Python Programming

Lecture 4

2/22/2009

Today

- Modules
  - Usage and Design
  - Programming With Modules
    - `__builtins__` `sys` and `readline` **(again!)**
- Scoping, Nesting

Modules

- Groupings of code, classes, objects, and functions
    - In only ***one*** file or package/directory of files
- We **`import`** modules into our current context/interpreter
- Each .py file is considered a module
- We use modules for organization and extendability

The `import` Procedure

- Usually at the top of the file, but not necessarily
  - `import module1, module2, ...`
- Kinda dumps another program into this one
- Access module's contents through dot operator
  - `module1.function`, `module2.class`, etc.
- Looks in local directory, and in module path
  - `/usr/lib/python2.6/site-packages`
  - Complete path stored in `sys.path` in a dictionary

`import` is like an assignment

- Creating a new variable in the context
    - `import mymod`
  - Creates a variable **`mymod`** that references that module object, the compiled mymod.py
- What if you don't want it named **`mymod`**?
    - `import mymod as cooler_then_mymod`
- What does each of these do?
    - `from mymod import func1,class2`

    -
      ```
      from mymod import func1 as coolio,\
                         class2 as coolid
      ```
    - `from mymod import *`

More on `import`

- The module is *run/compiled*
  - Parts of the module is executed
- How do you organize your code to stop accidental execution?
- **`__name__`** convention
  - **`if __name__ == "__main__":`**
    then you are the current executing module
  - Otherwise you are the imported module

reload()

- Reload a module, i.e. re-execute
  - Given existing module object
- Overwrites existing name space
  - Local, and top level
- Almost never needed.
- What kind of problems can occur?

Module Design

```
#!/usr/bin/python
# mymod.py

#this will run on an import
print "I run all the time"

def add2(x): return x+2

def minus2(x):return x-2

def main():
   x = 10
   print "x=10"
   print"x=",
   print minus2(add2(x))

if __name__ == "__main__":
  main()
```

```
#!/usr/bin/python
# myothermod.py
import mymod

def main():
   x = 10
   print "x=10"
   x = mymod.add2(x)
   x = mymod.minus2(x)
   print "x=",x


if __name__ == "__main__":
   print “I run when
          executed”
  main()
```

Modules ...

```
#!/usr/bin/python
# myothermod.py
from mymod import add2,minus2

def main():
    x = 10
    print "x=10"
    # Now in Context
    x = add2(x)
    x = minus2(x)
    print "x="x


if __name__ == "__main__":
    print “I run when
        executed”
   main()
```

No Protection

- Programmer can arbitrarily add to an imported module

```
>>> import math
>>> math.x = 1
>>> print math.x
1
```

- Underscore for obfuscate? Not in modules ...

```
#priv_mod.py

__x = 1
__y = 2
print __x,__y
```

```
>>> import priv_mod
1 2
>>> priv_mod.__x,priv_mod.__y
>>> dir(priv_mod)
['__builtins__', '__doc__', '__file__',
'__name__', '__package__', '__x', '__y']
```

Packages

- Groupings of modules in directory
  - Control of import procedure
    - What gets imported and what doesn't
- **`__init__.py`**
  - Special file the describes the package and the import procedure
  - Placed in the directory with modules
  - Contains modules to import, preprocessing, and other package operations

More Packages

- Directory referred to as *container*
  - Sub-directories are allowed
- Each container contains a __init__.py file

```
mymodules\
   __init__.py
   math\
      __init__.py
      mytrig.py
      mycalc.py
   xml\
      __init__.py
      myHTML.py
      MyXML.py
```

```
>>> import mymodules.math.mytrig
```

__init__.py

- Can peform a slew of intializations
  - Module variables, functions, etc.
  - Control what is importe (some privacy protection)
- Define what happens on an **`import *`**
  - **`__all__`** list of modules to import
- Generally, doesn't need any code but *must* be in place if you want to import from the directory

What module have we already used?

- Where do all these functions come from
  - `range(), xrange()`
  - `int(), float(), open()`
- What about **`EOFError`**?
- *They must come from somewhere!!*
  - before running anything python essentially executes
  - `from __builtins__ import *`

dir(__builtin__)

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception',
'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__debug__', '__doc__',
'__import__', '__name__', 'abs', 'all', 'any', 'apply', 'basestring', 'bool',
'buffer', 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len',
'license', 'list', 'locals', 'long', 'map', 'max', 'min', 'object', 'oct',
'open', 'ord', 'pow', 'property', 'quit', 'range', 'raw_input', 'reduce',
'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode',
'vars', 'xrange', 'zip']
```

The `sys` Module

- You will use the **`sys`** module often
  - *This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter*
- **`stdin, stdout, argv, exit()`**
- **`ps1, ps2`**
- Very useful stuff
  - Traceback inormation,

`readline` Module

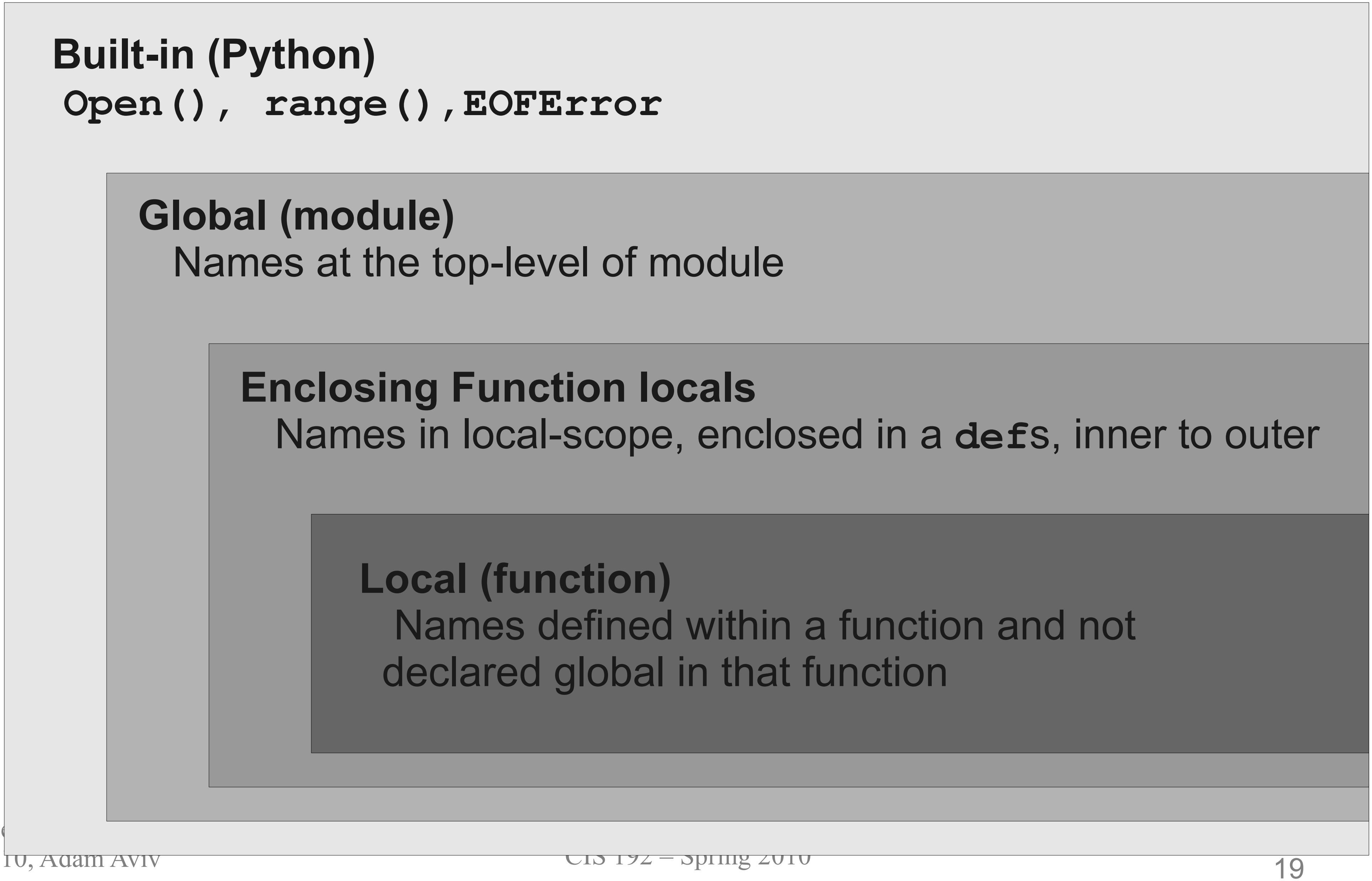
- Nice features for **`raw_input()`**
- Emacs bindings, history, etc.
- Only have to import it, DEMO
  - Doesn't work on Windows

```
#!/usr/bin/python
# readline_ex.py
import sys,readline

if __name__ == "__main__":

   while True:
      s = raw_input('$>')
      print “You Said”, s
```

Scoping

- Enclosing module is a global scope
- Global scope spans a single file only
- Each call to a function creates a new local scope
- Assigned names are local unless *declared* global
- All other names are enclosing locals, globals, or built-ins

LEGB rule



Built-in (Python)

`Open(), range(),EOFError`

Global (module)

Names at the top-level of module

Enclosing Function locals

Names in local-scope, enclosed in a **`def`**s, inner to outer

Local (function)

Names defined within a function and not declared global in that function

Scoping Example

- What is the output of these program?

```
#!/usr/bin/python
# global_ex1.py

a = 10
def add_2_a():
   a = a + 2
   print a
print a
```

```
#!/usr/bin/python
# global_ex2.py

a = 10
def add_2_a():
   a = 12
   print a
print a
```

The `global` tag

- Global moves a variable into the local context

```
#!/usr/bin/python
# global_ex3.py

a = 10
def add_2_a():
    global a
    a += 2
    print a

add_2_a()
print a
```

```
#!/usr/bin/python
# global_ex4.py

def add_2_a():
    global a
    a += 2
    print a

add_2_a()
... NameError: global name 'a'
    is not defined
```

Nested Scoping

- **`def`** is an executable statement
- How does this scoping work?

```
#!/usr/bin/python
# nesting_ex1.py

def f1():
    x = “hello”
    def f2(): return x
    return f2()

print f1()
```

```
#!/usr/bin/python
# nesting_ex2.py

def f1():
    x = “hello”
    def f2(): return x
    return f2

fun = f1()
print fun()
```

Mutable is Different

```
#!/usr/bin/python

b = []

def f1():
    b.append(1) #<--- this is allowed, why?
    print b

f1()
print b

def f2():
    b += [1] #<--- something is fishy here, why?
    print b

f2()
print b
```

Nesting Scope (cont)

- What is going on here? Why does this work?

```
#!/usr/bin/python
# def_nesting3.py

def exp(N):
   def action(X):
      return X ** N
   return action

exp_2 = exp(2)
exp_3  = exp(3)

print exp_2(2)
print exp_3(2)
```