Object Oriented Programming - Revisit (part 2)

Scope:

- Inheritance
- Properties
- Name Mangling

1. Inheritance

Inheritance allows one class to inherit all attributes and methods of another class. This is one of the major benefits of object oriented programming.

- **Parent Class** is the class being inherited from, also called **base class**.
- **Child class** is the class that inherits from another class, also called **derived class**.

Benefits:

- Reuse Quality Code: Reuse existing code which is already tested.
- Improve Code Readability: Program structure is short and concise.
- Improve Code Reliability: Avoid code duplication and easier to debug.
- Save Time and Effort

Basic Syntax

- Without specifying parent class, the class inherits from `object` class.
- The `__base__` attribute of a class returns its base class.
- `issubclass()` function checks whether a class is a subclass of another.

In [1]:

```python
class Parent:
    pass

class Child(Parent):
    pass

print(Parent.__base__)
print(Child.__base__)
print(issubclass(Child,Parent))
```

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

Inheritance

Base Class

Define a class `Circle`, which has property `radius`, and a method `get_area()` which calculates area of the circle.

- Initialize its property in its constructor function, i.e. `__init__()` function.
- Implement its `__str__()` function which returns string `Circle: radius=x`.

In [2]:

```
class Circle:

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

    def get_area(self):
        import math
        return math.pi * (self.radius**2)

    def __str__(self):
        return '{}: radius={}'.format(self.__class__.__name__, self.radius)

c = Circle(2)
print(c)
print(c.get_area())
```

```
Circle: radius=2
12.566370614359172
```

Derived Class

Implement another class `Cylinder` which extends from `Circle`.

- Without any coding, `Cylinder` class is able to access to all attributes in `Circle` class.

In [3]:

```
class Cylinder(Circle):
    pass

c2 = Cylinder(2)
print(c2.radius)
print(c2)
print(c2.get_area())
```

```
2
Cylinder: radius=2
12.566370614359172
```

Method Overriding

In above `Cyclinder` example, the `get_area()` method doesn't return the correct value. To calculate are of a Cyclinder, we need its `height` property too.

Override `__init__()`

The cyclinder constructor `__init__()` function needs to take in 2 parameters, `radius` and `height` .

- After implementation, you can no longer use call constructor with `Cyclinder(2)` because it expects 2 positional arguments.

In [4]:

```
class Cylinder(Circle):

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


c2 = Cylinder(2,5)
print(c2)
```

```
Cylinder: radius=2
```

Override `__str__()`

We need to override `__str__()` function so that its returned string include `height` value too.

In [5]:

```
class Cylinder(Circle):

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

    def __str__(self):
        return '{}: radius={}, height={}'.format(
            self.__class__.__name__, self.radius, self.height)


c2 = Cylinder(2,5)
print(c2)
```

```
Cylinder: radius=2, height=5
```

The `super()`

The `Circle.get_area()` method returns area of circle. We still need to override the `get_area()` function in `Cylinder` class to return `2 * circle + 2 * pi * radius * height` .

The `get_area()` function in base class `Circle` is stil useful to get the area of circle. To access it, we can use `super()` object.

The `super()` returns object of parent class. Through it, we can access parent version of overriden attribute(s).

In [6]:

```python
class Cylinder2(Cylinder):

    def get_area(self):
        import math
        area = super().get_area() * 2
        area = area + 2 * math.pi * self.radius * self.height
        return area

c2 = Cylinder2(2,5)
print(c2.get_area())
```

```
87.96459430051421
```

Method Overloading? Not Supported

What is method overloading?

- Multiple methods of same name, same return data type, but different input parameters.

Python does **NOT** support method overloading.

In [7]:

```python
class Adder:

    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def add(x, y, z):
        return x + y + z
```

The 2nd definition of `add()` method overwrites 1st definition. Thus following code will cause a Error.

In [8]:

```python
# Adder.add(1,2)
```

2. Properties (optional)

In object oriented programming, it is common practice to use `setter` and `getter` function to encapsulate a variable in class.

In [9]:

```
class Person:

    def __init__(self, name = ''):
        self._name = name

    def get_name(self):
        return self._name

    def set_name(self, val):
        self.name = val

p = Person()
p.set_name('Bob')
p.get_name()
```

Out[9]:

```
''
```

Property is a simple method to decorate the class's setter and getter.

- It makes getter and setters look like a normal attribute.

In [10]:

```
class Person:

    def __init__(self, name = ''):
        self._name = name

    def get_name(self):
        return self._name

    def set_name(self, val):
        self._name = val

    name = property(get_name, set_name)

p = Person()
p.name = 'Bob'
print(p.name)
```

```
Bob
```

An alternative way is to use `@property` decorator.

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

In [11]:

```
class Person:

    def __init__(self, name = ''):
        self._name = name

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

    @name.setter
    def name(self, val):
        self._name = val


p = Person()
p.name = 'Bob'
print(p.name)
```

```
Bob
```

Read-only Attributes

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

For example, following `Circle` class defines 2 read-only computed properties `area` and `perimeter` .

In [12]:

```
import math

class Circle:

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

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

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

3. Private/Public Attributes (optional)

All methods and variables in a Python class or object are public, i.e. they can be accessed by users.

- Python has NO access modifier, i.e. like `public` & `protected` & `private` in C# or Java.
- It uses a convention to indicate whether an attribute is for system use or class-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 all consenting adults.

a) System Attribute `__attr__`

Attributes with **double-leading and double-trailing underscores** are defined by Python. They are called `magic attributes` or `system attributes` . Such attributes should not be used.

For example, the `__class__` , `__name__` property, the `__init__()` and `__str__()` methods.

b) Class/Module Attribute `_attr`

Attributes with **single-leading underscores** are for internal use in the class or module.

- 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.

c) Name Mangling Attribute `__attr`

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

Name Mangling

Python interpretor will prefix such attributes with `_classname` , e.g. `__foo` in class `Bar` will become `_Bar__foo` .

In [13]:

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

t = Test()
print(t._a)
print(t._Test__b)
```

```
a
b
```

Avoid Accidental Method Overriding

Name mangling is used to avoid accidental overriding of attributes in the subclass.

In following example, class `B` inherits `test()` method from `A` .

In [14]:

```
class A:
    def _test(self):
        print("Running test...")

    def test(self):
        self._test()

class B(A):
    pass

#     def _test(self):
#         print("Unintended test method in B")

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

```
Running test...
```

Unintentionally, class `B` may implement another method `_test()` which may overrides `_test()` method in A. This will break the `test()` method inherited from class `A` .

In [15]:

```
class A:
    def _test(self):
        print("Running test...")

    def test(self):
        self._test()

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

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

```
Unintended test method in B
```

To avoid such accident, we can rename `_test()` to `__test()` .

In [16]:

```
class A:
    def __test(self):
        print("Running test...")

    def test(self):
        self.__test()

class B(A):
    def __test(self):
        print("Unintended test method in B")

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

```
Running test...
```