Object Oriented Programming

Objectives:

- Define and understand classes and objects.
- Understand encapsulation and how classes support information hiding and implementation independence.
- Understand inheritance and how it promotes software reuse.
- Understand polymorphism and how it enables code generalisation.
- Exclude: method overloading and multiple inheritance

0. Recap

We have used many Python's built-in classes, e.g. `int` , `str` , `float` , `list` , `tuple` etc.

Question:

How to find out the class of an object, e.g. `a = 'abc'` ?

- The built-in method `type()` returns type (class) of an object.
- It actually make use of the `__class__` attribute of the object.

In [8]:
```
a = 'abc'
print(type(a))
print(a.__class__)
```

```
<class 'str'>
<class 'str'>
```

Question:

What are the attributes in the object `a = 'abc'` ?

- Make use of help function provided by Jupyter Notebook.
- Or use built-in `dir()` method.

In [29]:
```
a.*?
```

Question:

How to check whether an object belongs to a particular class?

- The `isinstance()` method can be used to test whether an object belongs to a class.
- All classes have an attribute `__name__` which returns string representation of class name.

In [14]:
```
print(isinstance(a, str))
print(type(a) is str)
print(a.__class__ is str)
print(type(a).__name__)
```

```
True
True
True
str
```

In Python, everything is an object.

- This includes classes (types).
- The `id()` method can be used to get unique ID of an object.

Question:

What is the ID of the `str` class, and ID of a str object `a = 'abc'` ?

In [61]:
```
print(id(str))
a = 'abc'
print(id(a))
print(id(type(a)))
```

```
140728172114000
1850098056192
140728172114000
```

1. Class Basics

Classes are blueprints/template for objects. They define the **structure** and **behavior** of objects.

- Python is highly object-oriented.
- But it does not force you to use it until you need to do so.

Creating a new object is called `instantiation` . An **object** of a class is also called an **instance** of that class.

- Multiple objects can be created from same class.

Class Definition

Classes are defined using the `class` keyword followed by CamelCase name.

- Class instances are created by calling the class as if it is a function.

In [24]:
```
class Vehicle:
    pass

v = Vehicle()
isinstance(v, Vehicle)
```

Out[24]:
```
True
```

When you print an instance, Python shows its class and its memory location.

In [25]:
```
print(Vehicle)
print(v)
```

```
<class '__main__.Vehicle'>
<__main__.Vehicle object at 0x000001AEC7D15B38>
```

Instance Attributes

You can assign values to an object using dot notation. These values are called `attributes` of the instance.

- In Python, you do NOT need to declare a variable before using it. Similarly, you do NOT need to declare an attribute for an object before you use it either.
- Assigning value to an non-existence attribute will create that attribute.
- But using a non-existing attribute directly will cause an `AttributeError` Exception.

Exercise:

- Create an object `v` of class `Vehicle`
- Check whether `v` has an attribute `color` using either `hasattr()` or `dir()` method
- Assign value `blue` to its attribute `color`
- Check existence of `color` attribute again

In [33]:
```
v = Vehicle()
print(hasattr(v, 'color'))
v.color = 'blue'
print(hasattr(v, 'color'))
```

```
False
True
```

Question:

What happens if you try to print out an non-existence attribute `horsepower` ?

- It causes `AttributeError` because `wheels` attribute does not exist

In [34]:
```
print(v.horsepower)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-34-df1c99ddb0c6> in <module>
----> 1 print(v.horsepower)

AttributeError: 'Vehicle' object has no attribute 'horsepower'
```

Initializer Method __init__()

Python class has an initializer method, `__init__()`, which will be automatically called to initialize the newly created object.

- `__init__()` is a **dunder** method which generally are used by Python compiler.
- Its definition is similar to function definition except that its first argument is `self`.
- It can take in additional arguments.

Instance Attributes

Its common to initialize **Instance Attributes** in the initializer method `__init__()`.

Keyword `self`

To access any instance method or instance attribute in the class, you need to prefix it with `self.`.

In [36]:
```
class Vehicle:
    def __init__(self, horsepower, color='black', wheels=4):
        self.horsepower = horsepower
        self.color = color
        self.wheels = wheels

v = Vehicle(100)
v.wheels = 6
print(v.horsepower, v.color, v.wheels)
```

```
100 black 6
```

Instance Methods

Methods are functions defined within a class. **Instance Methods** are functions can be called on objects.

- It defines the **behavior** of objects of the class.
- Methods are called using `instance.method()`.

Argument `self`

- The `self` attribute must be the first input parameter for all instance methods.

- The `self` attribute is refer to current object of the class, i.e. the instance calling the method.
  - This is similar to the `this` in C# or Java.
- When a instance method is called, `self` argument is omitted.

In [37]:
```
class Vehicle:
    def __init__(self, horsepower, color='black'):
        self.horsepower = horsepower
        self.color = color

    def engine_power_hp(self):
        return self.horsepower

    def engine_power_kw(self):
        return self.horsepower * 0.745699872

v = Vehicle(100, 'Blue')
'{} hp = {} kw'.format(v.engine_power_hp(), v.engine_power_kw())
```

Out[37]:
```
'100 hp = 74.5699872 kw'
```

Naming Conventions

A class may contain following attributes:

- instance variables
- class variables
- constructor
- instance methods
- static methods
- class methods

Python has a recommend convention for naming of classes and their attributes. It is good to follow these convention for readability of your code.

https://visualgit.readthedocs.io/en/latest/pages/naming_convention.html (https://visualgit.readthedocs.io/en/latest/pages/naming_convention.html)

Docstring

Similiar to modules and functions. You can add docstring to class and its methods.

- Docstring is enclosed by triple-quotes
- It must be the 1st statement in the class
- Docstrings can be accesses by `__doc__` attribute
- It is used by the `help()` function

In [ ]:
```
class Vehicle:
    '''Class Vehicle
        attributes: horsepower, color
        methods: engine_power_hp(), engine_power_kw()
    '''

    def __init__(self, horsepower, color='Black'):
        self.horsepower = horsepower
        self.color = color

    def engine_power_hp(self):
        '''Return engine power in Horsepower'''
        return self.horsepower

    def engine_power_kw(self):
        '''Return engine power in kW'''
        return self.horsepower * 0.745699872


print(Vehicle.__doc__)
print(Vehicle.engine_power_kw.__doc__)
help(Vehicle)
```

2. Convert to Object to String

Python provides 2 methods `str()` and `repr()` to convert an object to string.

In [39]:
```
s = list(range(5))
print(str(s))
print(repr(s))
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

Difference between str() and repr()

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter.

In [43]:
```
import datetime
now = datetime.datetime.now()
print(str(now))
print(repr(now))
```

```
2020-03-11 00:44:45.310487
datetime.datetime(2020, 3, 11, 0, 44, 45, 310487)
```

The `repr()` method returns a printable representational string of the given object, which would yield an object with the same value when passed to eval().

In [45]:
```
s = repr(now)
print(s)

now2 = eval(s)
print(now == now2)
```

```
datetime.datetime(2020, 3, 11, 0, 44, 45, 310487)
True
```

The `str()` method returns the "informal" or nicely printable representation of a given object, which is suitable to present information to end-user.

In [47]:
```
s = str(now)
print(s)

# now3 = eval(s)    # error
```

```
2020-03-11 00:44:45.310487
```

By default, the `print()` method uses `str()` to convert object to string.

In the `format()` method, you can use converstion flag `!s` and `!r` to call `str()` and `repr()` methods respectively.

In [ ]:
```
now = datetime.datetime.now()
print("Now:\n {0} \n {0!s} \n {0!r}".format(now))
```

Implement `__str__()` and `__repr__()` for Custom Object

By default, our `Vehicle` class inherits `__str__()` and `__repr__()` methods from `Object` class, which print class name and memory location of the object.

In [51]:
```
class Vehicle:
    def __init__(self, plate):
        self.plate = plate

v1 = Vehicle('A1234')

print(str(v1))
print(repr(v1))
```

```
<__main__.Vehicle object at 0x000001AEC7D29240>
<__main__.Vehicle object at 0x000001AEC7D29240>
```

Internally, `repr()` method calls `__repr__()` method of the given object, and `str()` method calls `__str__()` method of given object.

Exercise:

For our `Vehicle` class to support `str()` and `repr()` methods, we can implement `__repr__()` and `__str__()` methods in the class. For example,

- the `str()` will print out `"Vehicle: A1234"`
- the `repr()` will print out `"Vehicle('A1234')"`

In [54]:
```
class Vehicle:
    def __init__(self, plate):
        self.plate = plate

    def __str__(self):
        return 'Vehicle: {}'.format(self.plate)

    def __repr__(self):
        return "Vehicle('{}')".format(self.plate)


v = Vehicle('A1234')

print(str(v))
print(repr(v))

# Create new object from string
v2 = eval(repr(v))
```

```
Vehicle: A1234
Vehicle('A1234')
```

3. Class Attributes

Class Attributes are attributes which belong to class instead of a particular object.

- It can be accessed through either class or instance.

Example 1

Create a `Time` class contains 3 instance attributes, `hour` , `minute` and `second` .

- Implement its `__str__()` method which return time in "hh:mm:ss" format.

Sample Output

```
10:20:30
```

In [ ]:
```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return "{}:{}:{}".format(self.hour, self.minute, self.second)

## Test
t1 = Time(10, 20, 30)
print(t1)
```

Exampe 1 (cotinued)

To support validation of initial values, we defined 2 class attributes MAX_HOUR and MAX_MIN_SEC, which has value 24 and 60 respectively.

- These class attributes are used during input validation.

In [62]:
```
class Time:

    MAX_HOUR = 24
    MAX_MIN_SEC = 60

    def __init__(self, hour=0, minute=0, second=0):
        if hour < 0 or hour >= Time.MAX_HOUR:
            raise ValueError('Hour must be between 0 and 23')
        if minute < 0 or minute >= Time.MAX_MIN_SEC or second < 0 or second >= Ti
            raise ValueError('Minute/Second must be bewteen 0 and 59')
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return "{}:{}:{}".format(self.hour, self.minute, self.second)


t1 = Time(10, 20, 30)
print(t1.MAX_HOUR)
print(Time.MAX_HOUR)
## Code raise execption when input is invalid
# t2 = Time(25, 61, 61)
```

```
24
24
```

Example 2

We can use class attributes to keep a rolling value which is shared among all instances. For example, we would like to keep track of number of Customers and assign each customer a unique serial number.

Sample Output

```
1
2
3
3
```

In [58]:
```
class Customer:

    next_serial = 1

    def __init__(self):
        self.serial = type(self).next_serial
        type(self).next_serial += 1

## Test
s1 = Customer()
s2 = Customer()
print(s1.serial)
print(s2.serial)
print(Customer.next_serial, s1.next_serial, s2.next_serial)
```

```
1
2
3 3 3
```

Instance Attribute vs Class Attribute

Instance attributes belong to a particular instance.

- Modifying instance attribute of an instance does not affect other instances.

Class attributes are shared among all instances.

- They can be accessed not only through class but also through an instance.

Modify a Class Attribute

Modification to class attribute can only be done with the notation `ClassName.AttributeName`. Otherwise, a new instance variable will be created.

In [70]:
```
class A:
    cls_attr = "class attribute"

x = A()
print(x.cls_attr)

A.cls_attr = "my value"
print(x.cls_attr)
print(x.cls_attr is A.cls_attr)
```

```
class attribute
my value
True
```

Question:

In above example, what if you modify `A.cls_attr = "my value"` to `x.cls_attr = "my value"` ?

Mutable vs Immutable (Confusing)

Modification to attribute of immutable type will create a new object.

In [71]:
```
class A:
    cls_attr = [1,2,3]

x = A()
x.cls_attr.append(4)
print(A.cls_attr)

x.cls_attr = [1,2,3,5]
print(A.cls_attr)
print(x.cls_attr)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 5]
```

4. Static Methods and Class Methods

Static Methods

In Python, all instance methods have `self` as their first argument.

Static methods in Python are similar to instance methods, the difference being that a static method is bound to a class rather than the objects for that class.

- A static method is a method which does not has `self` as its first argument.
- It can be called without an object of that class.

- This also means that static methods cannot modify the state of an object as they are not bound to it.

Static method are declared using `@staticmethod` decorator.

- The `@staticmethod` decorator is optional. But static method without `@staticmethod` decorator cannot be called from its instance.

For example, we add a static method in `Time` class to check if a hour is AM or PM.

In [72]:
```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    @staticmethod
    def get_am_pm(hour):
        if hour < 12:
            return 'AM'
        return 'PM'

## Test
print(Time.get_am_pm(13))
## You cannot do following if the static method is declared without @staticmethod
t1 = Time()
print(t1.get_am_pm(13))
```

```
PM
PM
```

Class Methods

Class methods are much like **static method**. They are methods that are bound to a class rather than its object.

The difference between a static method and a class method is:

- Static method knows nothing about the class and just deals with the parameters.
- Class method works with the class since its parameter is always the class itself.

To create a class method, use `@classmethod` decorator.

Example:

To buy a car in Singapore, buyer needs to purchase a Certificate of Entitlement or COE. COEs were divided into several categories.

- Cat A: Cars with engine capacity at 1600cc & below, and the engine power should not exceed 97 kilowatts (kW)
- Cat B: Cars with engine capacity above 1600cc, or the engine power output exceeds 97 kW

Implement a `Car` class with following specifications.

- It has 2 instance attributes `engine_capacity` and `category`
- Its initializer accepts 1 input to initialize `engine_capacity`
- It has a class method `get_category()` which returns category A or B based on parameter engine_capacity value
- It also defines a class attribute `MAX_CAT_A_CC` with value 1600.

In [ ]:
```
class Car:

    def inst_category(self):
        if self.engine_capacity <= type(self).MAX_CAT_A_CC:
            return 'A'
        return 'B'

    @staticmethod
    def static_category(engine_capacity, MAX):
        if engine_capacity <= MAX:
            return 'A'
        return 'B'

    MAX_CAT_A_CC = 1600

    @classmethod
    def get_category(cls, engine_capacity):
        if engine_capacity <= cls.MAX_CAT_A_CC:
            return 'A'
        return 'B'

    def __init__(self, engine_capacity):
        self.engine_capacity = engine_capacity
        # Use class method
#         self.category = type(self).get_category(engine_capacity)
        # Use instance method
#         self.category = self.inst_category()
        # Use static method
        self.category = type(self).static_category(engine_capacity, type(self).MA

c1 = Car(1000)
print(c1.category)
c1 = Car(1700)
print(c1.category)
```

5. Properties

It is common to have **getter** and **setter** methods in the class, which manage access and modification to attributes in the class.

Example:

- In `Person` class, `_name` attribute is marked as "private".
- For ourside world to access and modify `_name` , the `getName()` and `setName()` methods are provided.

In [ ]:
```
class Person:
    def __init__(self, name):
        self._name = name

    def getName(self):
        return self._name

    def setName(self, value):
        self._name = value

person1 = Person('John')
print(person1.getName())
person1.setName('Smith')
print(person1.getName())
```

Property Decorator

Python provides `@property` decorator for getter and setter methods. Property makes getter and setters look like a normal attribute.

- The `@property` decorator marks the getter method
- The `@attr.setter` decorator marks the setter method for attribute attr

In [73]:
```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def my_name(self):
        s = "[{}]".format(self._name)
        return s

    @my_name.setter
    def my_name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value


p1 = Person('John')
print(p1.my_name)
print(p1._name)
p1.my_name = 'Smith'
print(p1.my_name)
p1.my_name = ''
```

```
[John]
John
[Smith]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-73-f55f3e3ce39a> in <module>
     20 p1.my_name = 'Smith'
     21 print(p1.my_name)
---> 22 p1.my_name = ''

<ipython-input-73-f55f3e3ce39a> in my_name(self, value)
     11     def my_name(self, value):
     12         if not value:
---> 13             raise ValueError("Name cannot be empty")
     14         self._name = value
     15

ValueError: Name cannot be empty
```

Read-only Computed Attribute

It is common to use @property to implement a read-only computed attribute.

Exercise:

Construct a class `Circle` which has a "private" attribute `radius` .

- Implement its initializer to initialize `radius` from input
- Implement property methods of `radius`
- Implement 2 read-only properties which returns area and perimeter

In [5]:
```
import math

class Circle:

    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        self._radius = value

    @property
    def area(self):
        return math.pi*(self._radius**2)

    @property
    def perimeter(self):
        return 2 * math.pi * self._radius

## Test
c = Circle(5)
print(c.area, c.perimeter)
```

```
78.53981633974483 31.41592653589793
```

6. Inheritance

Similar to other programming languages, Python allows class inheritance.

In following code sample, both class `B` and `C` inherit from class `A`.

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass
```

The special attribute `__base__` returns its 1st base class. To get all base classes, use attribute `__bases__`.

We can test whether a class is subclass of one or more classes using `issubclass()` method.

In [2]:
```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

print(A.__base__)
print(B.__base__)
print(issubclass(B,(A)))
print(issubclass(C,(A)))
```

```
<class 'object'>
<class '__main__.A'>
True
True
```

Code Reuse

A sub-class can be derived from a base-class and inherit its methods and variables. This allows sharing of implementation between classes, which avoids code duplication

Exercise

Study the code in `Teacher` and `Student` classes and perform following:

- Abstract common code in `Teacher` and `Student` class to a `Person` class
- Modify following two classes to inherit from `Person` class

```
class Teacher():
    def __init__(self, firstName, lastName):
        self._firstName = firstName
        self._lastName = lastName

    @property
    def fullname(self):
        return self._firstName + ' ' + self._lastName

    def __str__(self):
        return '{}: {}'.format(type(self).__name__, self.fullname())

    def work(self):
        print("{} is working".format(self.fullname))

class Student():
    def __init__(self, firstName, lastName):
        self._firstName = firstName
        self._lastName = lastName

    @property
    def fullname(self):
        return self._firstName + ' ' + self._lastName

    def __str__(self):
        return '{}: {}'.format(type(self).__name__, self.fullname)

    def study(self):
        print("{} is studying".format(self.fullname))
```

In [11]:
```
class Person:
    def __init__(self, firstName, lastName):
        self._firstName = firstName
        self._lastName = lastName

    @property
    def fullname(self):
        return self._firstName + ' ' + self._lastName

    def __str__(self):
        return '{}: {}'.format(type(self).__name__, self.fullname)


class Teacher(Person):

    def work(self):
        print("{} is working".format(self.fullname))


class Student(Person):

    def study(self):
        print("{} is studying".format(self.fullname))

## Test
Teacher('Alan', 'Tan').work()
Student('Colin', 'Seng').study()
```

```
Alan Tan is working
Colin Seng is studying
```

Method Overriding

A subclass may override a method defined in its superclass.

Example:

- Class B doesnot override `hi()` method in class A
- Class C overrides `hi()` method in class B

In [28]:
```
class A:
    def hi(self):
        print('hi A')

class B(A):
    pass

class C(A):
    def hi(self):
        print('hi C')

b = B()
b.hi()
c = C()
c.hi()
```

```
hi A
hi C
```

Super Function - super()

With inheritance, the super() function allows us to call a method from the parent class.

In [76]:
```
class A:

    def __init__(self):
        self.x = 1000

    def hi(self):
        print('hi A')

    def hello(self):
        print('hello A')

class C(A):
    def __init__(self):
        super().__init__()
        self.y = 9999

    def hi(self):
        self.hello()
        super().hi()
        print('hi C')
        print(self.x)

c = C()
c.hi()
```

```
hello A
hi A
hi C
1000
```

7. Advanced Inheritance (Optional)

Abstract Method

Abstract method is a method without implementation. It defines signature of a method which can be implemented in subclasses.

There are 2 common ways of implementing abstract methods.

Method 1

- Raise `NotImplementedError` in the method body.
- Subclass still can be instantiated.

In [9]:
```
class A:
    def foo(self):
        raise NotImplementedError('Must override foo()!')

class B(A):
    def foo(self):
        print('Good!')

class C(A):
    pass


b = B()
b.foo()
c = C()
c.foo()
```

```
Good!
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-9-a0e7c415806b> in <module>
     14 b.foo()
     15 c = C()
---> 16 c.foo()

<ipython-input-9-a0e7c415806b> in foo(self)
      1 class A:
      2     def foo(self):
----> 3         raise NotImplementedError('Must override foo()!')
      4
      5 class B(A):

NotImplementedError: Must override foo()!
```

Method 2

- Use `@abstractmethod` decorator from abc (Abstract Base Class) module.

- Subclass cannot be instantiated if it does not implement the abstract class.

In [10]:
```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

class B(A):
    pass

b = B()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-69c12e00cb5c> in <module>
      9     pass
     10
---> 11 b = B()

TypeError: Can't instantiate abstract class B with abstract methods foo
```

Multiple Inheritance

Python supports multiple inheritance, i.e. a class may inherit from multiple base classes.

In following code sample, Class `D` inherits from both class `B` and `C`.

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

In [1]:
```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.__bases__)
print(issubclass(D,(B)))
print(issubclass(D,(A)))
print(issubclass(D,(A,int,str)))
```

```
(<class '__main__.B'>, <class '__main__.C'>)
True
True
True
```

Method Resolution Order

When an attribute is invoked in a class, Python will try to search for this attribute in current class and followed by its parent classes. The order of resolution is called **Method Resolution Order** (MRO).

Each class has a MRO list, which can be accessed using special attribute `__mro__` .

In [74]:
```
class A:
    x = 'A'

class B(A):
    pass
#     x = 'B'

class C(A):
    pass
#     x = 'C'

class D(B, C):
    pass

print(D.x)
print(D.__mro__)
```

```
B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__ma
in__.A'>, <class 'object'>)
```

8. "Private" Attribute and Name Mangling (Optional)

Everything is Public

All methods and attributes in Python class are pbulic, i.e. they can be accessed by users.

- Python has NO access modifier, i.e. `public & protected & private`, like in C# or Java.
- It's a convention to prefix a instance variable with `_` to indicate a method or an attribute is only for internal use.
- Such methods and attributes should not be used directly by users of the class. But you can still access them directly, which is useful for debugging purpose.
- ***In Python, we are consenting adults.***

double_leading_and_trailing_underscore

Python use this convention for special variables or methods (so-called “magic method”) such as `__eq__()`, `__len__()`.

- These methods provides special syntactic features or does special things.
- User might modify such special method in rare case. E.g. You customize `__init__()` to initialize an object.
- User should not define his own method in such convention.

single-leading-underscore

Pyton uses single leading underscores is a **convention** to indicate an attribute is for internal use. For example, user of a class `A` should not be using `A._attr` directly because `_attr` is meant for internal use in class.

- This is just a convention which has no effect to Python interpretor

Note: When a moudle is imported, method and variable with single-leading-underscore will NOT be imported.

double-leading-underscore

When a class attribute is named with double-leading-underscore, it invokes **name mangling**.

Name Mangling

Python interpretor will prefix name with double-leading-underscore with `_classname`, e.g. `__foo` in class `MyClass` will become `_MyClass__foo`.

In [ ]:
```
class Test(object):
    def __init__(self):
        self._a = 'a'
        self.__b = 'b'

t = Test()
print(t._a)
print(t._Test__b)
## AttributeError
#print(t.__b)
```

Name Mangling - Avoid Attribute Overriden

In parent class, some member attributes are tied to the specific implementation in methods of parent class. Such attributes may be accidentally overwrite in subclass.

Name Mangling ensures that such attributes will not be overriden by a similar name in its sub-class.

Example

- Case 1: The `_test()` method in class `A` is overriden by `_test()` in class `B`
- Case 2: The `__test()` method in class `A` will not be overriden by `__test()` in class `B`

In [1]:
```
## Case 1

class A(object):
    def _test(self):
        print("Class A test method")

    def test(self):
        self._test()

class B(A):
    def _test(self):
        print("Class B test method")


a = A()
a.test()
b = B()
b.test()
```

```
Class A test method
Class B test method
```

In [4]:
```python
## Case 2

class A(object):
    # Name is modified to be _A__test()
    def __test(self):
        print("Class A test method")

    def test(self):
        # Actually calling _A__test()
        self.__test()

class B(A):
    # Name is modified to be _B__test()
    def __test(self):
        print("Class B test method")

a = A()
a.test()
b = B()
b.test()
```

```
Class A test method
Class A test method
```

9. Polymophism (Optional)

Polymophism in object-oriented programming means to process objects differently based on their data type. In another word, one method can have different implementations, either in the same class or between different classes.

- **Method Overriding:** Same method with different implementations in **derived classes**.
- **Method Overloading:** Same method with different implementations in **same class**.

Method Overriding

A subclass can override a method in the base class.

In following example, class `Pet` can have a method `talk()` and its subclasses `Dog` and `Cat` can make different sounds in their `talk()` method.

In [77]:
```
class Pet:
    def talk(self):
        print('????')

class Dog(Pet):
    def talk(self):
        print('woof')

class Cat(Pet):
    def talk(self):
        print('meow')

def animal_sound(animal):
    animal.talk()

animal_sound(Pet())
animal_sound(Dog())
animal_sound(Cat())
```

```
????
woof
meow
```

Method Overloading - NOT AVAILABLE

Python doesnot support method overloading. It keeps only the latest definition of the method.

In [14]:
```
def add(a,b):
    return a+b

def add(a,b,c):
    return a+b+c

add(1,2,3)
## Raise a TypeError
# add(2,3)
```

Out[14]:
```
6
```

Implement Multiple Initializers using Class Method (Optional)

Python doesn't support method overloading. Class methods are used as Factory methods, which is good for implementing alternative initializers.

Example

- Class `Color` has 3 instance attribues `red`, `green` and `blue`. Its `__init__()` initializer takes in 3 arguments to initialize the 3 instance attributes
- Implement a class method `from_json()` which accepts a JSON string `'{"red":100, "green":150, "blue":200}'` to create a Color object

In [ ]:
```
class Color:

    def __init__( self, red=0, green=0, blue=0 ):
        self.red = red
        self.green = green
        self.blue = blue

    def __str__(self):
        return '({},{},{})'.format(self.red, self.green, self.blue)

    ## define class method here
    @classmethod
    def from_json(cls, rgb_json):
        import json
        rgb = json.loads(rgb_json)
        return cls(rgb['red'], rgb['green'], rgb['blue'])


## Test
c1 = Color(100, 150, 200)
c2 = Color.from_json('{"red":100, "green":150, "blue":200}')
print(c1)
print(c2)
```