The current topic: Python

- ✓ Introduction
- Object-oriented programming: Python
  - ✓ Features, variables, numbers, strings, Booleans, while loops
  - ✓ If statements, sequences, functions, modules
  - – Next up: Dictionaries, command-line arguments, files, classes, inheritance, polymorphism
- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog

Announcements

- Lab 1 has been posted.
  - – Six exercises.
  - – We've already covered the material needed for the first three exercises.
  - – Today we'll also finish covering material needed for the sixth exercise.

Dictionaries

- A *dictionary* is a container but it is not a sequence. It is a mutable set of key-value pairs.

```
d = { 'Newton': 1642, 'Darwin': 1909 }
d['Newton']   # 1642
d['Darwin']  = 1809
d['Turing']              # KeyError

if 'Turing' in d:        # False
    print d['Turing']
else:
    d['Turing'] = 1912

# Getting a list of keys:
d.keys()   # ['Darwin', 'Turing', 'Newton']
```

- Similar to maps in Java and C++.

Dictionaries: Using a default value

- Example:

```
bdays = ['May', 'Jun', 'Jun', 'Apr', 'May', 'Jun']
freq = {}
for v in bdays:
    if v in freq:
        freq[v] += 1
    else:
        freq[v] = 1
print freq   # {'Apr': 1, 'Jun': 3, 'May': 2}

# We can replace the if/else with:
freq[v] = freq.get(v, 0) + 1
```

- Observe that `freq.get(v, 0)` returns `freq[v]` if this exists, and 0 otherwise.

Dictionaries: Storing properties

- Python uses dictionaries to store properties.
- Modules have an attribute `__dict__` that is a dictionary listing the items (functions, classes, variables) in the module.

```
import sys
sys.__dict__.keys()  # A list of items in sys.
```


Python's built-in help

- Use the `help()` function to learn more about a module, function, class, or method. Some examples:

```
help(range)      # the range() function

import sys
help(sys)        # the sys module

L = [1, 2, 3]
help(L)          # list objects
help(list)       # same as help(L) since L is a list
help(L.append)   # the append method

#watch out:
help(L.append())   # TypeError, argument missing
```


Command-line arguments

- `sys.argv` is a list with all the command-line arguments.
  - The first command-line argument, `sys.argv[0]`, is the name of the program.
  - Standard list operations can be used. e.g. `len(sys.argv)` tells you how many arguments there are.
  - Suppose file `testargs.py` is as follows:

```
#!/u/prof/ajuma/share/bin/python2.5
import sys
print sys.argv
```

  - Running `testargs.py`:

```
$ python2.5 testargs.py first 2 3three
['testargs.py', 'first', '2', '3three']

$ ./testargs.py 321 123 abc def
['./testargs.py', '321', '123', 'abc', 'def']
```


Working with Files

- Open a file with the built-in function `open()`.
  - Specify a mode: 'r' means read, 'w' means write, 'a' means append.

```
inputFile1 = open('filename.txt', 'r')
outputFile1 = open('someOtherFile.txt', 'w')
outputFile2 = open('someExistingFile.txt', 'a')
```

- Reading from an open file:
  - The `readline()` method can be used to read an open file line-by-line. It returns the empty string '' when end-of-file is reached.

```
nextLine = inputFile1.readline()
while nextLine:   # works since empty string is False
    print "The next line is:", nextLine,
    nextLine = inputFile1.readline()
```

  - Another approach: iterate through the file just as you would through a list.

```
for nextLine in inputFile1
    print "The next line is:", nextLine,
```

Working with Files

- When you read lines from a file, you'll probably want to remove the newline character from the end of each line.
  - Use the `strip()` method to remove this, and to also removing any other leading or trailing whitespace.

```
mystr = '    abcd     \n     '
cleanstr = mystr.strip()
cleanstr       # 'abcd'
```

- Writing to an open file:
  - Use the `write()` method. Note that newlines are not automatically added in, so you need to include them yourself if you want them.

```
outputFile1.write("csc326")
outputFile1.write(" same line\n")
outputFile1.write("start of a new line")
```

- Use the `close()` method to close an open file.

```
outputFile1.close()
```


Object-oriented concepts: a review

- Some definitions you've seen before:
  - *Class*: A plan for the creation of objects.
  - *Object*: A structure built according to the plan in a class, with associated operations.
  - *Inheritance*: Designing a child class by extending or specializing the plan of a parent class.
  - *Overriding*: A child class replaces a method defined in a parent class with its own definition.
  - *Polymorphism*: An action carried out on an object that may belong to any of several classes (in a hierarchy) is specialized to the actual class of the actual object at runtime.

Object-oriented concepts: a review

- More definitions you've seen before:
  - *Instance*: An object belonging to a class. Literally, an "example" (as in the phrase "for instance"), because any object is an example of all the objects that may be created using its class.
  - *Instance variable*: A variable belonging to an object. Every object has its own separate copy of an instance variable.
  - *Class variable*: A variable belonging to a class – that is, shared among all the instances of a class. These are also known as *static* variables in C++ and Java.
    - As we'll see, "class" and "static" aren't equivalent terms in Python.
- Reference: Sebesta, Chapter 12.

Classes in Python

- Python has two kinds of classes, known as *new-style* classes and *classic* classes.
  - New-style classes were introduced in Python 2.2.
  - Some differences:
    - Each new-style class defines a new type. On the other hand, all instances of classic classes are of type "instance".
    - New-style classes can use Python's built-in types (lists, tuples, etc.) as base classes.
    - Classic classes aren't required to have parent classes. All new-style classes must be descendants of the `object` class (this is similar to Java).
    - New-style classes handle multiple-inheritance more intelligently.
    - New-style classes can include *static* methods and *class* methods.
- We'll only use new-style classes.
- Reference: http://www.python.org/download/releases/2.2.3/descrintro/
  - A write-up by Guido van Rossum, describing the differences and explaining some of his design decisions.

A simple class

- A class `C` whose parent is `object`:

```
class C(object):
    x = 7  # class variable, not instance variable!
    y = 10 # class variable

    def m(self):
        self.y = 20  # assigns to *instance* variable

a = C()  # creates an instance of C
b = C()
a.x     # 7
a.y     # 10 (class variable)
a.m()
a.y     # 20 (instance variable)
b.y     # 10
C.y = 75
b.y     # 75
a.y     # 20 (instance variable hides class variable)
```


A simple class

- Observations about class `C`:
  - `class C(object):`
    This says that object is the parent class. If we leave out the parent class altogether, we get a classic class, not a new-style class.
  - `def m(self):`
    The `self` parameter must be the first parameter of every instance method. When `m` is called (`a.m()` in our example), `self` is automatically set to the instance on which `m()` is being called. This is like the word `this` in C++/Java. Note that technically you can use any word you like in place of `self`, but it's conventional to use `self`.
  - `self.y = 20`
    Unlike C++/Java, you have to use `self` in order to access instance variables. Any **assignments** to `self.<variable>` (or, outside of a class, to `<objectname>.<variable>`) are always to instance variables and not to class variables. As usual in Python, instance variables are created whenever they are first assigned to.

A simple class

- More observations about class `C`:
  - `a = C()`
    Creating an object looks like a "call" to the class. Unlike Java, there is no `new` operator to create objects.
  -
```
a.y     # 10 (class variable)
a.m()
a.y     # 20 (instance variable)
```
    When resolving a **reference** (as opposed to an assignment) to `<objectname>.<variable>` (or, within a class, to `self.<variable>`), Python first looks for an instance variable named `<variable>`, and if none is found, looks for a class variable (and then looks at class variables for parent classes, and so on up the inheritance tree).

A class with a constructor

- A class `D` with a constructor that takes one parameter, `x`:

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

    def m(self):
        return self.x + 2

a = D(5)
a.m()     # 7
a.x       # 5
```

- The constructor must be called `__init__()`. If you don't provide a constructor, one is inherited from `object` (this doesn't do much).
  - Note that names that begin and end with two underscores are special methods or values.

Inheritance

- Parent and Child classes:

```
class Parent(object):
    x = 5
    def n(self):
        return self.x + 2

class Child(Parent):
    x = 8
    def m(self):
        return self.x + 4 + self.n()
           # self.n() is equivalent to Parent.n(self)

p = Parent()
c = Child()
p.n()   # 7
c.m()   # 22 (and not 19)
c.n()   # 10 (and not 7)
```


Inheritance

- Methods and variables are inherited, and can be overridden.
- Parent and Child classes, where Child overrides a method:

```
class Parent(object):
    x = 5
    def m(self):
        return self.x + 2

class Child(Parent):
    x = 8
    def m(self):
        return self.x + 4 + Parent.m(self)

p = Parent()
c = Child()
p.m()   # 7
c.m()   # 22 (and not 19)
```


Instance methods and variables

- To call an instance method `m()` from *within* a class, we have to call `self.m()`.
- To call an instance method `m()` of a parent class P from within a child class that has overridden `m()`, we call `P.m(self)`.
- All instance methods and variables are public by default.
  - Later we'll see that we can get around this, to a certain extent.
- There is only one instance variable of a given name for each object. So if a parent class `P` and a child class `C` both assign to `self.x`, they are assigning to the same variable.

Polymorphism

- *Polymorphism* is the property of "having many shapes". In object-oriented programming, it means taking on the attributes of the actual object instead of the apparent object.

```
class P(object):
    def talk(self):
        return 'Hi: ' + self.msg()
    def msg(self):
        return 'Parent'

class C(P):
    def msg(self):
        return 'Child'

c = C()
c.talk()    # 'Hi: Child'
d = P()
d.talk()    # 'Hi: Parent'
```

Polymorphism

- Observations about `c.talk()`:
  - The call `self.msg()` in P's `talk()` gets the `msg()` in C, because the actual object is a C.
    - Well, the actual object is also a P, since every C is a P, but the "lowest" version of `msg()` is the one that gets used.
  - On the other hand, when we call `d.talk()`, the actual object is an instance of P and not of C.

Polymorphism in Java/C++

- In Java and C++, polymorphism is more visible than in Python, because in those languages every variable has a type.
  - For example, in Java:

```
P o = new C();
o.talk();
```

    At compile time, the compiler thinks the `talk()` call is to P's `talk()`, but at runtime C's `talk()` gets called.

  - Polymorphism is more complicated in C++.
    - Need to declare methods as `virtual` to get polymorphic behaviour.
    - In C++ there are both stack-allocated and heap-allocated objects. Stack allocation is for directly declared variables. Heap allocation is for is objects constructed at runtime by the `new` operator. (In Java, all objects are heap-allocated.)

The object construction process

- Every class has a *static* method called __new__.
  - We'll discuss static methods later on.
  - `o = C()` calls `C.__new__(C)`, which returns a new instance of C.
    - If additional arguments are given to `C()`, these are also passed on to `__new__`.
  - Next, `C.__init__()` is called to initialize the new instance.
    - Again, any additional arguments are passed on to `__init__`.
    - If C doesn't have an `__init__()` method, the parent class' `__init__()` is called. Ultimately, this can result in `object`'s `__init__()` being called, and this doesn't do much.
  - If C has a parent class P, and C has an `__init__()` method, then P's `__init__()` method is **not** called unless C's `__init__()` calls it explicitly:

```
class C(P):
    def __init__(self):
        P.__init__(self)
        # additional stuff specific to C would go here.
```


Exercises

- Write classes A, B, and C, where C's parent is B, and B's parent is A. A's parent is `object`. Experiment with inheritance and polymorphism.
- Continuing on with our Fibonacci number example:
  - Recall that the first two numbers in the Fibonacci sequence are both 1. But it makes sense to consider a sequence defined the same way where the first two numbers are something else. (e.g. 2, 7, 9, 16, 25, 41,..., is such a sequence where we've chosen 2 and 7 as the first two numbers.)
  - So write a class `NewFibonacci` whose constructor takes two numbers; the class uses these two numbers as the first two numbers in the sequence.
  - The class should have a method `calculate(n)` that returns the n-th number in the sequence.
  - Also add a method `next()`. The first call to `next()` returns the first number in the sequence, the second call returns the second number, and so on. You'll obviously need instance variables to save state between calls.
  - Finally, add a method `writeToFile(n, filename)`, that writes the first n numbers in the sequence to the file named `filename`, one number per line.