Module 05: Object Oriented

Agenda

- Classes explanation and example
- Creating Instances
- Hide Implementation
- Class destructors
- Class Inheritance
- Override base class methods

Class

- Python defines a set of predefined types of objects, like int, string, list, method etc.
- User can define its own, user defined type of object using *class* keyword
- The name of the class immediately follows the keyword *class* followed by a colon. The name of class is a new user defined type:

```
class Test:
        'Optional class documentation string'
        pass
```

Class – cont’d

- Python classes can contain different attributes, like methods, data members, docstring, etc
- classes may define special methods, with predefined names and meaning and format like __XXX__.
  - They usually used for operators overloading and built-ins overriding
  - They are automatically invoked
  - For example:
    - __init__ - responsible for class instantiation for the newly-created class instance
    - __str__ - returns string representation of an object

self in Class's methods

- Python implements methods in a way that makes the instance, to which the method belongs, be *passed* automatically, but not *received* automatically

- The first parameter of methods is the instance the method is called on

- This parameter usually called *self*

Demo

class BankAccount Example

Lets create Bank Account class

```
class BankAccount(object):
    commission = 5.40  # class variable shared by all instances

    def __init__(self, client_name, client_id, balance):
        self._client_name = client_name
        self._client_id = client_id
        self._balance = balance

    def withdraw(self, amount):
        if self._balance - amount > 0:
            self._balance -= (amount + BankAccount.commission)
            return True
        return False

    def  deposit(self, amount):
        self. _balance += amount - BankAccount.commission
        return True

    def  __str__(self):
        return "client: {} has {} $".format (self. _clientName,  self. _balance);
```

class BankAcount – explanation

- The variable *commission* is a class variable whose value would be shared among all instances of a this class – static attribute

- It can be accessed as *BankAccount.commission* from inside or outside the class.

- The first method __init__() is a special method automatically called by python to initialize newly-created class instance

Creating Instances

- To create a new instance of a class simply specified a class name with all its parameters and assigned the result to some variable

ba = BankAccount(“Tal Moshe”, 12345678, 1000)

- This statement invokes the __init__ method
- Now we can access all the instance attributes

```
ba.deposit(500)
print(ba)                    #prints Tal Moshe has 493.60 $”
```

Hide Implementation

- Python defines private attributes by convention
- Attributes, whose name starts with an underscore (e.g. _spam) should be treated as a non-public and should be never accessed outside the class
- Attributes, whose name starts with two leading underscores (e.g. __spam) will be treated as strongly private (Python simply changes their names in outside class access)
- It is still possible to access or modify a variable that is considered to be non-public

Classes clean-up

- Since the memory in python is managed, destruction/cleanup is usually needed for resources

- One of the ways to manage object cleanup is by defining the __del__() method
- The __del__() is called when the instance is about to be destroyed by GC

```
class File:
    def __init__(self):
        self._file_object = open("some_file.txt")

    def __del__(self):
        self._file_object.close()
```

Classes clean-up – cont’d

- The problem with __del__ is that it will be called at unpredictable time (if ever) for objects with circular referencing

```
class Foo:
    def __init__(self, x):
   self.x = x
    # x => bar instance

    def __del__(self):
   print  ("end of Foo")

bar = Bar()
foo = Foo(bar)
bar = None  #will not call the __del__
```

Classes clean-up – cont’d

- The better solution for object clean-up and the recommended one is to add to the class support of context manager (*with* statement)
- Not like __del__() it has no side effects
- To use the with statement, create a class with the following methods:
  - __enter__
  - __exit__

Class Inheritance

- When we need to extend the existed class functionality and to add an extra features with a smart reuse of existed class – the solution is inheritance
- In inheritance, the class that performs the inheritance called derived class and the one who we inherits (extends) from - called base class
- The child class inherits all attributes of its parent class
- A derived class can override any method of its base class, and a method can call the method of a base class with the same name

```
class DerivedClass (BaseClass1 [, BaseClass2, ...]):
    'Optional class documentation string'
     commands
```

Class Inheritance – cont’d

- Student Bank Account example:

```
class StudentBancAccount(BankAccount):
    def __init__(self, client_name, client_id, balance, college_name):
        BancAccount.__init__(self, client_name, client_id, balance)
        self.college_name = college_name

    def __str__(self):
        return "{} {}".format(BancAccount.__str__(self), self.college_name)
```

Class Inheritance – cont'd

- Base method overriding can be done using *super:*

```
class StudentBankAccount(BankAccount):
    def __init__(self, client_name, client_id, balance, college_name):
        super().__init__(client_name, client_id, balance)
        self.college_name = college_name
```

- Note:
  - In python 3.x super syntax is much easier: super().__init__(….)

- *isinstance(obj, type)* - return true if the *obj* argument is an instance of the *type* argument, or of a subclass thereof.

Questions