Object Oriented Python

Atividade II

INE5416 - Paradigmas de Programação

Daniel de Souza Baule (16200639)

Classes

The simplest class we can possible create in python would be something like this:

```
class Person:
  pass
```

In this case, we used the `class` statement to indicate the declaration of a new class, `person`, followed by an indented block of statements that contains the body of our class. Since we wanted a class as simple as possible, we used the `pass` statement to leave the body of our class empty.

If we wanted, we could make it a little more complex, adding attributes and methods to it, like so:

```
class Person:

  def say_hi(self):
    print('Hi')
```

Class methods are like functions, with one extra parameter, `self`. This parameter refers to the object itself, and is automaticaly provided by Python when the method is called.

Our `Person` can now say hi, but not much more... Let's give it a name so it can at least introduce itself:

```
class Person:

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

  def say_hi(self):
    print('Hi, my name is ' + self.name)
```

We used the `__init__` method to give our class a name, since it is run as an object or class is instantiated, making it useful for any initialization.

We can test this by instantiating a `Person` and calling the Method in question:

```
P = Person("John")
P.say_hi()
```

Saving it as person.py and running it in console, we are greeted by John himself:

```
$ python2 person.py
Hi, my name is John
```

Inheritance

One of the main reasons to use Object Oriented Programming is the reuse of code, and one of the ways to achieve that is through the Inheritance mechanism. Inheritance can be imagined as a Type and Subtype relationship between classes.

Let's imagine the following situation, you want to write a program to keep track of Teachers and Students in a college. Every teacher and student share some common characteristics, like name, age and address, but each also has unique characteristics of their own, since only Teachers have a salary, and only Students get grades, for example.

In order to take advantage of Inheritance and the reuse of code, we can create a class to represent every School Member:

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")
```

Every school member has their own name and age, but nothing else so far. Since both Teachers and Students are School Members, we can create classes to represent each, and inherit this shared information from our `SchoolMember` class, this is done as in the examples bellow:

```
class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))
```

```
class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Initialized Student: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Marks: "{:d}"'.format(self.marks))
```

We now have a Base Class (or Superclass) `SchoolMember` from which our Derived Classes (or Subclasses) `Teacher` and `Student` inherit. This means that any change made to our `SchoolMember` class will affect our subclasses, while changes on the subclasses have no effect on the superclass, making it much easier to add or change information down the road.

Since every `Teacher` or `Student` is also a `SchoolMember`, they can be referred as such in our code if we want, for example, to count the number of Members our School has. That is called **polymorphism**, when a subclass can be treated as an instance of the superclass.

Inheritance is also not limited to one class, both our teacher and student could also have Inherited from the `Person` class we created earlier, for example. Whe call when a class inherits from more than one superclass **multiple inheritance**, and when it only inherits from one, **simple inheritance**.

Methods

We have created Methods for our classes before, but python allows us to use different kind of methods to take better advantage of classes and inheritance

Static methods

All the methods we created so far had a `self` parameter, meaning that they received a class instance as a parameter, and would not work had the class not being instantiated.

Sometimes we write methods in classes which have no need for an instance of said class to be present for their execution, say we modified our `say_hi` method from before:

```
class Person:

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

  def say_hi():
    print('Hi`)
```

Since our person no longer introduces itself, there is no need for this method to require a named instance to be called. So we just removed the `self` parameter, making it a static method

Abstract methods

Going back to the school example, say we wanted to add a method for each `SchoolMember` to introduce themselves, since Teachers and Student have differing information to say, we could not make a simple method in our superclass, but we can't allow a SchoolMember to be created without such a method.

Abstract methods exist for this kind of situation, they are methods declared in a superclass, but implemented only on subclasses. Our `say_hi` method would be implemented in `SchoolMember` like this:

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")

    def say_hi(self):
        raise NotImplementedError
```

Meaning that unless a subclass overwrites this method, an error flag will be raised, but only as soon as the method is called, forcing its implementation.