15. OOP - Part 2

October 21, 2022

1 Encapsulation

- We can restrict access to methods and variables using Encapsulation.
- This prevents data from direct modification

1.1 Public Access Modifier

- The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

1.2 Protected Access Modifier

- Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it.
- Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

[1]:
```
class Person:
    def __init__(self, fname, lname, gender="M"):
        self._firstname = fname
        self._lastname = lname
        self.gender = gender

    def _printname(self):
        print(self._firstname, self._lastname)
```

[2]:
```
class Student(Person):
    '''Class to define student and its methods'''
    def __init__(self, fname, lname, gender, school_name = "ACTS"):
        super().__init__(fname, lname, gender)
        self.school_name = school_name # also new properties can be added

    def school_info(self):
        '''Function to get info about school.'''
        print("School Name:", self.school_name)
        # print(self._firstname)
```

[3]:
```
s6 = Student("Ritik","Rawat","M")
```

[4]:
```
s6.school_info()
```

```
School Name: ACTS
```

[5]:
```
s6._firstname
```

[5]:
```
'Ritik'
```

[6]:
```
s6._printname()
```

```
Ritik Rawat
```

[8]:
```
p8 = Person("Ritik","Rawat","M")
```

[9]:
```
p8._firstname
```

[9]:
```
'Ritik'
```

[16]:
```
class CsStudent(Student):
    def __init__(self, fname, lname, gender):
        Student.__init__(self,fname, lname, gender)
        self.spl = "Algorithms"

    def info(self):
        print(self.spl)
        print(self._firstname)

        return self.spl, self._firstname

#  return multiple example
```

[17]:
```
css = CsStudent("Ritik","Rawat","M")
```

[18]:
```
css.info()
```

```
Algorithms
Ritik
```

[18]:
```
('Algorithms', 'Ritik')
```

[15]:
```
css._printname()
```

```
ABC Rawat
```

[14]:
```
css._firstname = "ABC"
```

- Above, we used `p8._firstname` to modify `_firstname` attribute. However, it is still accessible in Python. Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class.

1.3 Private Access Modifier

- The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

[19]:
```
class Person:
    def __init__(self, fname, lname, gender):
        self._firstname = fname
        self.__lastname = lname
        self.gender = gender

    def printname(self):
        print(self._firstname, self.__lastname)
```

[20]:
```
class Student(Person):
    '''Class to define student and its methods'''
    def __init__(self, fname, lname, gender, school_name = "ACTS"):
        super().__init__(fname, lname, gender)
        self.school_name = school_name # also new properties can be added

    def school_info(self):
        '''Function to get info about school.'''
        print("School Name:", self.school_name)
        print(self.__lastname)
```

[21]:
```
s7 = Student("Ritik","Rawat","M")
```

[22]:
```
s7.school_info()
```

```
School Name: ACTS
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [22], line 1
----> 1 s7.school_info()

Cell In [20], line 10, in Student.school_info(self)
      8 '''Function to get info about school.'''
      9 print("School Name:", self.school_name)
---> 10 print(self.__lastname)

AttributeError: 'Student' object has no attribute '_Student__lastname'
```

[23]:
```
s7.printname()
```

```
Ritik Rawat
```

[24]:
```
p8 = Person("Ritik","Rawat","M")
```

[25]:
```
p8.printname()
```

```
Ritik Rawat
```

[26]:
```
p8.__lastname
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [26], line 1
----> 1 p8.__lastname

AttributeError: 'Person' object has no attribute '__lastname'
```

2 Magic Methods or Dunder Methods

- Magic methods in Python are the special methods that start and end with the double underscores.
- They are also called dunder (double underscore) methods.
- Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.
- Built-in classes in Python define many magic methods.
- Use the `dir()` function to see the number of magic methods inherited by a class.

[28]:
```
dir(Person)
```

[28]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'printname']
```

```
[29]: num = 10
```

```
[30]: num + 20
```

```
[30]: 30
```

```
[31]: num.__add__(20)
```

```
[31]: 30
```

- As you can see, when you do `num+20`, the `+` operator calls the `__add__(20)` method. You can also call `num.__add__(20)` directly which will give the same result.
- However, magic methods are not meant to be called directly, but internally, through some other methods or actions.
- Magic methods are most frequently used to define overloaded behaviours of predefined operators in Python.
- For instance, arithmetic operators by default operate upon numeric operands. This means that numeric objects must be used along with operators like `+`,`-`, `*`, `/`, etc. The + operator is also defined as a concatenation operator in string, list and tuple classes. We can say that the + operator is overloaded.
- In order to make the overloaded behaviour available in your own custom class, the corresponding magic method should be overridden. For example, in order to use the + operator with objects of a user-defined class, it should include the `__add__()` method.

```
[32]: dir(Person)
```

```
[32]: ['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'printname']
```

2.1 `__new__()` method

- Languages such as Java and C# use the `new` operator to create a new instance of a class.
- In Python the `__new__()` magic method is implicitly called before the `__init__()` method.
- The `__new__()` method returns a new object, which is then initialized by `__init__()`.

2.2 `__str__()` method

- Another useful magic method is `__str__()`.
- It is overridden to return a printable string representation of any user defined class.
- We have seen `str()` built-in function which returns a string from the object parameter.
- For example, `str(12)` returns `'12'`. When invoked, it calls the `__str__()` method in the int class.

[33]:
```
class Person:
    def __init__(self, fname, lname, gender):
        self.firstname = fname
        self.lastname = lname
        self.gender = gender

    def printname(self):
        print(self._firstname, self.__lastname)

    def __str__(self):
        return f'firstname: {self.firstname}, lastname: {self.lastname}, gender:
 ↪ {self.gender}'
```

[34]:
```
p23 = Person("Peter","Parker",28)
```

```
[35]: p23.__str__()
```

```
[35]: 'firstname: Peter, lastname: Parker, gender: 28'
```

```
[36]: print(p23)
```

```
firstname: Peter, lastname: Parker, gender: 28
```

3 Operator Overloading

- Python operators work differently for built-in classes.
- For example, the + operator will,
  - perform arithmetic addition on two numbers
  - merge two lists
  - concatenate two strings.
- This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

3.1 Overloading + operator:

- To overload the + sign, we will need to implement `__add__()` function in the class.
- For example + to add two points in a Cartesian system

```
[37]: class Point:
          def __init__(self,x = 0,y = 0):
              self.x = x
              self.y = y

          def __add__(self, other):
              x = self.x + other.x
              y = self.y + other.y
              return Point(x,y)
```

```
[38]: p1 = Point(2,3)
      p2 = Point(1,2)
```

```
[39]: p3 = p1 + p2
```

```
[42]: p3.x, p3.y
```

```
[42]: (3, 5)
```

```
[43]: id(p3)
```

```
[43]: 139817168132128
```

```
[44]: p3.x, p3.y
```

[44]: (3, 5)

3.2 Magic methods for operators

Operator	Magic Method
+	`__add__(self, other)`
-	`__sub__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`
//	`__floordiv__(self, other)`
%	`__mod__(self, other)`
**	`__pow__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`
==	`__eq__(self, other)`
!=	`__ne__(self, other)`

[45]:
```
# Example: __ge__()
class Distance:
    def __init__(self, x = None,y = None):
        self.ft = x
        self.inch = y

    def __ge__(self, x):
        val1 = self.ft*12 + self.inch
        val2 = x.ft*12 + x.inch

        if val1 >= val2:
            return True
        else:
            return False
```

[46]:
```
d1 = Distance(2,1)
d2 = Distance(4,10)
```

[47]:
```
d1 >= d2
```

[47]:
```
False
```

[48]:
```
d2 >= d1
```

[48]:
```
True
```

4 Abstraction

- Abstraction is one of the most important features of object-oriented programming. It is used to hide the background details or any unnecessary implementation.
- For example, when you use a washing machine for laundry purposes. What you do is you put your laundry and detergent inside the machine and wait for the machine to perform its task. How does it perform it? What mechanism does it use? A user is not required to know the engineering behind its work. This process is typically known as data abstraction, when all the unnecessary information is kept hidden from the users.

- In Python, we can achieve abstraction by incorporating abstract classes and methods.
- Any class that contains abstract method(s) is called an abstract class.
- Abstract methods do not include any implementations – they are always defined and implemented as part of the methods of the sub-classes inherited from the abstract class.

- By default, Python does not provide abstract classes.
- Python comes with a module named ABC. This provides the base for defining Abstract Base classes.
- `ABC` works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the keyword `@abstractmethod`.

- In programming, decorator is a design pattern that adds additional responsibilities to an object dynamically.

- In Python, a function is the first-order object. So, a decorator in Python adds additional responsibilities/functionalities to a function dynamically without modifying a function.

- In Python, a function can be passed as an argument to another function. It is also possible to define a function inside another function, and a function can return another function.

- So, a decorator in Python is a function that receives another function as an argument. The behavior of the argument function is extended by the decorator without actually modifying it. The decorator function can be applied over a function using the `@decorator` syntax.

[49]:
```python
from abc import ABC, abstractmethod
```

[50]:
```python
class Shape(ABC):

    @abstractmethod
    def print_area(self):
        pass
```

[51]:
```python
shpe = Shape()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [51], line 1
----> 1 shpe = Shape()
```

```
TypeError: Can't instantiate abstract class Shape with abstract method print_area
```

[52]:
```
class Rectangle(Shape):

    def __init__(self):
        self.length = 10
        self.breadth = 5
```

[53]:
```
rshpe_1 = Rectangle()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [53], line 1
----> 1 rshpe_1 = Rectangle()

TypeError: Can't instantiate abstract class Rectangle with abstract method␣
 ↪print_area
```

[54]:
```
class Rectangle(Shape):

    def __init__(self):
        self.length = 10
        self.breadth = 5

    def print_area(self):
        return self.length * self.breadth
```

[55]:
```
rshpe_2 = Rectangle()
```

[56]:
```
rshpe_2.print_area()
```

[56]:
```
50
```