Object Oriented Programming

in Python

Doc String :

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the __doc__ special attribute of that object.

doc_string.py

```
#! /usr/bin/env python

__author__ = "Panthotanvir"
__license__ = "whatever"
__docformat__ = 'rst and sphinx'

# must say class name convention and function name convention
# Class names should normally use the CapWords convention.
# Function names should be lowercase, with words separated by
underscores as necessary to improve readability.
# Use one leading underscore only for non-public methods and instance
variables
# Wrap lines so that they dont exceed 79 characters
# Use 4-space indentation, and no tabs.

class MainClass(object):
    """This docstring is for class documentation using Sphinx  and
rest for python

         The first line is brief explanation. The main idea is to
document
        the class and methods's arguments with

    - Method name and its parameters

    - and to provide sections such as **Example** using the double
commas syntax::

          :Example:

          followed by a blank line !

      followed by a blank line
```

```
    - Finally special sections such as **See Also**, **Warnings**,
**Notes**
      use the sphinx syntax (*paragraph directives*)::

          .. seealso:: blabla
          .. warnings also:: blabla
          .. note:: blabla
     .. todo:: blabla

    .. note::
        There are many other directives such as versionadded,
versionchanged,
        rubric, centered, ... See the sphinx documentation for more
details.

    Here below is the results of the :func:`function1` docstring.

    """




def method_with(self, arg1, arg2, arg3):
        """This docstring is for method documentation
        returns (arg1 / arg2) + arg3

        This is a longer explanation

        Then, you need to provide optional subsection in this order
(just to be
        consistent and have a uniform documentation. Nothing prevent
you to
        switch the order):

          - parameters using ``:param <name>: <description>``
          - type of the parameters ``:type <name>: <description>``
          - returns using ``:returns: <description>``
          - examples (doctest)

        :Example:
        >>> import template
        >>> a = template.MainClass1()
        >>> a.function1(1,1,1)
        2
```

```
        .. note:: can be useful to emphasize
            important feature
        .. seealso:: :class:`MainClass2`
        .. warning:: arg2 must be non-zero.
        .. todo:: check that arg2 is non zero.
       """
        return arg1/arg2 + arg3

if __name__ == "__main__":
   print "Its working"
   print MainClass.__doc__
   print MainClass.method_with.__doc__
```

Output:

```
Its working
“this is same as blue colored text in MainClass  class”
“this is same as blue colored text in method_with()  method in this
above class”
```

Object Oriented Programming

Features:

- **Encapsulation**
- **Polymorphism**
- **Inheritance**
- **Abstraction**

Class Definition and Object Instantiation

Class definition syntax: The class is a blueprint that defines a nature of a future object.

```
class subclass[(superclass)]:
[attributes and methods]
```

Object instantiation syntax: An *instance* is a specific object created from a particular class.

```
object = classname()
```

Attributes and methods invoke:

```
object.attribute
object.method()
```

Constructor:

A *constructor* is a special kind of method that Python calls when it instantiates an object using the definitions found in your class.

One important conclusion that can be drawn from the information so far is that, __init__() is not a constructor. The function __init__() is called immediately **after** the object is created and is used to initialize it.

Technically speaking, constructor is a method which creates the object itself. In Python, this method is **__new__()**. A common signature of this method is

```
__new__(cls, *args, **kwargs)
```

When __new__() is called, the class itself is passed as the first argument automatically.

Some important things to remember when implementing __new__() are:

- __new__() is always called before __init__().
- First argument is the class itself which is passed implicitly.
- Always return a valid object from __new__(). Not mandatory, but thats the whole point.

constructor.py

```
#! /usr/bin/env python


class Constructor(object):
    """ This class is for the example of construtor
    """
    def __init__(self, name):
        self.name = name
        print "the name is ", self.name

obj = Constructor("ipv Cloud")
```

Output:

```
the name is  ipv Cloud
```

Super():

Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class.

super can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable.

Super.py

```
#! /usr/bin/env python

class Apple(object):
    def __init__(self):
        print "init Apple"

class Ball(Apple):
    def __init__(self):
        print "init Ball"
        super(Ball,self).__init__()

class Cat(Ball):
    def __init__(self):
        print "init Cat"
        super(Cat,self).__init__()

class Dog(Cat):
    def __init__(self):
        print "init Dog"
        super(Cat,self).__init__()

d = Dog()
```

Output:

```
init Dog
init Ball
init Apple
```

*Note: the syntax changed in Python 3.0: you can just say **super().__init__()** instead of **super(ChildB, self).__init__()***

Encapsulatiion:

The *encapsulation* hides the implementation details of a class from other objects. Encapsulation is the mechanism for restricting the access to some of an object's components, this means that the internal representation of an object can't be seen from outside of the objects definition.

encapsulation.py

```
#! /usr/bin/env python

class Encapsulation(object):
     def __init__(self):
        self.var = 1    # OK to access directly
        self._var = 2   # should be considered non-public
        self.__var = 3  # considered private, name mangled

    def public(self):
      print "This is public method"

    def _non_public(self):
      print "This is non-public but still can be accessed"

    def __private(self):
      print "This is private method"

obj = Encapsulation()

print "public modifer using obj.var: ", obj.var
print "non-public modifer using obj._var: ", obj._var
print "private modifer in python using name mangling which can
accessed: ",  obj._Encapsulation__var
#print "private modifer can not access directly using obj.__var and
getting an error: ", obj.__var

obj.public()
obj._non_public()
obj._Encapsulation__private()
obj.__private()
```

Output:

```
public modifer using obj.var:): 1
non-public modifer using obj._var:  2
private modifer in python using name mangling which can accessed:  3
This is public method
This is non-public but still can be accessed
This is private method

  File "D:\python\office\encapsulation.py", line 27, in <module>
    obj.__private()
AttributeError: Encapsulation instance has no attribute '__private'
```

Polymorphism

The *polymorphism* is the process of using an operator or function in different ways for different data input.

polymorphism.py

```
#! /usr/bin/env python

__author__ = "Panthotanvir"

class Polymorphism:
    """
      This class is an example of polymorphism
   """

    def run_account(self, *args):
      """polymorphism example in method overloading
      """
      print" ".join(args)

# creating instance of a class, create an object obj of a class
Ploymorphism
obj = Polymorphism()

# passing parameter to a class
obj.run_account("one")
obj.run_account("one", "two")
obj.run_account("one", "two", "three")
```

Output:

```
one
one two
one two three
```

Duck Typing:

Duck typing is concerned with establishing the suitability of an object for some purpose. With normal typing, suitability is assumed to be determined by an object's type only. In duck typing, an object's suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object. Duck typing implies a new way to express polymorphism using duck typing.

duck_type.py

```
#! /usr/bin/env python

class Duck:
    """
    this class implies a new way to express polymorphism using duck
typing.
    This class has 2 functions: quack() and fly() consisting no
parameter.
    """
    def quack(self):
        print("Quack, quack!");

    def fly(self):
      print("Flap, Flap!");


class Person:
    def quack(self):
        print("I'm Quackin'!");

    def fly(self):
        print("I'm Flyin'!");


def in_the_forest(mallard):
```

```
    """ This function is used for express polymorphism behavior except
inheritance """
    mallard.quack()
    mallard.fly()



duck = Duck()
person = Person()

# passing object to in_the_forest() function
in_the_forest(Duck())
in_the_forest(Person())
```

Output:

```
Quack, quack!
Flap, Flap!
I'm Quackin'!
I'm Flyin'!
```

Inheritance:

Inheritance is used to indicate that one class will get most or all of its features from a parent class.

When you are doing this kind of specialization, there are three ways that the parent and child classes can interact:

1. Actions on the child imply an action on the parent.
2. Actions on the child override the action on the parent.
3. Actions on the child alter the action on the parent.

inheritance.py

```
#! /usr/bin/env python
```

```
__author__ = "Panthotanvir"
class Parent(object):
# Wrap lines so that they dont exceed 79 characters
# Use 4-space indentation, and no tabs.

    def __init__(self):  # construtor declearation
      """ Constructor delearation """

        print "parent construtor called"

    def override(self):
         print "container created"

    # Use blank lines to separate functions and classes, and larger
blocks of code inside functions.

    def imply(self):  # inheritence example
         print "swift installed and example of inheritance"

    def alter(self):
        print "parent altered"

class Child(Parent):
    def __init__(self):  # construtor declearation
        print "child construtor called"

    def override(self):  # overriding example
         """ This function is used for overriding and example of
docstring  """

         print "account created and example of overriding"

    def alter(self):
       print "Before alter"
       super(Child,self).alter()
       print "After alter"


obj = Child()

print "result of isinstance of parent class:", isinstance(obj, Parent)
print "result of isinstance of child class:", isinstance(obj, Child)

print "result of issubclass parent of child", issubclass(Parent,
Child)
```

```
print "result of issubclass child of parent", issubclass(Child,
Parent)

# Actions on the child imply an action on the parent
obj.imply()

# Actions on the child override the action on the parent.
obj.override()

# Actions on the child alter the action on the parent.
obj.alter()
```

Output:

```
child construtor called
result of isinstance of parent class: True
result of isinstance of child class: True
result of issubclass parent of child False
result of issubclass child of parent True
swift installed and example of inheritance
account created and example of overriding
Before alter
parent altered
After alter
```

Multiple Inheritance:

In languages that use multiple inheritance, the order in which base classes are searched when looking for a method is often called the Method Resolution Order, or MRO.

Python has known at least three different MRO algorithms:

1. classic

2. Python 2.2 new-style

3. Python 2.3 new-style (a.k.a. C3).

Classic MRO: classes used a simple MRO scheme: when looking up a method, base classes were searched using a simple **depth-first left-to-right** scheme.

multiple_classic.py

```
#! /usr/bin/env python

# this is an example of classic MRO

class A:

    def save(self):

        print "A.save"

class B(A): pass

class C:

    def save(self):

        print "C.save"

class D(B, C): pass

d = D()

d.save()
```

Output:

```
A.save
```

Python 2.2 new-style: The MRO would be pre-computed when a class was defined and stored as an attribute of each class object. **The computation of the MRO was officially documented as using a depth-first left-to-right traversal of the classes as before. If any class was duplicated in this search(Diamond Inheritance Problem), all but the last occurrence would be deleted from the MRO list.** So, for our below example, the search order would be D, B, C, A (as opposed to D, B, A, C, A with classic classes).

multiple_new.py

```
#! /usr/bin/env python
```

```
# this is an example of Python 2.2 new-style
class A:
  def save(self):
     print "A.save"
class B(A): pass
class C(A):
  def save(self):
     print "C.save"
class D(B, C): pass
obj = D()
obj.save()
```

Output:

```
A.save
```

Python 2.3 new-style(C3): The C3 superclass linearization of a class is the sum of the class plus a unique merge of the linearizations of its parents and a list of the parents itself. The list of parents as the last argument to the merge process preserves the local precedence order of direct parent classes.

The merge of parents' linearizations and parents list is done by selecting the first head of the lists which does not appear in the tail of any of the lists. Note, that a good head may appear as the first element in multiple lists at the same time, but it is forbidden to appear anywhere else. The selected element is removed from all the lists where it appears as a head and appended to the output list. The process of selecting and removing a good head to extend the output list is repeated until all remaining lists are exhausted. If at some point no good head can be selected, because the heads of all remaining lists appear in any one tail of the lists, then the merge is impossible to compute due to cyclic dependencies in the inheritance hierarchy and no linearization of the original class exists.

Example: https://en.wikipedia.org/wiki/C3_linearization

Problem:

```
#! /usr/bin/env python
class A(object):
        def save():
                print "A.save"
class B(object):
        def save():
                print "B.save"
class X(A, B): pass
class Y(B, A): pass
class Z(X, Y): pass
obj = Z()
obj.save()
```

Output:

```
Traceback (most recent call last):
  File "multiple_c3.py", line 11, in <module>
    class Z(X, Y): pass
TypeError: Error when calling the metaclass bases
    Cannot create a consistent method resolution
order (MRO) for bases A, B
```

This is the inconsistency problem of multiple inheritance which can not be solved using C3. C3 algorithm discards this type of multiple inheritance.

Abstraction:

Abstract base classes are a form of interface checking more strict than individual hasattr() checks for particular methods. By defining an abstract base class, you can define a common API for a

set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins to an application.

abc works by marking methods of the base class as abstract

abstract_base.py

```
#! /usr/bin/env python
import abc

class PluginBase(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def load(self, input):
        """Retrieve data from the input source and return an
object."""
        print "loaded"

    @abc.abstractmethod
    def save(self, output, data):
        """Save the data object to the output."""
        pass

#p = PluginBase()
#p.load()
```

Output:

```
Traceback (most recent call last):
  File "D:\python\office\abstract_base.py", line 16, in <module>
    p = PluginBase()
TypeError: Can't instantiate abstract class PluginBase with abstract
methods load, save
```

Abstract_subclass.py

```
#! /usr/bin/env python
import abc
from abstract_base import PluginBase

class SubclassImplementation(PluginBase):

    def load(self):
```

```
        print "load overridden"

    def save(self):
        print "save overridden"

if __name__ == '__main__':
    print 'Subclass checked:', issubclass(SubclassImplementation,
PluginBase)
    print 'Instance checked:', isinstance(SubclassImplementation(),
PluginBase)
    obj = SubclassImplementation()
    obj.load()
```

Output:

```
Subclass checked: True
Instance checked: True
load overridden
```