Section A3 Polymorphism

Operator Overloading

The use of operator overloading is another way of implementing polymorphism. Therefore, *operator overloading* is the process of using the same operator in multiple forms depending on the operands used. For example, when the plus operator (+) is called, it invokes the special __add__ method. However, this operator acts differently for different data types. If two integers are used, the + operator *adds* the numbers. If two strings are used, the + operator *merges* strings.

To summarise, the moment you use the following operators,

- + operator calls an obj.__add__() method.
- - operator calls an obj.__sub__() method.
- * operator calls an obj.__mul__() method.

In [5]:
```
class Quantity:
    def __init__(self, y1, y2):
        self.y1 = y1
        self.y2 = y2

    def __str__(self):
        return "{},{}".format(self.y1,self.y2)

    def __add__(self, other):
        y1 = self.y1 + other.y1
        y2 = self.y2 + other.y2
        return (y1,y2)

q1 = Quantity(1, 2)
q2 = Quantity(3, 3)

#print the attributes in the instances individually
print(q1)
print(q2)

#print the attributes in the instances combinedly
print(q1+q2)
```

Out[5]:
```
1,2
3,3
(4, 5)
```

Operator	Magic Method

+	__add__(self, other)
-	__sub__(self, other)
/	__truediv__(self, other)
*	__mul__(self, other)
<	__lt__(self, other)
>	__gt__(self, other)
==	__eq__(self, other)

Method Overloading
This means that if you have a class with two methods of the same name and different parameters, then the method is said to be overloaded.
the method add() is overloaded.

```
class Total:
    def add(self, x, y):
        s = x + y
        print(s)

    def add(self, x, y, z):
        s = x + y + z
        print(s)

obj = Total()
obj.add(10, 20)
```

```
TypeError                                 Traceback (most recent call last)
547.py in
      9
     10 obj = Total()
---&gt; 11 obj.add(10, 20)

TypeError: add() missing 1 required positional argument: 'z'
```

Unlike other OOP languages, Python does not support method overloading.

Even though Python does not support method overloading, there is a pythonic way of implementing method overloading using *default arguments*.

```
class Total:

    def add(self, x = None, y = None, z = None):
        if x != None and y != None and z != None:
            print (x + y + z)
        else:
            print(x + y)

obj = Total()
obj.add(10, 20)
obj.add('python', 'programming')
obj.add(30, 40, 50)
```

```
30
pythonprogramming
120
```

Method Overriding

A method is said to be *overridden* when a method in child class has the same name and same signature (i.e., same name, the same number of parameters, and the same return type) as that of a method in the parent class.

OOP Best Practices

Good Names

You should use the upper-case camel style for the names of classes. For example, GoodName, , not unconventional class names such as Good_Name, goodName, good_name, and GOODName.

Explicit Instance Attributes

place an instance's attributes in the __init__ method.

Use of Properties

The below-mentioned code illustrates the basic form of use of the property decorator to implement getters and setters. Once this property is created, you can use it as regular attributes using the dot (.) notation, even though it is implemented using methods under the hood.

```python
class Student:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def name(self):
        print("Getter for the name")
        return f"{self.first_name} {self.last_name}"

    @name.setter
    def name(self, name):
        print("Setter for the name")
        self.first_name, self.last_name = name.split()

student = Student("John", "Smith")
print("Student Name:", student.name)
student.name = "Johnny Smith"
print("After setting:", student.name)
```

```
Getter for the name
Student Name: John Smith
Setter for the name
Getter for the name
After setting: Johnny Smith
```

Meaningful String Representations

The __str__ method defines the string in a human-readable format and allows more customisation. Therefore, this is good for logging or displaying some information about the object. The __repr__ method defines an “official” string representation of the object, which can be used to construct the object again.

```
class Person:

    def __init__(self, person_name, person_age):
        self.name = person_name
        self.age = person_age

    def __str__(self):
        return f'Person name is {self.name} and age is {self.age}'

    def __repr__(self):
        return f'Person(name={self.name}, age={self.age})'


p = Person('John', 12)
print(p)
print(p.__str__())
print(p.__repr__())
```

```
Person name is John and age is 12
Person name is John and age is 12
Person(name=John, age=12)
```

Instance, Class and Static Methods

When creating classes, you can define three kinds of methods, *instance*, *class* and *static* methods. You need to consider which method(s) you should use for the functionalities of your problem domain.

If the methods are not concerned with individual instance objects, you have the option of using class or static methods. These two methods can be easily defined with applicable decorators @classmethod and @staticmethod. The difference between these two methods is that the class method allows you to access or update attributes related to the class, whereas the static method is independent of any instance or the class itself.

```python
from datetime import date

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # class method (create a Person object by birth year)
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # static method (check if a Person is adult or not)
    @staticmethod
    def isAdult(age):
        return age &gt; 18

p1 = Person('john', 26)
p2 = Person.fromBirthYear('john', 1996)

print(p1.age)
print(p2.age)

print(Person.isAdult(19))
```

```
26
26
True
```

Encapsulation using Private Attributes

One important way to apply encapsulation is to prefix attributes and methods with an underscore (_) or two underscores (__), denoting *protected*, and *private*, respectively.

```python
class Student:
    def get_mean_gpa(self):
        grades = self._get_grades()
        gpa_list = Student._converted_gpa_from_grades(grades)
        return sum(gpa_list) / len(gpa_list)

    def _get_grades(self):
        # fetch grades from a database
        grades = [99, 100, 94, 88]
        return grades

    @staticmethod
    def _converted_gpa_from_grades(grades):
        # convert the grades to GPA
        gpa_list = [4.0, 4.0, 3.7, 3.4]
        return gpa_list
```

Separation of Concerns & Decoupling

you want your individual classes to have separate concerns. Having these responsibilities separated, your classes become smaller, easier to manage future changes and maintainable.

Documentation

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the `def` line.
- PEP 257 describes good docstring conventions. Note that most importantly, the `"""` that ends a multiline docstring should be on a line by itself.

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

- For one-liner docstrings, keep the closing `"""` on the same line.

```
"""Return an ex-parrot."""
```