CS 11 python track: lecture 4

- Today:
  - More odds and ends
    - assertions
    - "`print >>`" syntax
    - more on argument lists
    - functional programming tools
    - list comprehensions
  - More on exception handling
  - More on object-oriented programming
    - inheritance, multiple inheritance, etc.

Odds and ends (1)

- Assertions

```
# 'i' should be zero here:
assert i == 0
# If fail, exception raised.
```

- "print to" syntax

```
import sys
print >> sys.stderr, "bad!"
```

Note on error messages

- Error messages should always go to `sys.stderr`
- Two ways to do this:

```
import sys
print >> sys.stderr, "bad!"

sys.stderr.write("bad!\n")
```

- Either is fine
- Note that `write()` doesn't add newline at end

Odds and ends (2) – arg lists

- Default arguments, keyword arguments

```
def foo(val=10):
    print val
foo()        # prints 10
foo(20)      # prints 20
foo(val=30)  # prints 30
```

- Default args must be at end of argument list

Odds and ends (3) – arg lists

- Arbitrary number of arguments

```
def foo(x, y, *rest):
    print x, y
    # print tuple of the rest args:
    print rest
>>> foo(1, 2, 3, 4, 5)
1 2
(3, 4, 5)
```

Odds and ends (4) – arg lists

- Keyword args:

```
def foo(x, y, **kw):
    print x, y
    print kw
>>> foo(1, 2, bar=6, baz=7)
1 2
{ 'baz' : 7, 'bar' : 6 }
```

Odds and ends (4) – arg lists

- Arbitrary number of args + keyword args:

```
def foo(x, y, *rest, **kw):
    print x, y
    print rest
    print kw
>>> foo(1, 2, 3, 4, 5, bar=6, baz=7)
1 2
(3, 4, 5)
{ baz : 7, bar : 6 }
```

Functional programming tools (1)

- First-class functions:

```
def foo(x):
    return x * 2
>>> bar = foo
>>> bar(3)
6
```

Functional programming tools (2)

- lambda, map, reduce, filter:

```
>>> map(lambda x: x * 2, [1, 2, 3, 4, 5])
[2, 4, 6, 8, 10]
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
>>> sum([1, 2, 3, 4, 5])  # easier
15
>>> filter(lambda x: x % 2 == 1, range(10))
[1, 3, 5, 7, 9]
```

List comprehensions

```
>>> vec = [2, 4, 6]
>>> [3 * x for x in vec]
[6, 12, 18]
>>> [3 * x for x in vec if x > 3]
[12, 18]
>>> [3 * x for x in vec if x < 2]
[]
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

`try/finally` (1)

- We put code that can raise exceptions into a `try` block
- We catch exceptions inside `except` blocks
- We don't have to catch all exceptions
  - If we don't catch an exception, it will leave the function and go to the function that called that function, until it finds a matching `except` block or reaches the top level
- Sometimes, we need to do something regardless of whether or not an exception gets thrown
  - *e.g.* closing a file that was opened in a `try` block

try/finally (2)

```
try:
    # code goes here...
    if something_bad_happens():
        raise MyException("bad")
finally:
    # executes if MyException was not raised
    # executes and re-raises exception
    #   if MyException was raised
```

try/finally (3)

- Typical example of try/finally use:

```
try:
    myfile = file("foo") # open file "foo"
    if something_bad_happens():
        raise MyException("bad")
finally:
    # Close the file whether or not an
    # exception was thrown.
    myfile.close()
    # If an exception was raised, python
    # will automatically reraise it here.
```

try/finally (4)

- Execution profile:

```
try:
    # ... code (1) ...
finally:
    # ... code (2) ...
# ... code (3) ...
```

- When no exception raised: (1) then (2) then (3)
- When exception raised: (1) then (2) then exit function

try/finally (5)

- This is also legal:

```
try:
    # code that can raise exceptions
except SomeException, e:
    # code to handle exceptions
finally:
    # code to execute whether or not
    # an exception was raised
```

try/finally (6)

```
try:
    # ... code (1) ...
except:
    # ... code (2) ...
finally:
    # ... code (3) ...
# ... code (4) ...
```

- When no exception raised: (1) then (3) then (4)
- When exception raised and caught: (1) then (2) then (3) then (4)
- When exception raised but not caught: (1) then (3) then exit function

Exception classes

- Exception classes, with arguments:

```
class MyException:
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)
try:
    raise MyException(42)
except MyException, e:
    print "bad! value: %d" % e.value
```

More on OOP -- inheritance

- Often want to create a class which is a specialization of a previously-existing class
- Don't want to redefine the entire class from scratch
  - Just want to add a few new methods and fields
- To do this, the new class can inherit from another class; this is called inheritance
- The class being inherited from is called the parent class, base class, or superclass
- The class inheriting is called the child class, derived class, or subclass

Inheritance (2)

- Inheritance:

```
class SubClass(SuperClass):
    <statement-1>
    ...
    <statement-N>
```

- Or:

```
class SubClass(mod.SuperClass):
    # ...
```

  - if `SubClass` is defined in another module

Inheritance (3)

- Name resolution:

```
 foo = Foo() # instance of class Foo
foo.bar()
```

- If `bar` method not in class `Foo`
  - superclass of `Foo` searched
  - etc. until `bar` found or top reached
  - `AttributeError` raised if not found
  - Same thing with fields (`foo.x`)

Inheritance (4)

- Constructors:
  - Calling `__init__` method on subclass doesn't automatically call superclass constructor!
  - Can call superclass constructor explicitly if necessary

Inheritance (5)

```
class Super:
  def __init__(self, x):
    self.x = x

class Sub(Super):
  def __init__(self, y):
    Super.__init__(self, y)
    self.y = y
```

Inheritance example (1)

```
class Animal:
    def __init__(self, weight):
        self.weight = weight
    def eat(self):
        print "I am eating!"
    def __str__(self):
        return "Animal; weight = %d" % \
            self.weight
```

Inheritance example (2)

```
>>> a = Animal(100)
>>> a.eat()
I am eating!
>>> a.weight
100
>>> a.fly()
AttributeError: Animal instance has no attribute 'fly'
```

Inheritance example (3)

```
class Bird(Animal):
    def fly(self):
        print "I am flying!"
b = Bird(100) # Animal's __init__() method
b.eat()
I am eating!
b.fly()
I am flying!
```

Exceptions again

- Can use inheritance to make it easy to generate simple exception subclasses:

```
class MyException(Exception):
    pass
```

- This is the same as the previous `MyException` code, but much simpler to write
- Superclass (`Exception`) already does everything that `MyException` can do, so just inherit that functionality

Multiple inheritance (1)

- Multiple inheritance:

```
class SubClass(Super1, Super2, Super3):
    <statement-1> . .
    <statement-N>
```

- Resolution rule for repeated attributes:
- Left-to-right, depth first search
  - sorta...
  - Actual rules are slightly more complex
  - Don't depend on this if at all possible!

Multiple inheritance (2)

- Detailed rules:
  - `http://www.python.org/2.3/mro.html`
- Usually used with "mixin" classes
  - Combining two completely independent classes
  - Ideally no fields or methods shared
  - Conflicts then do not arise

Mixin example

```
class DNASequence:
    # __init__ etc.
    def getBaseCounts(self): ...
    # other DNA-specific methods
class DBStorable:
    # __init__ etc.
    # methods for storing into database
class StorableDNASequence(DNASequence, \
    DBStorable):
    # Override methods as needed
    # No common fields/methods in superclasses
```

Private fields

- Private fields of objects
  - at least two leading underscores
  - at most one trailing underscore
  - e.g. `__spam`
- `__spam` → `_<classname>__spam`
  - `<classname>` is current class name
- Weak form of privacy protection

Next week

- We'll talk about the new features and changes in Python 3.x (3.0, 3.1, etc.)