Imperative programming with Python

Class #6

Facundo Carreiro

ILLC, University of Amsterdam

January 2015

Polymorphism

- Let's write a function that counts the number of occurrences of an element in a list (once again)

```
def count(e, l):
    c = 0
    for i in l:
        if e == i:
            c = c + 1
    return c
```

```
>>> count(1, [4,1,'a',1])
2
```

Polymorphism

- Suppose we mess up and use it with a string

```
>>> count('a', 'hella␣good')
1
```

- ...or with tuples

```
>>> count(7, (4,7,9,2,5,7,2,7))
3
```

- A function is called *polymorphic* when it works (as intended) with several different data types.
- If the assumptions on the types are properly specified, it is an important way to reuse code and encapsulate an action.

Polymorphism

- In this case, what are we asking `e` and `l` for?

```
def count(e, l):
    c = 0
    for i in l:
        if e == i:
            c = c + 1
    return c
```

- `l` should be iterable.
- `e` should be comparable.

Inheritance

```
class Log(object):
    def __init__(self):
        print 'Initializing␣Log'
        self.log = []

    def add(self, m):
        self.log.append(m)

    def see(self):
        print self.log
```

- Inheritance is a feature to generate new classes by specializing existing ones.

```
class DebugLog(Log):
    def add(self, m):
        print m
        Log.add(self, m)
```

- The `DebugLog` class *'inherits'* all attributes and methods from `Log`.
- It defines an *'is a'* subtype relationship.

Inheritance: method resolution order

- When calling a method, the most specific one gets executed.

```
>>> d = DebugLog()
Initializing Log
```

- When we create a `DebugLog` object `Log.__init__` gets called.

```
>>> d.add('Something happened')
Something happened
```

- `DebugLog.add` overrides the base method `Log.add`.

```
>>> d.see()
['Something happened']
```

- As `DebugLog.see` doesn't exist, `Log.see` gets executed.

Inheritance: a slight variant

- Suppose we want to show a message when we start logging

```
class DebugLog(Log):
    def __init__(self):
        print 'Logging␣started␣at␣' + time.strftime('%H:%M:%S')
    ...
```

```
>>> d = DebugLog()
Logging started at 19:32:50
```

- Then, somewhere in our code, we add a line to the log

```
>>> d.add('Log␣this␣line')
Log this line
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in add
  File "<stdin>", line 7, in add
AttributeError: 'DebugLog' object has no attribute 'log'
```

- The `log` attribute doesn't exist because `Log` was not initialized!
- Suggested HW: Fix it.

Inheritance: a more complex example

```
class FileLog(Log):
    def __init__(self, fn):
        print 'Initializing␣FileLog'
        Log.__init__(self)
        self.filename = fn

        # reload log into memory
        f = open(self.filename)
        self.log = [line.strip() for line in f]
        f.close()

    def add(self, m):
        Log.add(self, m)
        f = open(self.filename, 'a')
        f.write(m + "\n")
        f.close()
```

- In this case, the base class is properly initialized

```
>>> f = FileLog('log.txt')
Initializing FileLog
Initializing Log
```

Inheritance: a more complex example (v2)

- Look at

```
        # reload log into memory
        f = open(self.filename)
        self.log = [line.strip() for line in f]
        f.close()
```

- Design discussion: is it ok to access `self.log` directly?
- It is better if we use `Log.add`.

```
class FileLog(Log):
    def __init__(self, fn):
        print 'Initializing␣FileLog'
        Log.__init__(self)
        self.filename = fn

        # reload log into memory
        f = open(self.filename)
        for line in f:
            Log.add(self, line.strip())
        f.close()
    ...
```

Exceptions

We have seen that many things could go wrong during runtime

```
>>> f = open('inexistent.txt')
IOError: [Errno 2] No such file or directory: 'inexistent.txt'
```

```
>>> int('not an int')
ValueError: invalid literal for int() with base 10: 'not an int'
```

```
>>> 10 * (1/0)
ZeroDivisionError: integer division or modulo by zero
```

```
>>> 4 + spam*3
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
TypeError: cannot concatenate 'str' and 'int' objects
```

Exceptions

- These errors are examples of *exceptions*
- The represent (not necessarily fatal) errors
- Exception are of a special *type* called `Exception`
- `IOError`, `ValueError`, etc. inherit from `Exception`

Exceptions

- Exceptions (instances) are *thrown*
- Until someone *catches* them
- For that, we use the `try` and `except statements`

```
try:
    f = open('somefile.txt')
except:
    print 'Something bad happened'

# things continue even if an exception took place
```

- The `except:` statement catches *all* exceptions

Exceptions

- We can be more picky about which exception to handle

```
try:
    f = open('somefile.txt')
    v = int(f.readline())

except IOError:
    print 'Something␣bad␣happened␣with␣the␣file'

except ValueError:
    print 'Something␣bad␣happened␣while␣converting␣the␣line'

except:
    print "Something␣bad␣happened␣and␣we␣didn't␣expect␣it"
```

This works similar to the `if...elif...else` construction.

Exceptions

- Don't forget that exceptions are *objects*
- We get more information about the errors by inspecting them

```
try:
    f = open('somefile.txt')
    v = int(f.readline())

except IOError, ioe:
    print 'Error␣with␣file:␣%s' % ioe.filename

except ValueError, ve:
    print 'Error␣converting:␣%s' % ve.args

except Exception, e:
    print "Error:␣%s" % e
```

Exceptions are thrown

- Exception catching can be deferred
- They go up the call stack making every function to return immediately

```
def process_file(fn):
    try:
        f = open(fn)
        v = int(f.readline())
        f.close()
    except ValueError:
        v = -1
    return v

# main entry point
try:
    v = process_file('somefile.txt')
except IOError:
    print 'There is some problem with the file'
    # exit the program using the sys module
    sys.exit()

# do something with v
```

Exceptions are thrown

- You throw an exception using the `raise` statement

```
def get_color_by_name(c):
    if c == 'red':
        return (1,0,0)
    elif c == 'green':
        return (0,1,0)
    elif c == 'blue':
        return (0,0,1)
    else:
        raise KeyError('Color is not known')
```

- Design discussion: this function is not nice, why?

Pure & virtual methods

- *Virtual methods* are methods that can be overridden by a subclass
- Every method is virtual in Python
- *Pure virtual methods* are methods that should be implemented by a subclass. They can be 'simulated' in Python.

```
class Animal(object):
    def __init__(self, n):
        self.name = n

    def talk(self):
        raise NotImplementedError

class Cat(Animal):
    def talk(self):
        print self.name + 'says␣meooww!'

class Dog(Animal):
    def talk(self):
        print self.name + 'says␣woof␣woof!'
```

References

- Chapters 17 and 18 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- Errors and exceptions
  http://docs.python.org/tutorial/errors.html
- Built-in exceptions
  http://docs.python.org/library/exceptions.html
- String formatting
  http://docs.python.org/library/stdtypes.html#string-formatting