**Encapsulation, Polymorphism, & More**

**Done**

**Table Of Contents**

* + [What you will learn](http://learn.di-learning.com/courses/collection/18/course/13/section/62/chapter/310#What_you_will_learn)
  + [Useful Resources](http://learn.di-learning.com/courses/collection/18/course/13/section/62/chapter/310#Useful_Resources)
    - [I . Encapsulation](http://learn.di-learning.com/courses/collection/18/course/13/section/62/chapter/310#I_._Encapsulation)
    - [II. Polymorphism](http://learn.di-learning.com/courses/collection/18/course/13/section/62/chapter/310#II._Polymorphism)
    - [III. Multiple Inheritance](http://learn.di-learning.com/courses/collection/18/course/13/section/62/chapter/310#III._Multiple_Inheritance)
* [Feedback](http://learn.di-learning.com/courses/collection/18/course/13/section/62/chapter/310#Feedback)

Last updated : May, 2nd 2021

**What You Will Learn**

* Encapsulation
* Polymorphism
* Multiple inheritances

**Useful Resources**

* [Article on Encapsulation](https://medium.com/@kateolenya/encapsulation-in-python-3-1a93fd8fa9cd)

**I . Encapsulation**

We can restrict access to methods and variables. This prevents data from direct modification, which is called **encapsulation**. In Python, we denote private attributes using underscore as prefix i.e., single \_ or double \_\_.

Lets define an object class Computer and try to access its variables and methods both private and global.

class Computer():

def \_\_init\_\_(self):

self.name = "Apple Computer" # public

self.\_\_max\_price = 900 # private

def sell(self): # public method

print(f"Selling Price: {self.\_\_max\_price}")

def \_\_sell(self): # private method

print('This is private method')

def set\_max\_price(self, price):

self.\_\_max\_price = price

c = Computer()

**Note**: So once we try to access attributes and methods of our class - we should receive a restriction if they are private.

c.sell()

# >> Selling Price: 900

Explanation of the output Selling Price: 900:  
the\_\_max\_price which is a private attribute, is displayed by using the public sell() method.

c.\_\_sell()

# >> AttributeError: 'Computer' object has no attribute '\_\_sell'

Explanation of the output AttributeError: 'Computer' object has no attribute '\_\_sell':  
The interpreter cannot perform the \_\_sell() function because of the \_\_ underscore meaning it’s private and can’t be accessed by the user.

# change the price

c.\_\_max\_price = 1000

c.sell()

# >> The private attribute \_\_max\_price cannot be changed

# >> Selling Price: 900

# using setter function

c.set\_max\_price(1000)

c.sell()

# >> Selling Price: 1000

**II. Polymorphism**

In programming, polymorphism means different or related classes that use the same names for their functions.

Polymorphism allows the ability to use a standard interface for multiple forms or data types.

Let’s define two classes, Parrot and Penguin, both with functions called fly and swim and then we’ll call these functions with the flying\_test(bird) function.

class Parrot():

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin():

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

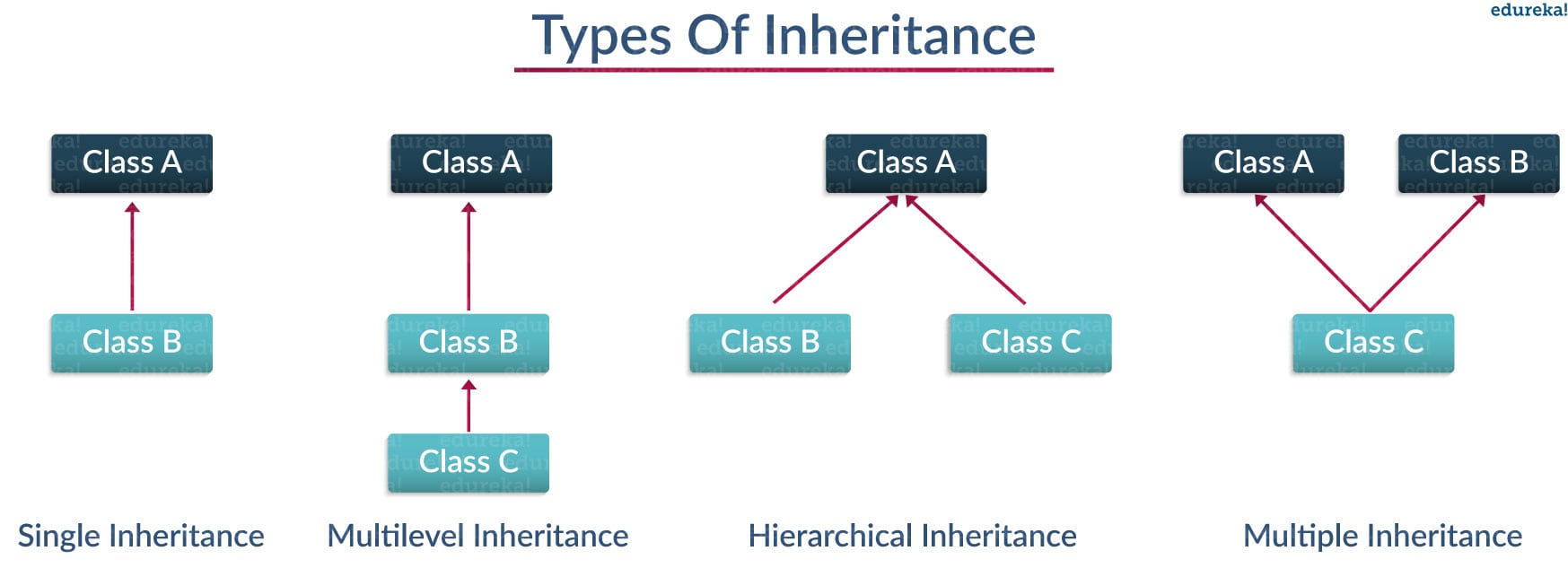
# >> Parrot can fly

flying\_test(peggy)

# >> Penguin can't fly

In the next section we learn that a child class (AlienDog) inherits all the methods from the parent class (Dog). However, in some situations, the method inherited from the parent class doesn’t quite fit into the child class (because of different attributes and characteristics). In such cases, you will have to re-implement the method in the child class.

**III. Multiple Inheritance**



A class can inherit from two different classes; in this case, the order of the parent class in a class definition will decide what will be inherited. The first parent class will prioritize classes below (meaning the functions last to be inherited can override functions from the parent class).

class Alien():

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

self.name = name

self.planet = planet

def fly(self):

print(self.name, 'is flying!')

def sleep(self):

print("Aliens don't sleep")

class Animal():

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

self.name = name

def sleep(self):

print("zzzZZZZZ")

class Dog(Animal):

def bark(self):

print("{} barked, WAF !".format(self.name))

class AlienDog(Alien, Dog):

def bark(self):

print("{} barked, 0ul10ul0u (that's how aliens dogs bark..) !".format(self.name))

my\_normal\_dog = Dog("Roger")

my\_normal\_dog.sleep()

# >> zzzZZZZZ

my\_normal\_dog.bark()

# >> Roger barked, WAF !

my\_alien\_dog = AlienDog("Rex", "Neptune")

print(my\_alien\_dog.planet)

# >> Neptune

my\_alien\_dog.fly()

# >> Rex is flying!

my\_alien\_dog.sleep()

# >> Aliens don't sleep

my\_alien\_dog.bark()

# >> Rex barked, 0ul10ul0u (that's how aliens dogs bark..) !

Here I have created two new classes, Alien and AlienDog, AlienDog inherit from Alien and Dog, but both classes have an \_\_init\_\_ and a sleep method. However, the functions of Alien will be transferred to AlienDog because Alien is before Dog in the class definition (class AlienDog(Alien, Dog)).

**Exercise**

Analyse the code below before executing it. What will be the output ? Why ?

class A():

def dothis(self):

print("doing this in A")

class B(A):

pass

class C():

def dothis(self):

print("doing this in C")

class D(B, C):

pass

d\_instance = D()

d\_instance.dothis()

**Exercise**

Analyse the code below before executing it. What will be the output ? Why ?

class Book():

def \_\_init\_\_(self, title, author, publication\_date, price):

self.title = title

self.author = author

self.publication = publication\_date

self.price = price

def present(self):

print(f'Title: {self.title}')

class Fiction(Book):

def \_\_init\_\_(self, title, author, publication\_date, price, is\_awesome):

super().\_\_init\_\_(title, author, publication\_date, price)

self.genre = 'Fiction'

self.is\_awesome = is\_awesome

if self.is\_awesome:

self.bored = False

print('Woow this is an awesome book')

else:

self.bored = True

print('I am very bored')

if \_\_name\_\_ == '\_\_main\_\_':

foundation = Fiction('Asimov', 'Foundation', '1966', '5£', True)

foundation.present()

print(foundation.price)

print(foundation.bored)

boring\_book = Fiction('boring\_guy', 'boring\_title', 'boring\_date', '9000£', False)

print(boring\_book.bored)

**Exercise 1: Bank Account**

**Instructions**

**Part I:**

1. Create a class called BankAccount that contains the following attributes and methods:
   * balance - (an attribute)
   * \_\_init\_\_ : initialize the attribute
   * deposit : - (a method) accepts a positive int and adds to the balance, raise an **Exception** if the int is not positive.
   * withdraw : - (a method) accepts a positive int and deducts from the balance, raise an **Exception** if not positive

**Part II** : Minimum balance account

1. Create a MinimumBalanceAccount that inherits from BankAccount.
2. Extend the \_\_init\_\_ method and accept a parameter called minimum\_balance with a default value of 0.
3. Override the withdraw method so it only allows the user to withdraw money if the balance remains higher than the minimum\_balance, raise an **Exception** if not.

**Part III**: Expand the bank account class

1. Add the following attributes to the BankAccount class:
   * username
   * password
   * authenticated (False by default)
2. Create a method called authenticate. This method should accept 2 strings : a username and a password. If the username and password match the attributes *username* and *password* the method should set the authenticated boolean to True.
3. Edit withdraw and deposit to only work if authenticated is set to True, if someone tries an action without being authenticated raise an **Exception**

**Part IV**: **BONUS** Create an ATM class

1. \_\_init\_\_:
   1. Accepts the following parameters: account\_list and try\_limit.
   2. Validates that account\_list contains a list of BankAccount or MinimumBalanceAccount instances.  
      **Hint**: isinstance()
   3. Validates that try\_limit is a positive number, if you get an invalid input raise an **Exception**, then move along and set try\_limit to 2.  
      **Hint**: Check out [this tutorial](https://www.programiz.com/python-programming/exception-handling)
   4. Sets attribute current\_tries = 0
   5. Call the method show\_main\_menu (see below)
2. Methods:
   1. show\_main\_menu:
      1. This method will start a while loop to display a menu letting a user select:
         * Log in : Will ask for the users username and password and call the log\_in method with the username and password (see below).
         * Exit.
   2. log\_in:
      1. Accepts a username and a password.
      2. Checks the username and the password against all accounts in account\_list.
         * If there is a match (ie. use the authenticate method), call the method show\_account\_menu.
         * If there is **no match** with any existing accounts, increment the current tries by 1. Continue asking the user for a username and a password, until the limit is reached (ie. try\_limit attribute). Once reached display a message saying they reached max tries and shutdown the program.
   3. show\_account\_menu:
      1. Accepts an instance of BankAccount or MinimumBalanceAccount.
      2. The method will start a loop giving the user the option to deposit, withdraw or exit.