Python Objects, Classes, Exceptions

CS156

Chris Pollett

Sep 15, 2014

Outline

- Objects and Classes
- Quiz
- Exception
- Modules
- Documentation Strings and Help
- Unit Testing

Introduction

- Last week, we were talking about the $A^\star$-algorithm
- We then switched gears and started introducing the Python programming language -- the language we will be coding in this semester.
- We talked about using interactive mode, running python from files, expressions, assignments, conditionals, looping, I/O, File I/O, strings, lists, tuples, sets, dictionaries, functions, generators, co-routines.
- After such a world-wind tour at the end of last day, out-of-breath, we mentioned Python classes.
- Today, we are going to start from that slide, talk about objects and classes in Python, then finish up the rest of our brief introduction to Python.

Objects and Classes

- We have already seen that lists, dicts, etc are objects and we can invoke methods on them much as we do in other languages. For instance, my_list.append("hello")
- Given an object we can see its methods by calling dir(my_obj). For example, a=[]; dir(a); gives:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

- You can see append in the list. Special methods such as operator overloads have '__' around them. For example, __add__ in the above corresponds to the + operator. We can actually use the + operator in a cumbersome way as: a.__add__([1, 2]).
- If you were writing your own class overriding __add__ would give you operator overloading of "+" for operations of the form my_obj + something_else. To override something_else + my_obj you would need to implement __radd__. To override += use overrider __iadd__, etc.
- __init__ represents the constructor.
- To define new classes we using the keyword class. For example,

```
class Stack(object): #this says stack inherits from object
    a_class_variable = 5 # this var behaves like a Java static var
    def __init__(self): #self = this in Java
        self.stack = [] #now stack is a field variable of Stack
        #in general using self.field_var is how we declare and
        #instantiate a instance variable
    def push(self, object): #the first argument of any method
        self.stack.append(object) # is the object itself
    def pop(self):
        return self.stack.pop()
    @property #properties are attributes that computes its value when accessed
    def length(self):
        return len(self.stack)
```

- To instantiate an instance of this class and use it one could do:

```
my_instance = Stack()
my_instance.push("hello")
print my_instance.length
print isinstance(my_instance, object) #returns True
```

```
print issubclass(Stack, object) #returns True
#type(my_instance) returns Stack
...
#etc
```

Inheritance, static methods, abstract classes.

- Stack above is almost identical to the built-in list object except for the push method. We could have instead inherited from list using:

```
class Stack(list):
    def push(self, object):
        #could refer to parent by using syntax list.some_method_of_list
        #or use super(list, self).some_method_of_list
        self.append(object)
```

- Python supports multiple inheritance: class MyClass(A, B, ...):
- We can use the super mechanism to distinguish between parent members. Python also uses name mangling on methods beginning with "__". The method __foo in class A would be mangled to _A__foo and so would be different from a __foo in class B.
- Static methods of classes can be created and used using the syntax:

```
class MyClass:
   @staticmethod
   def my_method():
      #some code

MyClass.my_method() #similar to Java
```

- Similarly, abstract methods and properties can be declared as:

```
#load in abstract class module
from abc import ABCMeta, abstractmethod, abstractproperty
class MyClass:
    __metaclass__ = ABCMeta # a metaclass is a class object that knows how to
                            # create other class objects. The default metaclass
                            # is type (Python 3, types.ClassType in Python 2).
                            # Here we are assigning ABCMeta
                            # to be used rather than type
                            # In Python 3, write MyClass(metaclass=ABCmeta)
    @abstractmethod
    def my_abstract_method(self):
       pass
    @abstractproperty
    def my_abstract_property(self):
       pass
```

Tic-tac-toe Board Example

- Here is another example of creating and using a class:

```
class Board:
    def __init__(self):
        self.board = ["...", "...", "..."]
    def add(self, player_piece, x, y):
        if player_piece != "x" and player_piece != "o":
            raise RuntimeError("Piece must be x or o")
        row = ""
        for i in range(3):
            if i == y:
                row += player_piece
            else:
                row += self.board[x][y]
        self.board[x] = row
    def draw(self):
        for i in range(3):
            print self.board[i]

b = Board()
b.add("x", 1, 1)
b.draw()
print
b.add("o", 0, 1)
b.draw()
```

Quiz

Which of the following is true?

1. In $A^\star$ search the frontier nodes are stored in a priority queue ordered by $f = c + h$.
2. Python has a switch/case construct.
3. To define a function in Python we use the Python keyword `function`.

Exceptions

- Like Java, Python has a syntax for exception handling, try-except-finally statements, which have the following syntax:

```
try:
    statement_block_0
except SomeError1 as error_name1:
    statement_block_1 #executed if a SomeError1 occurs
except SomeError2 as error_name2:
    statement_block_2
...
finally:
    statement_block_n # always gets executed
```

- For example,

```
try:
  f = open("file.txt", "r")
except IOError as e:
  print e
```

- To signal an exception you use the raise keyword:

```
raise RuntimeError("Something bad just happened")
```

- You define new exceptions by extending exception:

```
class MyException(exception): pass

#now could use as:
raise MyException("Whoa! A MyException occurred!")

#more control can be had by overriding __init__
#here we define an exception taking two arguments
class MyException2(exception):
    def __init__(self, errno, msg):
        self.args = (errno, msg)
        self.errno = errno
        self.errmsg = msg

raise MyException2(403, "Access Forbidden")
```

- An except: block for FooError will also respond to any subclasses of FooError

Modules

- Typically, when you code larger projects you split them into several files.
- If you want to use code in one file in another file in Python you use **import**.
- For example, in one file div.py you might have a collection of functions about division. This would be your module.
- To use this module in another file, you could put:

```
import div #notice not div.py
a, b = div.divide(198, 15) #notice function in div.py have to be prefixed with div.
```

- What if we want to use a different prefix then div? We could do:

```
import div as foo #now foo is the prefix
```

- What if we don't want to keep writing div.some_function ?

```
from div import divide #from div import *; would import all functions
print divide(198, 15)
```

- Modules can be bundled together in so-called **packages**, but I won't get into that unless we really need it.

Documentation Strings and Help

- The first statement of a module, class or function definition can be a string called a **documentation string**.
- For example,

```
def fact(n):
    "This function computes a factorial" #can use triple quoted strings
    if(n <= 1): return 1
    else: return n * fact(n - 1)
```

- The documentation string of such an object is associated with its __doc__ property which can be printed, etc:

```
print fact.__doc__
```

- Python has the built-in function help() which can be run from python in interactive mode to get information about modules.
- For example, I could put the above fact code into a file test.py. Then in interpretative mode type:

```
import(test)
help(test.fact)
```

  This would go to a screen that prints the documentation string.
- Python also comes with a pydoc command that can also give documentation information. For example, at a Unix prompt (not in Python interpretative mode), I could type:

```
pydoc test.fact
```

  and get the same result as help(test.fact) before

Unit Tests

- Python come with an interesting unit testing feature which can be directly used from the doc string.
- It also has a more robust unittest module which if you are interested in you can lookup.
- In a unit test we are trying to demonstrate that a software unit such as a function, class, module behaves as it should.
- Unit testing for a class or module can be often thought of as unit testing each of its methods/components, so we consider for now the function case.
- To test a function we work out some input output pairs for our function by hand, and check that our Python on those inputs gives the corresponding outputs.
- We usually want to choose our tests so that they cover all the edges cases and paths through the code. The degree to which we do this is called the **coverage** of our test.

doctest

- The unit testing feature associated with the doc string is invoked by using the doctest module. Below is an example

```
def my_pow(x, y):
    """ Computes x raised to the power of y
        For example,
        >>> my_pow(3, 4)
        81

        Here are some base cases:
        >>> my_pow(0,0)
        1
        >>> my_pow(1,0)
        1
        >>> my_pow(2,0)
        1

        Since our method does something different when y is odd or even
        we have a test for each case:
        >>> my_pow(2,4)
        16
        >>> my_pow(2,5)
        32
    """
    if y == 0:
        return 1
    tmp = my_pow(x, y/2)
    tmp = tmp * tmp
    if y == 2 * (y / 2) :
        return tmp
    else :
        return tmp * x
if __name__ == '__main__':
     #run as the main program from command line (as opposed to imported by
     #someone else
     import doctest
     doctest.testmod()
```

- The if __name__ == '__main__': at the end of the above checks if this file is being run by itself rather than being imported, if so we import doctest and then run the test cases, doctest.testmod().
- Running the test cases, means doctest will scan the file for doc strings. If it finds a line in such as string beginning with >>> followed by a space, it assumes that's an input to be tested and the next line should contain the output. After a sequence of such input output pairs to return to comment mode doctest expects a blank line.
- If we run python my_pow.py which has the above, the doctest will run six tests. If they all pass it outputs nothing, otherwise, it outputs the number that passed and

information about which failed.