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
  - – Next up: Exceptions, operator overloading, privacy
- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog

Announcements

- By the end of today's lecture, we will have covered all the material needed for Lab 1.
  - – Don't forget to look at the bulletin board for clarifications.
  - – Make sure you follow the specified input and output format for each question.
- Reminder: Office hours are Monday 1:30-2:30 and Wednesday 11-12.

Instance methods

- Instance methods can only be called on instances of their class (or its descendants)!

```
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)

class Unrelated(object):
    x = 8

c = Child()
Parent.m(c)    # 10
u = Unrelated()
Parent.m(u)    # TypeError (u isn't a Parent)
```


Exceptions

- A way to deal with errors.
  - – Recover from them if you can.
  - – Otherwise, stop execution.
- Idea: When something "bad" happens, an exception is raised. At this point, the program stops whatever it's doing and then:
  - – Goes to the highest level in the call stack (that is, the most recently called method or function) that can deal with the exception.
  - – If no level is able to deal with the exception, go back to the system that called the program (which results in the program halting with an error message).
- To "catch" (deal with) an exception, code that may raise the exception is enclosed in a "try...except" block where the "except" portion is executed when the exception occurs.
- Reference: Sebesta, Chapter 14.

Exceptions

Call stack

`m3.f3()` raises `E`
`m2.f2()` stmt C calls `m3.f3()`
`m1.f1()` stmt B calls `m2.f2()`
`main` stmt A calls `m1.f1()`

- Suppose function `f3()` in module `m3` raises an exception `E` as in the given call stack.
- If statement C is in a try...except block, this block will catch exception `E`.
- Otherwise, exception `E` is passed to the next level down in the call stack. So if statement B is in a try...except block, this block will catch exception `E`.
- Otherwise... (the same idea again). So if statement A is in a try...except block, this block will catch exception `E`.
- Otherwise (no one has caught the exception so far), the program stops and an error message is given to the user.

Try... Except

- In Python, built-in exceptions are descendants of the class `Exception`. An except clause catches all descendants of the exception it specifies.

```
try:
   L=[]
   L[4]   # IndexError
except Exception:
   print "Error caught"
print "continuing..."
```

Output:

```
Error caught
continuing...
```


Try... Except

- A Try...Except block can include multiple except clauses. The **first** one that catches a particular exception is the **only** one that gets executed. If none of the except clauses catches a particular exception, it gets passed back to the previous level of the call stack.

```
try:
   4 / 0    # This causes a ZeroDivisionError
except IndexError:
   print "catching IndexError"
except ArithmeticError:
   print "catching ArithmeticError"
except ZeroDivisionError:
   print "catching ZeroDivisionError"
```

Output:

```
catching ArithmeticError
```

- "`except ArithmeticError`" catches a `ZeroDivisionError` since `ArithmeticError` is the parent of `ZeroDivisionError`.

Try... Except... Else

- A Try...Except block can include an else clause that only gets executed when **no** exceptions are raised. That is, the else clause handles the non-exceptional case.

```
try:
   #do some stuff that might cause an exception

except (IndexError, TypeError):  # catch multiple exceptions
   #handle exceptions

else:
   # yay! Here we assume that the try clause succeeded without
   # raising any exceptions.
```

Try... Finally

- A Try block can include a finally clause that gets executed whether or not an exception is raised. This is a good place to do any cleanup (e.g. closing open files) that needs to occur regardless of whether there was an exception.

```
try:
   outFile = open("someFile", "w")
   #do some other stuff that might cause an exception

finally:
   outFile.close()
```

- The finally clause is run either:
  - After the try clause successfully executes.
  - When an exception is raised in the try clause; in this case, the finally clause is executed, *and then* the exception is passed on to enclosing/calling code.

Try... Except... Else... Finally

- Python 2.5 allows you to have except and finally clauses in the same block.

```
try:
   L[0]   # NameError (there is no variable L)
except NameError:
   print 'catching NameError'
except IndexError:
   print 'catching IndexError'
else:
   print 'no errors'
finally:
   print 'done'
print 'continuing execution'
```

Output:

```
catching NameError
done
continuing execution
```


Raising exceptions

- To raise an exception, use a `raise` statement.

```
try:
   raise Exception
except Exception:
   print "Exception raised"
```

Output:

```
Exception raised
```


User-defined exceptions

- User-defined exceptions aren't (yet) strictly required to be descendants of `Exception`, but this is a good practice to follow.

```
class SimpleE(Exception):
   pass  # pass tells Python "we have nothing to say here"

try:
   raise SimpleE  # treated as "raise SimpleE()"
except SimpleE:
   print 'caught SimpleE'
```

Output:

```
caught SimpleE
```

User-defined exceptions

- Exceptions can take arguments (that might give more information about what caused them to be raised). These arguments are passed to the exception's constructor.

```
class BetterE(Exception):
   def __init__(self, value):
      self.value = value
   def __str__(self):
      return self.value

try:
   raise BetterE("abcde")  # leaving out the argument will
                           # cause another exception!
except BetterE, e:    # e is the instance that was raised
   print e            # print uses e.__str__()
```

Output:

```
abcde
```

- Note that the catching code can get the actual exception instance.

Operator overloading

- *Operator overloading*: Operators are given multiple definitions, and the right one is chosen based on context.
  - e.g. The + operator means different things for ints and strings.

```
4 + 4        # for ints, + means addition
"ab" + "cd"  # for strings, + means concatenation
```

  - You can even give the + operator a new meaning for your own class. Just define an `__add__()` instance method in your class. Then, if `x` is an instance of your class,

```
x + y
```

    is treated as a call to

```
x.__add__(y)
```

- Reference: Sebesta, Sections 7.3 and 9.10.

Operator overloading

- Some other operators you can overload:
  - Define `__sub__()`, `__mul__()`, and `__div__()` instance methods to overload the – operator, the * operator, and the / operator.
    - `x – y` is treated as a call to `x.__sub__(y)`.
    - `x * y` is treated as a call to `x.__mul__(y)`.
    - `x / y` is treated as a call to `x.__div__(y)`.
  - Define a `__getitem__()` instance method to overload the indexing operator `[]`.
    - `x[i]` is treated as a call to `x.__getitem__(i)`.
    - `__getitem__()` should raise an `IndexError` when given an invalid index.
  - Define a `__len__()` instance method to overload the `len()` operator.
    - `len(x)` is treated as a call to `x.__len__()`.

Operator overloading

- Example: A class that acts like a list of powers of 2.

```
class PowersOfTwo(object):
   def __init__(self, n):
      self.size = n

   def __len__(self):
      return self.size

   def __getitem__(self, i):
      if i < self.size:
         return 2 ** i
      else:
         raise IndexError

c = PowersOfTwo(5)
len(c)   # 5
c[4]     # 16
c[2]     # 4
c[5]     # IndexError
```

Operator overloading

- Note that `__getitem__()` raises an `IndexError` when given an invalid index.
  - This allows us to iterate through a PowersOfTwo object.
  - When iterating through a PowersOfTwo object `c`, Python keeps indexing `c` until an `IndexError` is raised. That is, Python gets `c[0]`, `c[1]`, `c[2]`, etc., until there's an `IndexError`.

```
c = PowersOfTwo(6)
for x in c:
   print x
```

Output:

```
1
2
4
8
16
32
```


Private methods and variables

- Python does not have any way of *enforcing* privacy.
- Instead, *name mangling* can be used to indicate that a particular variable or method should be *treated* as if it were private.
  - Then, anyone who ignores this indication and writes code that accesses "private" variables or calls "private" methods does so at their own risk.
  - As we'll see, this also prevents naming conflicts between "private" methods/ variables of a parent class and "private" methods/variables of a child class.
- Name mangling: Whenever the name of a variable or method within a class begins with __ (two underscores), Python adds on `_<NameOfClass>` to the beginning.

```
class A(object):
   def __m(self):
      return 0
a = A()
a._A__m()    # 0
a.__m()      # AttributeError
```


Name mangling

- Another example, this time with inheritance:

```
class A(object):
   def __m(self):
      self.__y = 20
      self.x = 5
      return 1
   def callM(self)
      return self.__m()
class B(A):
   def __m(self):
      self.__y = 15
      self.x = 10
      return 2

c = B()
c._A__m()    # 1
c._B__m()    # 2
c.x          # 10 (instance variable x is shared by A and B)
c._A__y      # 20
c._B__y      # 15 (instance variable __y is not shared)
c.callM()    # 1  (A's __m() gets called)
```


Name mangling

- Observe that name mangling prevents naming conflicts: You can use a name `__<name>` without having to first check if an ancestor uses it too.
- The method `__m()` defined in B does not override the method `__m()` defined in A, and does not change the behaviour of A's `callM()` method.
  - The call to `__m()` within `callM()` is treated as a call to `_A__m()`.
- Similarly, the assignment to instance variable `__y` by B does not have any effect on the instance variable `__y` used by A.
  - On the other hand, the assignment to instance variable `x` by B is **to the same** instance variable `x` used by A.

Bound and unbound methods

- Consider the following class:

```
class C(object):
   def m(self):
      print 'm in C'
```

- `C.m` is an *unbound* method. That is, it is not bound to an instance of C. This means we need to specify an instance when calling it:

```
C.m()    # Error (need to provide an instance)
x = C()
C.m(x)   # Outputs 'm in C'
x.m()    # Outputs 'm in C'
```

- On the other hand, in the above example, `x.m` is a *bound* method – it is bound to instance `x`.

- What if we want to define methods that we can call without an instance? Such methods are called *class* methods or *static* methods in C++/Java.

Static and class methods

- In Python, *static* methods are **not** the same as *class* methods.
  - Both can be called without being bound to an instance.
  - The difference is that a class method gets the class on which it's called as a parameter.
  - Instance, class, and static methods:
    - An instance method gets the instance on which its called as a parameter `self`.
    - A class method gets the class on which its called as a parameter `cls`.
    - A static method gets neither.
  - The functions `staticmethod()` and `classmethod()` are used to identify static and class methods in a class.

Static and class methods

- An example:

```
class C(object):
   def sm():            # No 'self' parameter!
      print 'static'
   sm = staticmethod(sm)

   def cm(cls):         # 'cls' instead of 'self'
      print 'called on class', cls.__name__
   cm = classmethod(cm)

class D(C): pass

x = C()
C.sm()   # 'static'
x.sm()   # 'static'
C.cm()   # 'called on class C'
x.cm()   # 'called on class C'
y = D()
D.cm()   # 'called on class D'
y.cm()   # 'called on class D'
```


Exercises

- Continuing with the `NewFibonacci` class from the last set of exercises:
  - Overload the indexing operator `[]`. Specifically, define a `__getitem__(i)` method that returns the i-th number in the sequence. This method should raise an `IndexError` if i is negative.
  - Modify the constructor so that it raises an exception when either number it is given is negative. Define a new exception called `FibonacciError` for this purpose.
  - Use name mangling to make all instance variables "private".