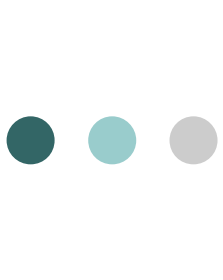
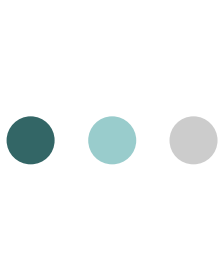
Files, Scopes, Name Spaces

HANDS-ON INTRODUCTION TO PYTHON



File

- Persistent storage even after program ends
- Represented in Python as type *file* (object)
- Typical file processing involves:
  1. File open
  2. Read/write operations
  3. File close



File methods

f = open(“filename”)	open a file, return file value
f = open(“filename”, “w”)	open a file for writing
f.read()	return a single character value
f.read(n)	return no more than n character values
f.readline()	return the next line of input
f.readlines()	return all the file content as a list
f.write(s)	write string s to file
f.writelines(lst)	write list lst to file
f.close()	close file

File

- **`readline`** return the next line of text, including a newline (return) character at the end
- Returns empty string when file empty

- Using for statement can also have the same result

```
>>> f = open('message.txt')
>>> for line in f:
...   print line
```

Operating System Command

- Useful OS command can be executed from python by including os module

```
>>> import os
>>> os.remove("gone.txt")

>>> os.curdir
'.'
>>> os.rename('oldfile.ext', 'newfile.ext')
>>>
```

Recovering from Exceptions

- File I/O operations can generate exceptions, an IOError

```
try:
    f = open('input.txt')
except IOError, e:
    print 'unable to open the file'
else:
    print 'continue with processing'
    f.close()
print ‘continue’
>>> unable to open the file
continue
```

Standard I/O

- **`print`** writes characters to a file normally attached to display window
- Input functions read from a file attached to keyboard
- These files can be accessed through **`sys`** module
- Input file : **`sys.stdin`**, output file: **`sys.stdout`**, error messages: **`sys.stderr`**
- **`stderr`** normally goes also to **`stdout`**

Standard I/O

- Can change these settings through sys

```
import sys
sys.stdout = open('output.txt', 'w')
sys.stderr = open('error.txt', 'w')
print "see where this goes"
print 5/4
print 7.0/0
sys.stdout.close()
sys.stderr.close()
```

In output.txt

```
see where this goes
1
```

In error.txt

```
Traceback (most recent call last):
  File "C:\Python26\...\scriptutils.py", line 325, in RunScript
    exec codeObject in __main__.__dict__
  File "lab3.py", line 14, in <module>
    print 7.0/0
ZeroDivisionError: float division
```

OS functions

- **`exit`** terminate a running Python program – **`sys.exit("message")`**
- **`sys.argv`** is a list of command line options being passed

```
import sys
print 'argument of program are ', sys.argv
>>>argument of program are
  ['D:\\shor\\COURSE\\1040\\lab\\lab3\\lab3.p
  y']
```

Pickle

- Useful in saving and restoring Python variables
- Also called serialization

```
import pickle
stackOne = Stack()
stackTwo = Stack()
stackOne.push( 12 )
stackTwo.push( 'abc' )
stackOne.push( 23 )
stackOne.pop()
f = open('pickle1.pyp','w')
pickle.dump([stackOne,
   stackTwo], f)
```

- Later on

```
>>> import pickle
>>> f = open('pickle1.pyp')
>>> [stackOne, stackTwo] =
   pickle.load(f)
>>> print stackOne.top()
12
>>> print stackTwo.top()
abc
```

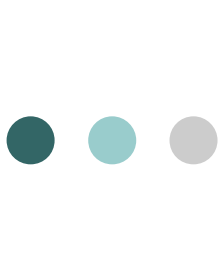
Reading from a URL

- **`urllib`** helps to read content of a file stored at a specific URL

```
import urllib
remotefile = urllib.urlopen("http://www.cse.cuhk.edu.hk")
line = remotefile.readline()
while line:
    print line
    line = remotefile.readline()
```

- Hides all the details of network access

The urllib module has been split into parts and renamed in Python 3.0 to urllib.request, urllib.parse, and urllib.error.



Identifiers in Program

- Names of variables, functions, modules can collide with others – same name used unintentionally
- Managed using name spaces
- Encapsulation of names through levels of abstraction
- Three levels of encapsulation
  - LEGB rule for simple variables
  - Qualified names
  - modules

LEGB

- **L**: local
- **E**: Enclosing function definitions
- **G**: Global
- **B**: built-in functions
- When Python is looking for meaning attached to a name, it search the scope in the order: *Local, Enclosing, Global, Built-in*

```
>>> x = 42
>>> def afun():
...     x = 12
...     print x
>>> afun()
12
>>> print x
42
```

Enclosing

- Occurs when one function is defined inside another
- Each function definition creates a new scope for variables at that level

```
>>> def a(x):
...     def b(y):
...             print x + y
...     y = 11
...     b(3)
...     print y
...
>>> a(4)
7
11
```

Scopes

- Described as a series of nested boxes
- To find a match for a given variable, the boxes are examined from inside out until the name is found
- Lambda create their own local scope
- Thus distinct from surrounding function scope

```
>>> def a(x):
...     f = lambda x: x + 3
...     print f(3)
...     print x
...
>>> a(4)
6
4
```

Built-in functions

- Functions that are initially part of any Python program e.g. open, zip, etc
- Can be overridden in a different scope
- For example, a programmer can define his/her own open.
- However it will prevent access to the standard function i.e. file open

dir function

- dir can be used to access a list of names in current scope
- Get global scope in topmost level

```
>>> z = 12
>>> def dirtest():
...     x = 34
...     print dir()
>>> print dir()
['__builtins__', '__doc__', '__name__',
  '__package__', 'dirtest', 'pywin', 'z']
>>> dirtest()
['x']
```

dir function

- Can accept an argument
- Return scope of the object

```
>>> dir (dirtest)
['__call__', '__class__', '__closure__', '__code__', '__defaults__',
    '__delattr__', '__dict__', '__doc__', '__format__', '__get__',
    '__getattribute__', '__globals__', '__hash__', '__init__', '__module__',
    '__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
    'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc',
    'func_globals', 'func_name']
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
    'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
    'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp',
    'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
    'tanh', 'trunc']
```

Global statement

- Tells the compiler indicated name is to refer to the variable in global scope rather than default(local)

```
>>> def fun():
...     global x
...     x = 42
>>> x=12
>>> fun()
>>> print x
42
```

- Used only when variable need be assigned

Class scope

- Class has its scope, but not part of LEGB
- A class method can see their surrounding scope, but cannot see the class scope
- Normally it's okay as classes defined at top level

```
def silly():
    x = 12
    class A:
        x = 42
        def foo(self):
            print x
            print self.x
    return A()
anA = silly()
anA.foo()
>>> 12
42
```

Class variables

- Variables defined at class level are shared by all instances
- Initialization only once
- Variables defined using self are unique to each instance

```
class CountingClass:
    count = 0
    def __init__(self):
        CountingClass.count = CountingClass.count + 1
>>>a = CountingClass()
>>>b = CountingClass()
>>>print CountingClass.count
2
```

Scopes, Names, and References

- Scope is property of a name, not a property of a value
- Two names can refer to the same value, and they have different scopes

```
class Box(object):
    def __init__( self, v):
        self.value = v

def newScope(x):
    y = x
    y.value = 42

a = Box(3)
newScope(a)
print a.value
>>>42
```

Qualified Names

- A period following a base e.g. **`object.attribute`**
- Base is first determined using LEGB rule
- Names can be qualified include
  - Classes
  - Instances or objects
  - Instances of built-in types e.g. list, dictionary
  - Modules

Qualified Names

- Names resolution are performed using dictionaries
- locals() and globals() return the current scope through dictionary
- Classes store their name space in a field __dict__
- Can be accessed (or modified!) by programmer

```
>>> CountingClass.__dict__
{'count': 2, '__module__': '__main__',
  '__doc__': None, '__init__': <function
  __init__ at 0x00FDE4F0>}
```

Modules

- Simply a Python file
- Only the handling of names in modules differs
- Import statement scans a file and execute each statement in program
- Names of all values in module are stored in its own dictionary
- Thus the qualified name modName.x is actually just modName.__dict__[‘x’]

```
>>> import string
>>> print type(string)
<type 'module'>
>>> print string.__dict__['split']
<function split at 0x00BD81B0>
```

Module

- Just like library
- Can have two ways when used:
  - **`Import modName`**
  - **`from modName import attribute`**
- For second way of using module
  - Means construct the module dictionary, the given attribute is then copied into local dictionary
  - Thus the attribute can be used without qualification in local space

Module

- Suppose we want to use bar method in module foo
- **`import foo`**
- .. Then use foo.bar to use it
- 2 run-time lookups needed in this case :
  1. locate foo,
  2. locate bar

- **`from foo import bar`**
- bar is called directly without qualification
- Only One search required
- More execution efficiency

Avoid Name Space collision

- Can use wild card '*' to import - from mod import *
- Has risk of name collision

```
>>> from math import *
>>> print e
2.71828182846
```

- Can Use as clause to avoid

```
>>> e = 42
>>> from math import e as eConst
>>> e
42
```

Creating your own module

- Just another Python program
- Only difference is it is being loaded by *import* statement
- Normally contains only classes and function definitions
- Can also have statements inside be executed
- Name of current module is held in internal variable called **`__name__`**
- Top level program executed by Python interpreter is of the name **`__main__`**
- Can use the following to conditionally executing those statements

```
if __name__ == '__main__':
    .. statements
```