Encapsulation

Encapsulation is the process of using private variables within classes to prevent unintentional or potentially malicious modification of data. By containing and protecting variables within a class, it allows the class and the objects that it creates to function as independent, self-contained, parts functioning within the machine of the program itself.

Through encapsulation variables and certain methods can only be interacted with through the interfaces designated by the class itself.

Python has different levels of restriction that control how data can be accessed and from where. Variables and methods can be public or private. Those designations are made by the number of underscores before the variable or method.

Public
Public variables and methods can be freely modified and run from anywhere, either inside or outside of the class.

Private
The private designation only allows a variable or method to be accessed from within its own class or object. You cannot modify the value of a private variable from outside of a class. Private variables and methods are preceded by two underscores

```
class Car(object):
        def __init__(self, make = 'Ford', model = 'Explorer', year = '2019',
                mileage = '2812', color = 'blue'):
        self.__make = make
        self.__model = model
        self.__year = year
        self.__mileage = mileage
        self.__color = color

    def getModel(self):
         return(self.__model)

mycar = Car()
print(mycar.getModel())
#print(mycar.__model)
```

Polymorphism

Polymorphism allows subclasses to have methods with the same names as methods in their super-classes. It gives the ability for a program to call the correct method depending on the type of object that is used to call it.

```
#Python code to illustrate duck typing

class User(object):
    def __init__(self, firstname):
        self.firstname = firstname

    @property
    def name(self):
        return self.firstname

class Animal(object):
    pass

class Fox(Animal):
    name = "Fox"

class Bear(Animal):
    name = "Bear"

# Use the .name attribute (or property) regardless of the type
for a in [User("Polymorphism"), Fox(), Bear()]:
    print(a.name)
```

```
Polymorphism
Fox
Bear
```

Abstract Classes

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class.

In fact, Python on its own does not provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called for obvious reasons - ABC.

The following Python code uses the abc module and defines an abstract base class:

```
class AbstractAnimal(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def walk(self):
        ''' data '''

    @abc.abstractmethod
    def talk(self):
        ''' data '''

class Duck(AbstractAnimal):
    name = ''

    def __init__(self, name):
        self.name = name
        print(name, 'created.')

    def walk(self):
        print('walks')

    def talk(self):
        print('quack')

obj = Duck('duck1')
obj.talk()
obj.walk()
```

```
duck1 created.
quack
walks
```

Using isinstance()

The following program is stored in file animal.py

```
# The Mammal class represents a generic mammal.
class Mammal:

    # The __init__ method accepts an argument for
    # the mammal's species.

    def __init__(self, species):
        self.__species = species

    # The show_species method displays a message
    # indicating the mammal's species.

    def show_species(self):
        print('I am a', self.__species)

    # The make_sound method is the mammal's
    # way of making a generic sound.

    def make_sound(self):
        print('Grrrrr')

# The Dog class is a subclass of the Mammal class.

class Dog(Mammal):

    # The __init__ method calls the superclass's
    # __init__ method passing 'Dog' as the species.

    def __init__(self):
        Mammal.__init__(self, 'Dog')

    # The make_sound method overrides the superclass's
    # make_sound method.

    def make_sound(self):
        print('Woof! Woof!')

# The Cat class is a subclass of the Mammal class.

class Cat(Mammal):

    # The __init__ method calls the superclass's
```

```
    # __init__ method passing 'Cat' as the species.

    def __init__(self):
        Mammal.__init__(self, 'Cat')

    # The make_sound method overrides the superclass's
    # make_sound method.

    def make_sound(self):
        print('Meow')


# This program demonstrates the use of isinstance() Pythin Library
import animals

def main():
    # Create an Mammal object, a Dog object, and
    # a Cat object.
    mammal = animals.Mammal('regular animal')
    dog = animals.Dog()
    cat = animals.Cat()

    # Display information about each one.
    print('Here are some animals and')
    print('the sounds they make.')
    print('-------------------------')
    show_mammal_info(mammal)
    print()
    show_mammal_info(dog)
    print()
    show_mammal_info(cat)
    print()
    show_mammal_info('I am a string')

# The show_mammal_info function accepts an object
# as an argument, and calls its show_species
# and make_sound methods.

def show_mammal_info(creature):
    if isinstance(creature, animals.Mammal):
        creature.show_species()
        creature.make_sound()
    else:
        print('That is not a Mammal!')
```

```
# Call the main function.
main()
```