Python for Data Analysis

Session 2. Object-oriented programming in Python

Session agenda

- Review of object-oriented programming in Python.
- Inheritance, multiple inheritance.
- Exceptions.
- Designing object-oriented programs, the notion of design patterns.
- Examples of design patterns implementation in Python (decorators, singleton pattern and etc.).

Object-oriented programming in Python

- Python is a multi-paradigm language and, as such, supports object-oriented programming (OOP) as well as a variety of other paradigms.
- OOP is the foundation of Python – almost everything in Python is objects (e.g. built-in types, functions and etc.)
- Let us briefly revise basic Python OOP syntax and concepts

Scopes and namespaces

- Namespace – mapping from names to objects.
- Examples of namespaces:
  - set of built-in names
  - global names in a module
  - local names in a function invocation
  - set of attributes of an object
- Scope – textual region of a Python program where a namespace is directly accessible. Scopes are determined statically, but are used dynamically
- Scope hierarchy:
  - Innermost scope – local names
  - Scope of enclosing function – non-local and non-global names
  - Next-to-last scope – current module's global names
  - Outermost scope – built-in names

Scopes and namespaces - example

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)
```

Scopes and namespaces - example

```
scope_test()
print("In global scope:", spam)

After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Class definition

- Classes are defined using the *class* keyword with a very familiar structure:

```
class ClassName:
    <statement-1>
    . . .
    <statement-N>
```

- There is no notion of a header file to include so we don’t need to break up the creation of a class into declaration and definition. We just declare and use it!

Class Objects

- `__new__()` – special static method, which automatically create new instances
- `__init__()` – special method, which is automatically invoked for instances created by `__new__` method
- Together these methods forms constructors in Python

```
class SampleClass:
    """A simple example class"""
    i = 123
    def __init__(self, i):
        print('SampleClass object created!')
        self.i = i
    def f(self):
        return 'Hello world'

>>> s1 = SampleClass(1)
SampleClass object created!
```

Data Attributes

- We can also add, modify or delete attributes at will.

```
x.year = 2016 # Add an ‘year' attribute.
x.year = 2017 # Modify ‘year' attribute.
del(x.year)   # Delete ‘year' attribute.
```

- There are also some built-in functions we can use to accomplish the same tasks.

```
hasattr(x, 'year')        # Returns true if year attribute
exists
getattr(x, 'year')        # Returns value of year attribute
setattr(x, 'year', 2017) # Set attribute year to 2017
delattr(x, 'year')        # Delete attribute year
```

Methods

- Methods – functions and procedures attributed to a class and its instances
- Methods has reference (self) to the class instance, which called them
- When method is called it is passed an implicit reference to the class instance
  - Calling x.f() is equivalent to MyClass.f(x)
- This behavior can be alter by built-in function decorators staticmethod() and classmethod()

Methods

```
class SampleClass:
    i = 12345
    def __init__(self, i):
        self.i = i

    def f(self, k):
        return self.i + k

    @staticmethod
    def g():
        print('Hello world')

    @classmethod
    def h(cls, k):
        return cls.i + k
```

Variables within classes

- Variables in a class fall under one of two categories:
  - Class variables - shared by all instances.
  - Instance variables - unique to a specific instance.

```
>>> class Dog:
...     kind = 'canine' # class var
...     def __init__(self, name):
...         self.name = name # instance var
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind # shared by all dogs
'canine'
>>> e.kind # shared by all dogs
'canine'
>>> d.name # unique to d
'Fido'
>>> e.name # unique to e
'Buddy'
```

Built-in Attributes

Besides the class and instance attributes, every class has access to the following:

- **`__dict__`**: dictionary containing the object's namespace.
- **`__doc__`**: class documentation string or None if undefined.
- **`__name__`**: class name.
- **`__module__`**: module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **`__bases__`**: a possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Operator overloading

- Operator overloading in Python is implemented providing explicit definitions to special methods (magic methods)
- Magic methods are invoked when corresponding syntax constructs are invoked (e.g. arithmetic operations, subscripting, comparison and etc.)
- All magic method’s names follow naming convention: __methodname__
- Full list of magic method’s names can be found in documentation

Inheritance

- The basic format of a derived class is as follows:

```
class
DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

- In the case of `BaseClass` being defined elsewhere, you can use `module_name.BaseClassName`

Inheritance - example

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet’s name is " + str(self.name)

class Dog(Pet):
        def __init__(self, name, age, breed):
                Pet.__init__(self, name, age)
            self.breed = breed
   def get_breed(self):
            return self.breed
```

Inheritance

- Python has three inheritance related built-in functions:
  - `isinstance(`*`object`*`, `*`classinfo`*`)` returns true if *object* is an instance of *classinfo* (or some class derived from classinfo).
  - `issubclass(`*`class`*`, `*`classinfo`*`)` returns true if *class* is a subclass of *classinfo*.
  - `super([type[, object-or-type]])` return a proxy object that delegates method calls to a parent or sibling class of *type*

Inheritance - example

```
class Dog(Pet):
        def __init__(self, name, age, breed):
                super().__init__(name, age)
            self.breed = breed
    def get_breed(self):
            return self.breed

>>> mydog = Dog('Ben', 1, 'Maltese')
>>> isinstance(mydog, Dog)
True
>>> isinstance(mydog, Pet)
True
>>> issubclass(Dog, Pet)
True
>>> issubclass(Pet, Dog)
False
```

Multiple inheritance

- You can derive a class from multiple base classes like so:

```
class DerivedClassName(Base1, Base2,
Base3):
    <statement-1>
    ...
    <statement-N>
```

- Attribute resolution is performed by searching DerivedClassName, then Base1, then Base2, etc.

Private variables

- There is no strict notion of a private attribute in Python.
- However, if an attribute is prefixed with a single underscore (e.g. `_name`), then it should be treated as private. Basically, using it should be considered bad form as it is an implementation detail.
- To avoid complications that arise from overriding attributes, Python does perform *name mangling*. Any attribute prefixed with two underscores (e.g. `__name`) is automatically replaced with `_classname__name`.
- *Bottom line*: if you want others developers to treat it as private, use the appropriate prefix.

Name mangling

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

class MappingSubclass(Mapping):
    def update(self, keys, values):
        for item in zip(keys, values):
            self.items_list.append(item)
```

Name mangling

```
>>> x = MappingSubclass([1, 2, 3])
---------------------------------------------------------------
TypeError                    Traceback (most recent call last)
<ipython-input-56-4ff7bd5497be> in <module>()
----> 1 x = map(MappingSubclass([1,2,3]))

<ipython-input-54-7e37656fc6f0> in __init__(self, iterable)
      2     def __init__(self, iterable):
      3         self.items_list = []
----> 4         self.update(iterable)
      5     def update(self, iterable):
      6         for item in iterable:

TypeError: update() missing 1 required positional argument:
'values'
```

Name mangling

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)
    __update = update # private copy of original update()

class MappingSubclass(Mapping):
    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Name mangling

```
>>> x = MappingSubclass([1,2,3])
>>> x.items_list
[1, 2, 3]
>>> x.update(['key1', 'key2'], ['val1', 'val2'])
>>> x.items_list
[1, 2, 3, ('key1', 'val1'), ('key2', 'val2')]
```

Structs in python

- You can create a struct-like object by using an empty class.

```
>>> class Struct:
...     pass
...
>>> node = Struct()
>>> node.label = 4
>>> node.data = "My data string"
>>> node.next = Struct()
>>> next_node = node.next
>>> next_node.label = 5
>>> print node.next.label
5
```

Exceptions

- Errors that are encountered during the execution of a Python program are *exceptions.*

```
>>> print(spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

- There are a number of built-in exceptions (complete list can be found in documentation)

Handling exceptions

- Explicitly handling exceptions allows us to control otherwise undefined behavior in our program, as well as alert users to errors. Use try/except blocks to catch and recover from exceptions.

```
>>> while True:
...     try:
...         x = int(input("Enter a number: "))
...         break
...     except ValueError:
...         print("Not a valid number. Try again.")
...
Enter a number: two
Ooops !! That was not a valid number. Try again.
Enter a number: 100
```

Handling exceptions

- The try/except clause options are as follows:

```
except:
#Catch all (or all other) exception types
except name:
#Catch a specific exception only
except name as value:
#Catch the listed exception and its instance
except (name1, name2):
#Catch any of the listed exceptions
except (name1, name2) as value:
#Catch any of the listed exceptions and its instance
else:
#Run if no exception is raised
finally:
#Always perform this block
```

Handling Exceptions

- There are a number of ways to form a try/except block.

```
>>> while True:
...     try:
...         x = int(input("Enter a number: "))
...         break
...     except ValueError:
...         print("Not a valid number. Try again.")
...     except (TypeError, IOError) as e:
...         print(e)
...     else:
...         print("No errors encountered!")
...     finally:
...         print("We may or may not have encountered errors…")
```

Raising an exception

- Use the raise statement to force an exception to occur. Useful for diverting a program or for raising custom exceptions.

```
>>>try:
...     raise IndexError("Index out of range")
... except IndexError as ie:
...     print("Index Error occurred: ", ie)
Index Error occurred:  Index out of range
```

Creating an exception

- User exceptions can by created by a new exception class derived from the *Exception* class.

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print( 'My exception occurred, value:', e )
...
My exception occurred, value: 4
```

Assertions

- Use the assert statement to test a condition and raise an error if the condition is false.

```
>>> assert a == 2
```

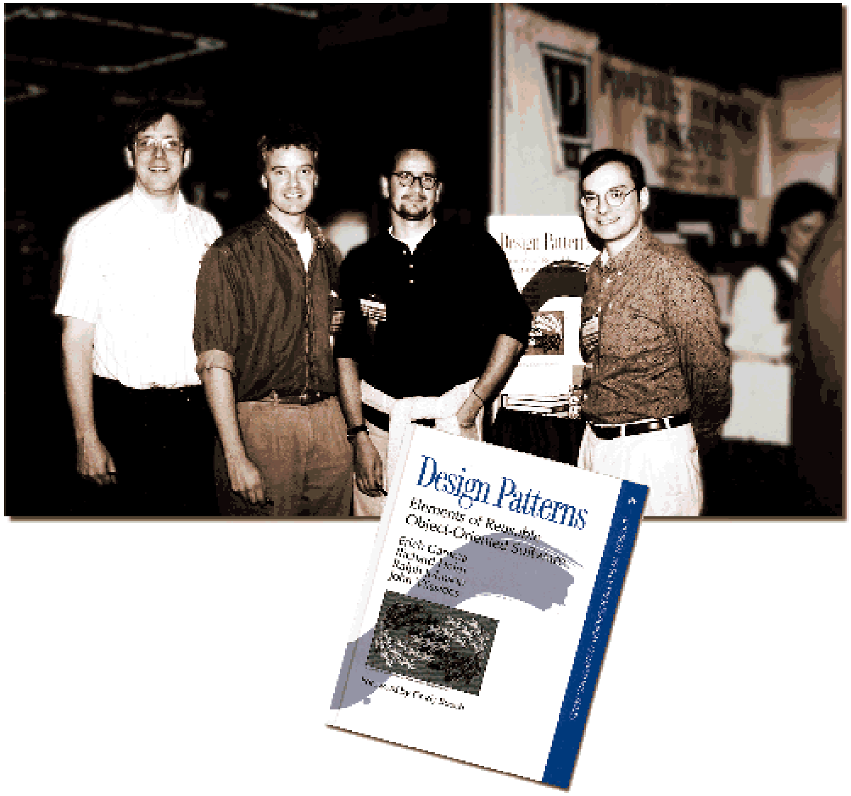
- Assertion statement is equivalent to

```
>>> if not a == 2:
...     raise AssertionError()
```

- Assertions can be deactivated by configuring Python interpreter (e.g. –O optimization flag switches off assertions)

Design patterns

- Everything starts with Gang of Four (GoF)
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns -- Elements of Reusable Object-Oriented Software*.



Design patterns

- Design patterns are a common way of solving well known problems in design of object-oriented programs.
- Two main principles are in the bases of the design patterns defined by the GoF:
  - Program to an interface not an implementation.
  - Favor object composition over inheritance.
- Design patterns allows to create reusable and easily extensible code

A note of caution

- Design patterns has received significant amount of criticism
- When applied light-mindedly can lead to complex code, which is hard to maintain and modify
- On the bright side:
  - Concepts behind the patterns are general and give good insight into how complex object-oriented programs can be developed
  - Other programmers use them and it is good to recognize a pattern once we encounter it
  - Programming languages have some design patterns integrated(implemented)

Design patterns classification

- Structural patterns
  - Concern class and object composition and design
- Creational patterns
  - Create objects for you, rather than having you instantiate objects directly
- Behavioral patterns
  - Deal with specific communication between objects

Structural patterns

Pattern name	Description
Adapter	Allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
Bridge	Decouples an abstraction from its implementation so that the two can vary independently
Composite	Composes zero-or-more similar objects so that they can be manipulated as one object
Decorator	Dynamically adds/overrides behavior in an existing method of an object
Façade	Provides a simplified interface to a large body of code
Flyweight	Reduces the cost of creating and manipulating a large number of similar objects
Proxy	Provides a placeholder for another object to control access, reduce cost, and reduce complexity

Creational patterns

Pattern name	Description
Abstract factory	Groups object factories that have a common theme
Builder	Constructs complex objects by separating construction and representation
Factory method	Creates objects without specifying the exact class to create
Prototype	Creates objects by cloning an existing object
Singleton	Restricts object creation for a class to only one instance

Behavioral patterns

Pattern name	Description
Chain of responsibility	Delegates commands to a chain of processing objects
Command	Creates objects that encapsulate actions and parameters
Interpreter	Implements a specialized language
Iterator	Accesses the elements of an object sequentially without exposing its underlying representation
Mediator	Allows loose coupling between classes by being the only class that has detailed knowledge of their methods
Memento	Provides the ability to restore an object to its previous state

Behavioral patterns

Pattern name	Description
Observer	Publish/subscribe pattern which allows a number of observer objects to see an event
State	Allows an object to alter its behavior when its internal state changes
Strategy	Allows one of a family of algorithms to be selected on-the-fly at runtime
Template method	Defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior
Visitor	Separates an algorithm from an object structure by moving the hierarchy of methods into one object
Observer	Publish/subscribe pattern which allows a number of observer objects to see an event