CS 3101-1 - Programming Languages: Python
Lecture 3: OOP, Modules, Standard Library I

Daniel Bauer (bauer@cs.columbia.edu)

October 1 2014
Contents

Object Oriented Python

Modules
Object Oriented Python

Modules
Object Oriented Programming (OOP) is at the core of Python.

- Everything is an object!
- Operations are methods on objects.
- Modularization.

We have seen examples of objects already:
- Objects of built-in data types (int, str, list, dict ... ).
- Functions.

- Can create our own types (classes).
- Python does not enforce OOP (unlike Java), but we need to understand at least what is going on.
Objects, Attributes, Methods, Classes

- Classes:
  - User defined types of objects (including their methods, attributes, relations to other objects).
  - Can be instantiated into an object / is a ‘blueprint’ that describes how to build an object.

Knights can eat, sleep, have a favorite color, and a title.
Objects, Attributes, Methods, Classes

- **Classes:**
  - User defined types of objects (including their methods, attributes, relations to other objects).
  - Can be instantiated into an object / is a ‘blueprint’ that describes how to build an object.

Knights can eat, sleep, have a favorite color, and a title.

- **Objects:** Grouping of data and behavior (code) into a functional ‘package’.

```python
l = Knight()
```

```python
l.name = "Launcelot",
```

```python
l.title = "the brave"
```
Objects, Attributes, Methods, Classes

- **Classes:**
  - User defined types of objects (including their methods, attributes, relations to other objects).
  - Can be instantiated into an object / is a ‘blueprint’ that describes how to build an object.

  Knights can eat, sleep, have a favorite color, and a title.

- **Objects:** Grouping of data and behavior (code) into a functional ‘package’. `l = Knight()`

- **Attributes:** data fields of the object.

  `l.name = "Launcelot", l.title = "the brave"` ...
Objects, Attributes, Methods, Classes

- **Classes:**
  - User defined types of objects (including their methods, attributes, relations to other objects).
  - Can be instantiated into an object / is a ‘blueprint’ that describes how to build an object.

  Knights can eat, sleep, have a favorite color, and a title.

- **Objects:** Grouping of data and behavior (code) into a functional ‘package’. `l = Knight()`

- **Attributes:** data fields of the object.

  `l.name = "Launcelot", l.title = "the brave" ...`

- **Methods:** functions that belong to the object and can access and manipulate the object’s data. All Methods are attributes.

  `launcelot.eat(food), launcelot.sleep() ...`
Class Definitions with `class`

- Class definitions contain methods (which are functions defined in the class’ scope), class attributes, and a docstring.

```python
class Knight(object):
    """ A knight with two legs, who can eat food. """
    legs = 2  # attribute

    def __init__(self):
        self.stomach = []

    def eat(self, food):
        self.stomach.append(food)
        print('Nom nom.')

>>> Knight.legs
2
>>> Knight.eat
<unbound method Knight.eat>
```

- Classes are objects too. Methods and attributes are are `attributes of the class object!`

```python
>>> Knight.legs
2
>>> Knight.eat
<unbound method Knight.eat>
```
Instantiating a Class to an Instance Object

class Knight(objec):

    legs = 2

    def __init__(self):
        self.stomach = []

    def eat(self, food):
        self.stomach.append(food)
        print('Nom nom."

Functions are instantiated into instance objects by calling a class object.

>>> k = Knight()
>>> k
<__main__.Knight object at 0x100e82c10>
>>> type(knight)
<class '__main__.knight'>
Calling Bound Methods on Instance Objects

```python
class Knight(object):
    legs = 2

    def __init__(self):
        self.stomach = []

    def eat(self, food):
        self.stomach.append(food)
        print('Nom nom.')

>>> k = Knight()
>>> k.eat("cheese")
Nom Nom.
>>> k.stomach
["cheese"]
```

- Instance objects contain bound methods as attributes.
- The first parameter in a method definition (‘self’) is bound to the instance object when a bound method is called.
Setting Up Instance Attributes and `__init__`

```python
class Knight(object):
    legs = 2

    def __init__(self, name):
        self.stomach = []
        self.name = name

    def eat(self, food):
        self.stomach.append(food)
        print('Nom nom.

>>> k = Knight("galahad")
>>> k.name
"galahad"
```

- The special method `__init__` is called automatically when an instance is created.
- Main purpose: `__init__` sets up attributes of the instance.
Class vs. Instance Attributes

```python
class Knight(object):
    legs = 2

    def __init__(self, name):
        self.stomach = []
        self.name = name

    def eat(self, food):
        self.stomach.append(food)
        print('Nom nom.')

>>> k = Knight("black")
>>> k.legs
2
>>> k.legs = 0
>>> k.legs
0
>>> Knight.legs
2
```

- Class attributes are visible in instances.
- re-binding attribute names in an instance creates a new instance attribute that hides the class attribute.
Inheritance

- Classes inherit from one or more base classes.
- Look up methods and class attributes in base classes if not found in class.

```python
class Knight(Warrior, TitleBearer):
    def __init__(self, name):
        ...
    def go_on_quest(self, quest):
        ...

>>> k1 = Knight("Galahad")
>>> k2 = Knight("Robin")
>>> k1.fight(k2)
>>> k2.eat("coconut")
```
Multiple Inheritance - Method Resolution Order

- Problem: Which `eat` method to use?
- Use first attribute found according to method resolution order.

```python
class Knight(Warrior, TitleBearer):
    def __init__(self, name):
        ...
    def go_on_quest(self, quest):
        ...

>>> k1 = Knight("Galahad")
>>> k2 = Knight("Robin")
>>> k1.fight(k2)
>>> k2.eat("coconut")
```
# Built-in Types as Base Classes

```python
class FlipDict(dict):
    """ A dictionary that can be inverted. """

def flip(self):
    """ Return a dictionary of values to sets of keys. """
    res = {}
    for k in self:
        v = self[k]
        if not v in res:
            res[v] = set()
            res[v].add(k)
    return res

>>> x = FlipDict((1,'a'), (2,'b'), (3,'a'))
>>> x
{1: 'a', 2: 'b', 3: 'a'}
>>> x.flip()
{'a': set([1, 3]), 'b': set([2])}
```
Polymorphism (1)

Inheritance allows to override methods of base classes in different ways.

May only know exact type of objects at runtime.

Can verify if type has a certain (transitive) base class.

```python
class Shape(object):
    def perimeter():
        """ abstract method """
        return

class Square(Shape):
    def perimeter():
        return self.side ** 2

class Circle(Shape):
    def perimeter():
        return self.radius * 2 * math.pi
```

```python
>>> x = Circle()
>>> isinstance(x, Shape)
True
```
Calling Base Class Implementations of Overloaded Methods

- Sometimes we want to call the base class version of a method.
- This is often the case for `__init__`.
- Use unbound method attribute of the base class.

```python
class Animal(object):
    def __init__(self, heads, tails, legs):
        self.heads = heads
        self.tails = tails
        self.legs = legs

class Cat(Animal):
    def __init__(self):
        Animal.__init__(self, 1, 1, 4)

>>> x = Cat()
>>> x.tails
1
```
Polymorphism (2) - Duck Typing

- Python is dynamically typed. Any variable can refer to any object.
- Explicit type checking (isinstance) at runtime is considered bad style.
- Instead use ‘duck typing’! (plus error handling).

Duck Typing

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

- As long as an object implements functionality, its type does not matter.
- Example: Equality (==), convert to string (str(), iterators)
Special methods (1)

- `__init__(self)` is called when an instance is created.
- `__str__(self)` returns a string representation of the object.
- `__repr__(self)` returns the ‘official’ string representation.

```python
class FarmersMarket(object):
    def __init__(self, name, state, town, zipc):
        self.name = name
        self.state, self.town, self.zipc = state, town, zipc
    def __str__(self):
        return '{0}
{1} , {2} {3} '.format(name, town, state, zipc)
    def __repr__(self):
        return 'FarmersMkt:<{0}:{1},{2} {3}>'.format(name, town, state, zipc)
```

```python
>>> str(mkt1)
CU Greenmarket
New York, NY 10027
>>> mkt1
FarmersMkt:<CU Greenmarket:New York,NY 10027>
```
Special methods (2) - Comparisons

- `__eq__(self, other)` used for `==` comparisons.
- `__lt__(self, other)` used for `<` comparisons.
- `__le__`, `__gt__`, `__ge__`, `__ne__`

```python
class Shape(object):
    def __eq__(self, other):
        return self.area() == other.area()
    def __lt__(self, other):
        return self.area() < other.area()
    def __gt__(self, other):
        return self.area() > other.area()

class Rectangle(Shape):
    def __init__(self, l, w):
        self.l, self.w = l, w
    def area(self):
        return self.l * self.w

>>> Rectangle(2,3) == Rectangle(1,6)
True
```
Special methods (3) - Comparisons in Python 2.7

This has been deprecated in Python 3! Do not use!

- If none of the previous comparisons is defined, `__cmp__(self, other)` is called.
  - Return 0 if `self` and `other` are equal
  - Negative integer if `self < other`
  - Positive integer if `self > other`

```python
class Shape(object):
    def __cmp__(self, other):
        return self.area() - other.area()

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return self.r * 2 * math.pi

>>> Circle(1) < Rectangle(4,2)
True
```
Special methods (4) - Hash functions

- `__hash__(self)` returns a hash code for the object.
- Hash code is used to index dictionaries / sets.
- Default: object id `id(object)`
- Every function that implements `__hash__` has to implement `__cmp__` or `__eq__` and all equal object must have the same hash code.

```python
class Rectangle(object):
    def __init__(self, w, h):
        self.w, self.h = w, h
    def __hash__(self):
        return 17 * self.h + 31 * self.w
    def __eq__(self, other):
        return self.__hash__() == other.__hash__()
    def __repr__(self):
        return "Rect:({0},{1})".format(self.w, self.h)
```

```python
>>> set([Rectangle(1,2), Rectangle(1,2), Rectangle(2,1)])
set([Rect:(1,2), Rect:(2,1)])
```
Special methods (4) - Hash functions

- \texttt{\_\_hash\_\_(self)} returns a hash code for the object.
- Hash code is used to index dictionaries / sets.
- Default: \texttt{id(object)}
- Every function that implements \texttt{\_\_hash\_\_} has to implement \texttt{\_\_cmp\_\_} or \texttt{\_\_eq\_\_} and all equal object must have the same hash code.

```python
class Rectangle(object):
    def \_\_init\_\_(self, w, h):
        self.w, self.h = w, h
    def \_\_hash\_\_(self):
        return 17 * self.h + 31 * self.w
    def \_\_eq\_\_(self, other):
        return self.\_\_hash\_\_() == other.\_\_hash\_\_()
    def \_\_repr\_\_(self):
        return "Rect:({0},{1})".format(self.w, self.h)
```

```bash
>>> set([Rectangle(1,2), Rectangle(1,2), Rectangle(2,1)])
set([Rect:(1,2), Rect:(2,1)])
```
Implementing Iterators - The Iterator Protocol

- Need an `__iter__(self)` method that returns the iterator itself.
- Need a `next(self)` method that returns the next element.
  - If no element is left, calling `next(self)` raises a `StopIteration` exception.

```python
class ReverseIterLst(list):
    def __iter__(self):
        self.index = len(self)
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        else:
            self.index -= 1
            return \\n            self[self.index]
```

```python
>>> l = \\ReverseIterLst([1,2,3])
>>> for x in l:
...    print x
...
3
2
1
```
Object Oriented Python

Modules
Program can consist of multiple modules (in multiple files).
- Independent groupings of code and data.
- Can be re-used in other programs.
- Can depend on other modules recursively.

So far we have used a single module:
- Used the interpreter’s interactive mode.
- Written single-file python programs.

We have seen examples:

```python
import sys
import random
import antigravity
```
Structure of a Module File

- A module corresponds to any Python source file (no special syntax).
- The module 'name' is typically in file 'name.py'.
- File can contain class and function definitions, code.
- Can contain a doc string (string in first nonempty line).

example_module.py

""" A module to illustrate modules. """

class A(object):
    def __init__(self, *args):
        self.args = args

def quadruple(x):
    return x**4

x = 42
print("This is an example module.")
Importing and Using a Module

- `import modulename [as new_name]` imports a module and creates a module object (use near top of file).
- Unindented top-level code in module is executed (including class and function definitions).
- All defined variables/names (including class and function names) become attributes of the module.

```python
>>> import example_module as ex
This is an example module.
>>> ex.x
42
>>> a = ex.A(1,2,3)
```
Importing Specific Attributes of a Module

- `from modulename import attr [as new_name]` loads the module and makes `attr` (a class, function, variable...) available in the namespace of the importing module.

```python
>>> from example_module import A
This is an example module.
>>> a = A(1,2,3)
>>> testmodule # we don’t get a module object
NameError: name ‘testmodule’ is not defined
```

- Can also import all attributes (considered bad style!)

```python
>>> from example_module import *
This is an example module.
>>> a = A(1,2,3)
```
Main Functions

- Problem: Modules often contain some test code that we do not want to run every time it is imported.
- Modules contain a special attribute \_\_name\_\_
- \_\_name\_\_ == '\_\_main\_\_' if this module is the first one loaded (i.e. passed to the interpreter on the command line).
- Always use the following main function idiom:

```python
somemodule.py

def main():
    ...

if __name__ == '__main__':
    main()
```
Packages

- Packages are modules that contain other modules as attributes.
- Packages allow you to create trees of modules.
- A package corresponds to a directory. (i.e. the package `graphtools.directed.tree` is in the file `graphtools/directed/tree.py`).
- Package directories must contain a file `__init__.py` containing the module code (even if its empty).