The current topic: Python

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

Announcements

- Lab 1 is due September 29th at **10:30 am**.
  - Submit each file using the `submitcsc326f` command, with the first argument (the assignment number) set to 1. For example, to submit `ex3.py`, use:

    ```
    submitcsc326f 1 ex3.py
    ```

  - After submitting, use the command

    ```
    submitcsc326f -l 1
    ```

    (the character after the dash is a lower case L) to see a list of the files you submitted, the size of each file, and the time each file was submitted. Your list of files should include `ex1.py`, `ex2.py`, `ex3.py`, `ex4.py`, `ex5.py`, `ex6.py`, and `MyList.py`.

Multiple Inheritance

- Like C++ but unlike Java, Python allows *multiple inheritance*.
  - This means that a class can have multiple parent classes.

    ```
    class A(object): ...
    class B(object): ...
    class C(A, B): ...
    ```

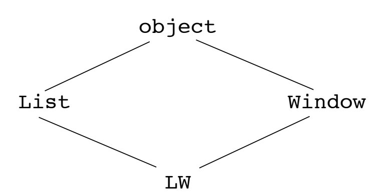
  - Issues to consider:
    - Suppose A and B each define a method `m()`, and C does not define such a method. Which `m()` gets called in the following situation?

      ```
      c = C()
      c.m()
      ```

    - Things get even more interesting with diamond-shaped inheritance. In the current example, `object` is an ancestor of C two different ways (through A and through B).
    - How do we make sure that each ancestor class' constructor gets called exactly once?

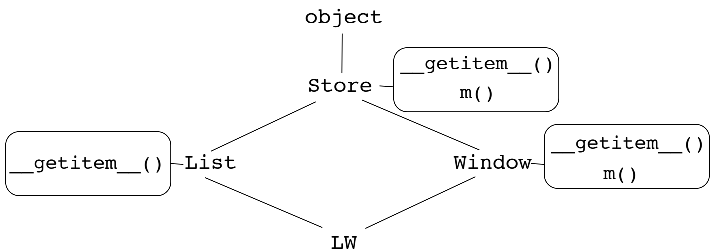
Multiple Inheritance

- An example:
  - Suppose we want an object that can store a bunch of data items *and* draw a picture on the screen.
  - Suppose we have a List class that can store a bunch of data items.
  - Suppose we have a Window class that can draw a picture on the screen.
  - Then we can define an LW class that has List and Window as parents.



Multiple Inheritance

- To make things more interesting, suppose List and Window are both children of Store.



- Suppose LW does **not** define its own `__getitem__()` or `m()`. Which `__getitem__()` does it inherit? Which `m()` does it inherit?

- Answer: Python defines a *method resolution order*. When looking for a method, it checks classes in the order specified by the method resolution order.

Multiple Inheritance

- The rules defining the method resolution order are complicated. One general idea is that the methods of a child class have priority over those of its parent. For example, methods in List and Window have priority over those in Store.
  - Details about the rules: http://www.python.org/download/releases/2.3/mro/

- To find out the method resolution order of a class, check its `__mro__` attribute.

```
class Store(object): pass
class List(Store): pass
class Window(Store): pass
class LW(List, Window): pass

LW.__mro__  # (LW, List, Window, Store)
```

- So LW inherits List's `__getitem__()` and Window's `m()`.

Multiple Inheritance

- Another issue: suppose Store, List, and Window each define an `__init__()` method. When writing LW's `__init__()` method, how can we make sure each of its ancestor's `__init__()` methods is called?
  - One solution: Each class' `__init__()` should call the `__init__()` of each of its parents.
  - Problem: If we did this, Store's `__init__()` will get called twice when we're constructing an LW object (once as a result of calling List's `__init__()` and once as a result of calling Window's `__init__()`).
  - Better solution: Call the ancestor's `__init__()` methods in the order specified by the method resolution order.

Multiple Inheritance

- The function `super()` can be used to determine what comes next in an object's method resolution order.
  - `super(C, o)` is the same object as `o` but looks for methods starting after class C in o's method resolution order.
  - For example, if object o is an instance of LW, then `super(List, o).__getitem__()` calls Window's `__getitem__()` method since Window follows List in `LW.__mro__`.

- To make sure each ancestor's `__init__()` gets called exactly once, add the line `super(C, self).__init__()` to the `__init__()` method of each class C.
  - Note that the C inside the `super()` call should match the name of the class within which this call is being made.
  - For now, we're glossing over the issue of passing arguments to `__init__()`. One solution is to use keyword parameters, which we haven't covered yet.

Multiple Inheritance

- Example:

```
class Store(object):
   def __init__(self):
      super(Store, self).__init__()
      # other stuff goes here

class List(Store):
   def __init__(self):
      super(List, self).__init__()

class Window(Store):
   def __init__(self):
      super(Window, self).__init__()

class LW(List, Window):
   def __init__(self):
      super(LW, self).__init__()
```

- This ensures that when an LW instance is constructed, the `__init__()` methods are called in the order LW, List, Window, Store.

Parameters and arguments

- *Parameters* appear in the definition of a function (or method).
  - They are a "placeholder" for the actual values passed in a call.
- *Arguments* (or *actual parameters*) appear in a function call and must correspond to the parameters in the function definition.
- Python passes parameters by copying their value, just like C/Java.
  - But Python variables always store references to objects.
  - "Copying" the parameter just copies the reference it stores.
  - This has the effect of passing the original object, not a copy of the object.

Parameters

- In a Python function definition, the parameter list has four parts:
  - Mandatory parameters.
    - This is all we've seen so far.
  - Optional parameters.
  - Extra non-keyword parameters specified as a tuple `*t`.
  - Extra keyword parameters specified as dictionary `**d`.
- Any of these parts may be omitted in a function definition, but the parts that do appear must appear in the order given above.

Arguments

- In a Python function call, the argument list has four parts:
  - Non-keyword arguments.
    - This is all we've seen so far.
  - Keyword arguments.
  - Non-keyword arguments given as single tuple `*t`.
  - Keyword arguments given as a single dictionary `**d`.
- The parts that appear in a function call must appear in the order given above. The function definition determines which of the above parts are required, optional, and not allowed.

Optional parameters

- To make a parameter optional, we have to provide a default value.

```
def incr(x, y=0):
   y += x
   return y

incr(4)     # 4
incr(6, 5)  # 11
incr(6)     # 6
```

- Another example:

```
def f(x, y=[]):
   y.append(x)
   return y

f(23)    # [23]
f(45)    # [23, 45]. Only one copy of the default value!
f(1)     # [23, 45, 1]
```


Optional parameters

- The default value of an optional parameter becomes an attribute of the function.
  - Default values are stored in an attribute called `func_defaults`.
    - This is a tuple that store default values in the order that they appear in the function declaration.
  - If the default value is a mutable object, and the function modifies this object, then future calls to the function get the modified object, not the original.
  - To keep the default value the same for every call, create a new object each time:

```
def f(x, y=None):
   if y == None:
      y = []   # A new object.
   y.append(x)
   return y

f(23)   # [23]
f(45)   # [45]
f(1)    # [1]
f.func_defaults   # (None,)
```


Keyword arguments

- What happens when there are multiple optional parameters?

```
def g(x, y=3, z=10):
   print 'x:', x, 'y:', y, 'z:', z
```

How do we call `g` if we want to specify a value for `z` but use the default for `y`?

```
g(1, , 3)   # SyntaxError
```

- Solution: Keyword arguments.

```
g(1, z=2)    # 'x: 1 y: 3 z: 2'
g(z=1, x=4)  # 'x: 4 y: 3 z: 1'
g(y=1, 2)    # SyntaxError: non-keyword arg after keyword arg
g(7, y=1, x=0)  # TypeError: multiple values for x
g(4,6,z=1)   # 'x: 4 y: 6 z: 1'
g(z=2, x=0, y=22)  # 'x: 0 y: 22 z: 2'
g(z=2)      # TypeError: no value given for x
```


Keyword arguments

- Any parameter, whether it's optional or mandatory, can be provided using a keyword argument in a function call.

- Keyword arguments can appear in any order, as long as all keyword arguments appear *after* all non-keyword arguments.

- When a call includes a mix of non-keyword and keyword arguments:
  - Python matches up the non-keyword arguments with parameters *by position* (the approach that you're used to seeing).
  - Then, Python matches up the keyword arguments with parameters *by name*.
  - If any mandatory parameter isn't given a value by this process, a TypeError occurs.
  - If any mandatory or optional parameter is given more than one value by this process, a TypeError occurs.

Extra non-keyword parameters

- A function can defined to take an arbitrary number of non-keyword parameters.
  - Include a parameter name preceded by a star when defining the function.
  - When the function is called, it is given the extra non-keyword arguments as a tuple.

```
def f(x, y=4, *z):
   print z

f(1)      # ()
f(1,3)    # ()
f(1,2,3)  # (3,)
f(1,2,3,4,5,6)   # (3, 4, 5, 6)
f(1,3,w=4)  # TypeError. w is an extra *keyword* argument
f(1,3,z=4)  # TypeError. z can't be given as a keyword arg.
f(1,3,y=10) # TypeError: multiple values for y
```

  - Observe that we can't call f with extra non-keyword arguments without first giving a value for y (rather than relying on the default value for y).

Extra keyword parameters

- A function can defined to take an arbitrary number of keyword parameters.
  - Include a parameter name preceded by a two stars when defining the function.
  - When the function is called, it is given the extra keyword arguments as a dictionary.

```
def g(x, y=4, *z, **d):
   print z, d

g(10,20)                # () {}
g(1,2,a=4,csc='326')  # () {'a': 4, 'csc': '326'}
g(b=4,c='jkl',x=1)    # () {'c': 'jkl', 'b': 4}
g(1,2,3,4,w='abc')    # (3,4) {'w': 'abc'}
```


A tuple of non-keyword arguments

- In a function call, a sequence of non-keyword arguments can be given as a single tuple by preceding the tuple with a star.

```
def f(x, y=4, *z):
   print y, z

t = (1,2,3)
f(*t)            # 2 (3,)
f(0, *t)         # 1 (2,3)
f(9, 0, *t)      # 0 (1,2,3)
f(9, 8, 0, *t)   # 8 (0,1,2,3)
f(y=2, *t)       # TypeError. Non-keyword arguments are matched
                 # before keyword arguments, so y got 2 values.
f(*(1, 2))       # 2 ()
```

- Note that even though the tuple of non-keyword arguments appears after keyword arguments in the function call, **all** non-keyword arguments are matched with parameters (by position) *before* keyword arguments are matched.

A dictionary of keyword arguments

- In a function call, a collection of keyword arguments can be given as a single dictionary by preceding the dictionary with two stars.

```
def g(x, y=4, **z):
   print y, z

d={'x':'3', 'a':1, 'b':'cde'}
g(**d)             # 4 {'a': 1, 'b': 'cde'}
```

- Observe that the call `g(**d)` is equivalent to the call `g(x='3',a=1,b='cde')`.

- More calls:

```
g(w=9,**d)         # 4 {'a': 1, 'b': 'cde', 'w': 9}
g(1,**d)           # TypeError: multiple values for x
g(y=0,**d)         # 0 {'a': 1, 'b': 'cde'}
g(1,**{'c':3, 'd':2})     # 4 {'c': 3, 'd': 2}
```

List comprehensions

- Idea comes from set notation.
- In math, if we have a set S, say S = {1, 2, 3, 4}, we can define a new set by writing:
  $T = \{2x \mid x \in S\}$
  Then T = {2, 4, 6, 8}.

- List comprehensions allow us to apply the same idea to construct lists.
  - And we don't need to start with a list – we can start with anything that can be iterated.

- A simple example:

```
S = [1, 2, 3, 4]
T = [2*x for x in S]
T   # [2, 4, 6, 8]
```


List comprehensions

- We can also specify conditions on the iteration.
  - Only objects satisfying the condition are used to construct the new list.

```
old = [90, 50, 15, 20, 40, 75]
new = [x+1 for x in old if x >= 40]
new    # [91, 51, 41, 76]
```

- Everything we can do with list comprehensions we can also do with for loops.
  - But list comprehensions are more compact (less typing).
  - List comprehensions actually execute faster than equivalent for loops.

List comprehensions

- List comprehensions can include nested iterations (like nested for loops):

```
A = [4, 8, 16, 32]
B = [0, 1, 2]

C = [x/y for x in A for y in B if y > 0]

C   # [4, 2, 8, 4, 16, 8, 32, 16]
```

- Note the order in which objects are processed – the iteration specified first (`x in A`) is treated as the "outer" loop and the iteration specified next (`y in B`) is treated as the "inner" loop.

List comprehensions

- List comprehensions can be useful for working with matrices (represented as nested lists).
  - Create a 3x3 matrix of 1s:

```
M = [[1 for i in range(3)] for j in range(3)]
M   # [[1,1,1], [1,1,1], [1,1,1]]
```

  - Multiply a matrix by a constant:

```
L = [[1,2,3], [4,5,6], [7,8,9]]
L = [[2*i for i in row] for row in L]
L   # [[2,4,6], [8,10,12], [14,16,18]]
```

  - Another way to do this:

```
L = [[1,2,3], [4,5,6], [7,8,9]]
L = [[2*L[i][j] for j in range(3)] for i in range(3)]
L   # [[2,4,6], [8,10,12], [14,16,18]]
```

List comprehensions

- Using objects other than lists:
  - The only requirement is that the object can be iterated.
  - Using a file:

```
# List of every line that starts with 'A' in file 'in.txt'.
L = [line for line in open('in.txt', 'r') if line[0] == 'A']
```

  - Using the `MyList` class from Lab 1:

```
m = MyList()
m.append('a')
m.append(22)
m.append(0.4)
L = [3*i for i in m]
L    # ['aaa', 66, 1.2]
```


Exercises

- Write a function that can be called with an arbitrary number of non-keyword arguments and returns the first argument that is divisible by 3. (Recall that we can use the % ("mod") operator to test divisibility.)
- Write a function that can be called with an arbitrary number of keyword arguments and returns the sum of all keyword arguments whose name is at least 2 characters long.
- Write a function that takes two matrices (of the same size) and returns their sum. The function should use list comprehensions to compute the sum.