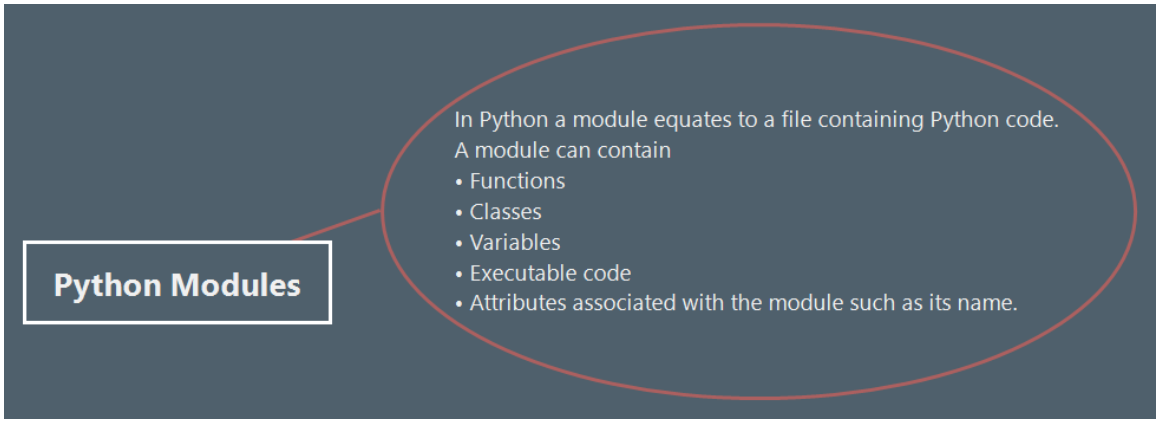


9.2.1. A module allows you to group together related functions, classes and code in

general. You can think of a module as being like a library of code

It is useful to organise your code into modules when the code either becomes large or when you want to reuse some elements of the code base in multiple projects.

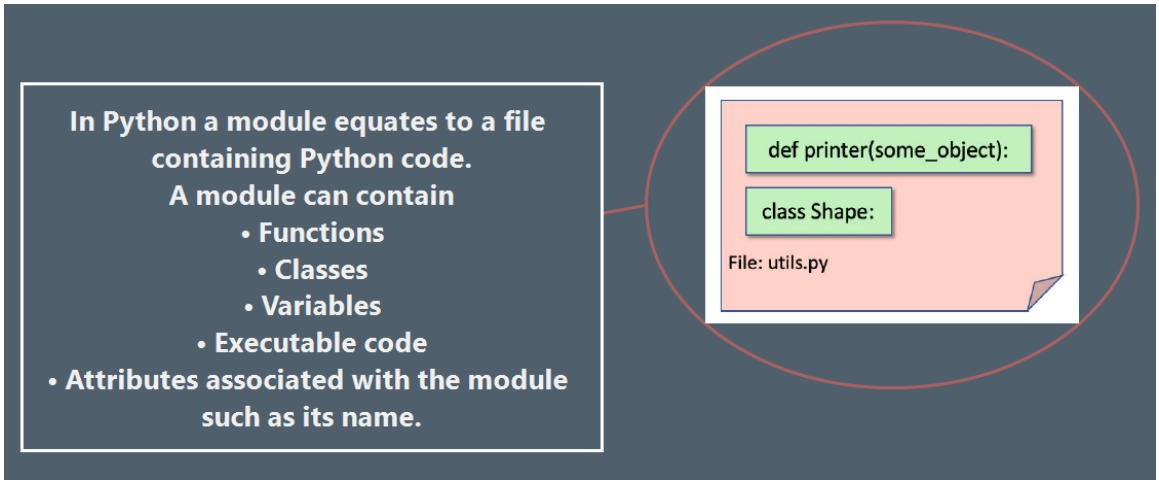
9.2.2. Python Modules



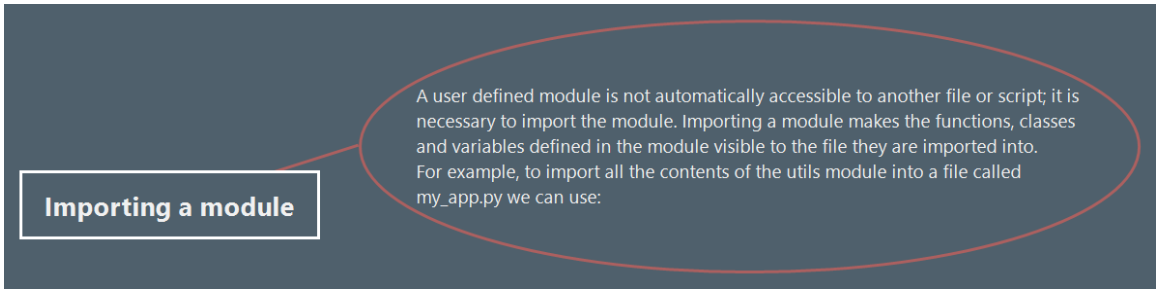
In Python a module equates to a file containing Python code.

A module can contain

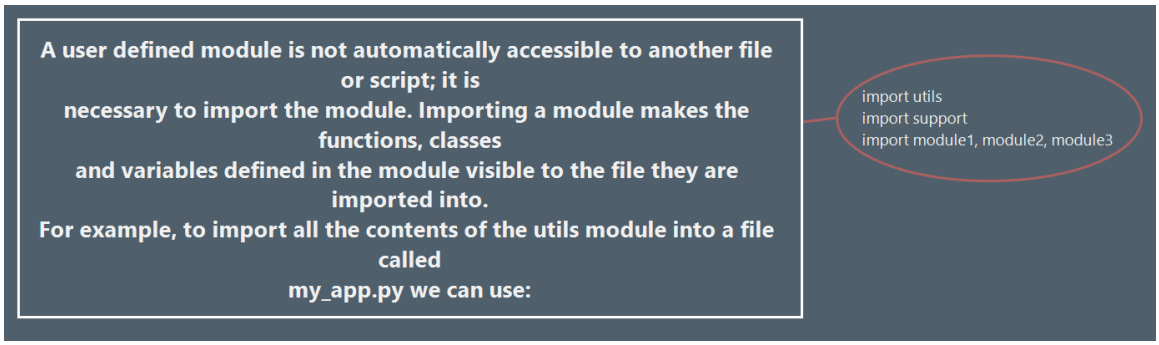
- **Functions**
- **Classes**
- **Variables**
- **Executable code**
- **Attributes associated with the module such as its name.**



9.2.3. Importing a module

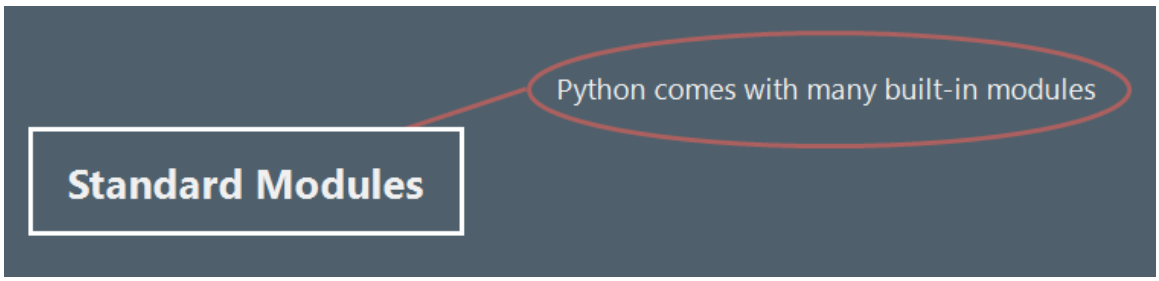


A user defined module is not automatically accessible to another file or script; it is
necessary to import the module. Importing a module makes the functions, classes
and variables defined in the module visible to the file they are imported into.
For example, to import all the contents of the utils module into a file called my_app.py we can use:

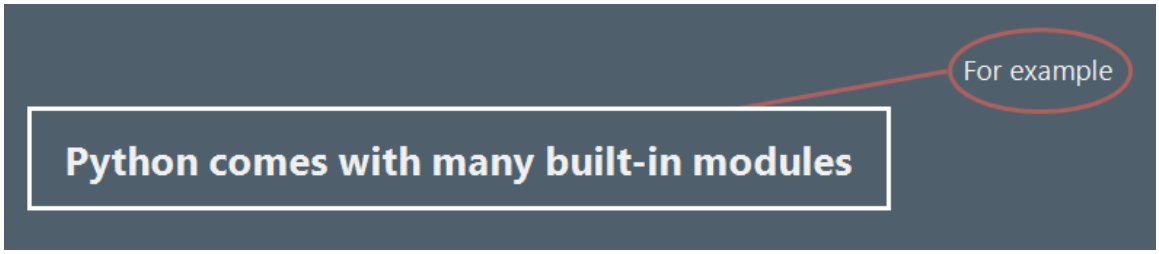



```
import utils
import support
import module1, module2, module3
```

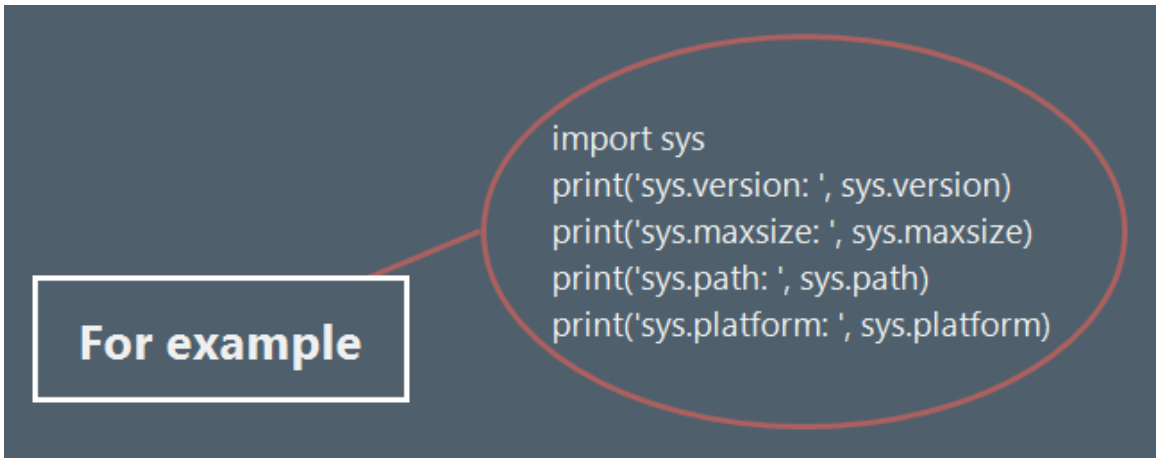
9.2.4. Standard Modules



Python comes with many built-in modules

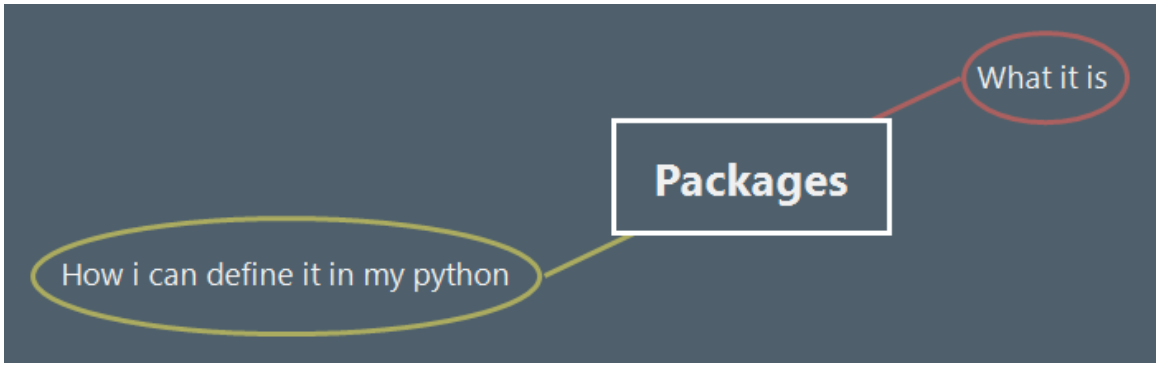


For example

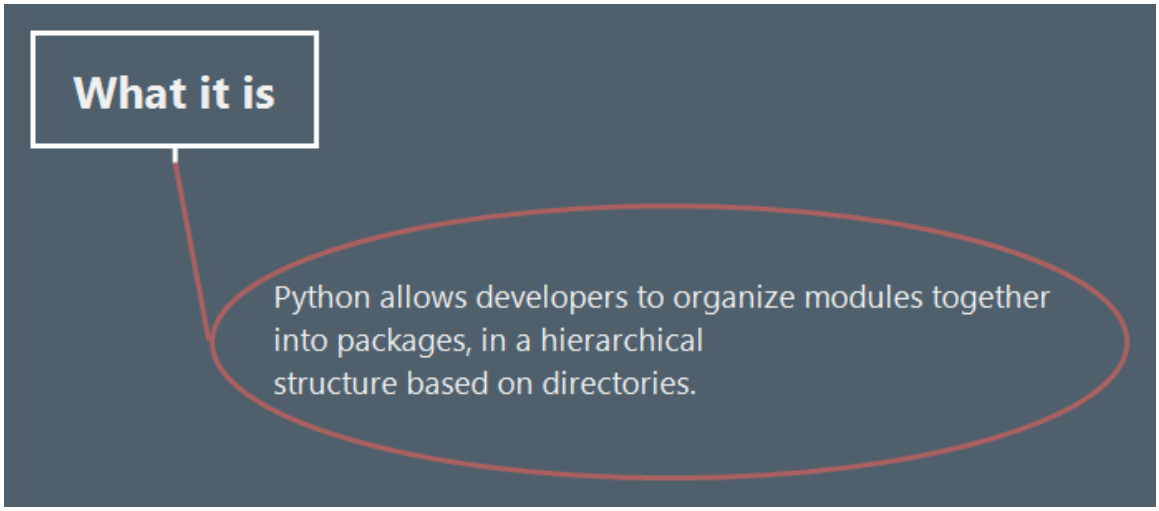



```
import sys
print('sys.version: ', sys.version)
print('sys.maxsize: ', sys.maxsize)
print('sys.path: ', sys.path)
print('sys.platform: ', sys.platform)
```

9.3. Packages



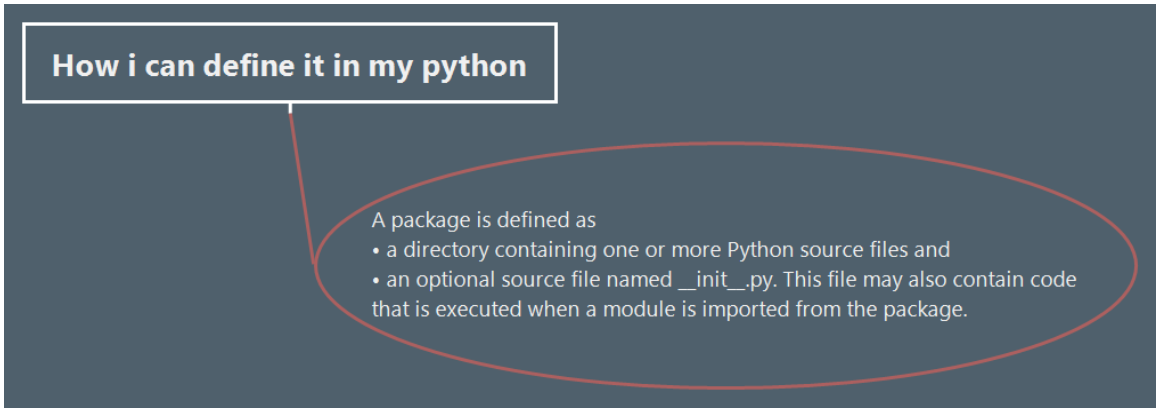
9.3.1. What it is



Python allows developers to organize modules together into packages, in a hierarchical

structure based on directories.

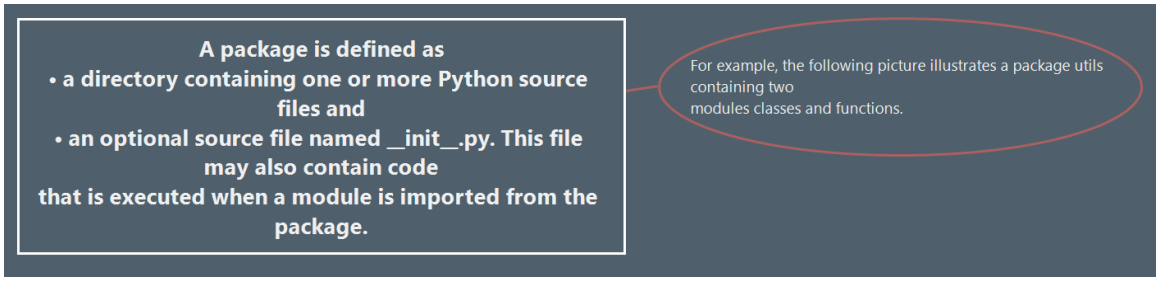
9.3.2. How i can define it in my python



A package is defined as

• a directory containing one or more Python source files and

• an optional source file named __init__.py. This file may also contain code that is executed when a module is imported from the package.



For example, the following picture illustrates a package utils containing two modules classes and functions.