LECTURE 5: OBJECT ORIENTED PROGRAMMING

ANNOUNCEMENTS

- QUIZ 1: September 20th
- 30 minutes, closed-book, no notes, no internet
- Mostly short answers

MAPPING, FILTERING, REDUCTION

- Mapping: One-to-one transformation through a function
  - [1,2,3,4,5] -> [1,4,9,16,25] (lambda x: x**2)
- Filtering: Apply a condition, retain ones that satisfy the condition
  - [1,2,3,4,5] -> [2,4] (lambda x: x % 2 == 0)
- Reduction: Apply a binary function to each member of a list in a row
  - [1,2,3,4,5] -> 15 (lambda x,y: x + y)
  - (((1+2)+3)+4)+5=15
- Reserved keywords: filter, map, reduce

EXAMPLES

```
>>> lst = [1,2,3,4,5]
>>> list(map(lambda x: x ** 2,lst))
[1, 4, 9, 16, 25]
>>> filter(lambda x:  x % 2 == 0,lst)
[2, 4]
>>> functools.reduce(lambda x,y: x + y,lst)
15
>>> list(map(lambda x: 1 if x % 2 == 0 else 0,lst))
[0, 1, 0, 1, 0]
```

REDUCE IN PYTHON3

- Import functools first
  - import functools

EXERCISE 1: FIND VALUES FOR KEYS

```
>>> c={'a':1,'b':2,'c':3}

>>> keys = ['a','b','c']

>>> list(map(lambda x: c[x],['a','b','c’])) # method 1
[1, 2, 3]

>>> list(map(lambda x,y: y[x],['a','b','c'],[c,c,c])) # method 2
[1, 2, 3]
```

FEATURES OF THE RESULTS

- The original list remains unchanged
- The three functions map, filter, and reduce produce new lists that are transformation of the argument
- A function that uses another function that is passed as an argument is referred to as a higher order function

LIST COMPREHENSION

```
>>> lst = [1,2,3,4,5]
>>> [x**2 for x in lst]
[1, 4, 9, 16, 25]

>>> [x**2 for x in lst if x%2 ==0]
[4, 16]

>>> [1 if x%2==0 else 0 for x in lst]
[0, 1, 0, 1, 0]
```

PYTHON CLASSES

- A class: a collection of functions and data values unified and linked to a common purpose
- To define a class in Python: Use the keyword *class*, followed by the class name, a parenthesized list of parent classes, and a colon
  - class MyClass:
- The body contains a collection of functions called methods and data values

PYTHON CLASSES

```
class BankAccount():
    def __init__(self):
        self.balance = 0

    def deposit(self,amount):
        self.balance += amount

    def withdraw(self,amount):
        self.balance -= amount

    def get_balance(self):
        return self.balance
```

PYTHON CLASSES

```
>>> my_account = BankAccount()
>>> my_account.get_balance()
0
>>> my_account.deposit(200)
>>> my_account.get_balance()
200
>>> my_account.withdraw(100)
>>> my_account.get_balance()
100
```

CLASSES VS OBJECTS

- Data fields: data that an object requires. A new data field is created the first time it is assigned a value
  - self.balance = 0
- Instances of a class are called *objects*.
  - my_account = BankAccount()
- self ← → this in Java
- __init__ function is the constructor
- Not directly invoked, called implicitly during object creation

CALLING OTHER METHODS

- See bank-account-xfer.py

OBJECTS ARE REFERENCES!

- Objects are internally stored as references.
- This is significant for both assignment and parameter passing

INHERITANCE CASE STUDY: A CHECKING ACCOUNT

- A checking account can process and record checks, in addition to the behavior of standard bank accounts
- Checks have numbers, to whom they are written, as well as an amount
- A dictionary is used to maintain all checks written; the check number is used as the key
- BankAccount: the parent class
- CheckingAccount: the child class
- Inheritance: the child class has all functions and data fields of the parent class, as well as its own new functions and data fields

INHERITANCE

```
class CheckingAccount (BankAccount):
    def __init__ (self, initBal):
        BankAccount.__init__ (self, init_bal)

        self.check_record = {}

    def process_check (self, number, to_who, amount):

        self.withdraw(amount)

        self.check_record[number] = (to_who, amount)

    def check_info (self, number):

        if self.check_record.has_key(number):

            return self.check_record[number]

        else:

            return 'no such check'
```

CONSTRUCTOR IN THE CHILD CLASS

- The constructor of the child class must explicitly invoke the constructor of the parent class
- When a class name is on the left of the dot, self must be passed as an explicit argument
- When self is on the left, it is omitted from the argument list

```
class CheckingAccount(BankAccount): # inheritance
    def __init__ (self, initBal):
        BankAccount.__init__ (self, init_bal) # important format!
        self.checkRecord = {} # own data field initialization
```

CHILD CLASS OPERATION

Create a checking account with an initial balance of $300

```
ca = CheckingAccount(300)
ca.processCheck(100, 'Gas Company', 72.5)
ca.processCheck(101, 'Electric Company', 53.12)
print(ca.checkInfo(100))
```

Child class inherits methods from parent, can invoke any of them

```
ca.deposit(50)
print(ca.get_balance())
```

See checking-account.py

CLASSES, TYPES AND TESTS

- Each class definition creates a new type. Use the function type() to check
- To test for the type/class of an object, use the built-in function isinstance(obj, cname)
  - True if obj belongs to the class cname or any child class of cname
- The built-in function issubclass(A, B) returns True if the class A is a subclass of B
- See is-test.py

NAME SPACES AND MODULES

- Name space
  - The LEGB rule
- Modules
  - The import statement
  - Creating own modules

SCOPE OR NAME SPACE

- A scope, or name space, is an encapsulation or a packaging of names
  - A single entity from the outside
  - A collection of names and values from the inside
- Different scopes can hold values with the same name without danger of collision
  - This allows programmers to use short, easy-to-remember names

THE LEGB RULE

- When Python looks for a meaning for a simple variable/name, it searches the scopes in the order: Local, Enclosing, Global and Built-in
- When defining functions
  - Variables assigned within the functions are local
- Local: only within the body of the function
- Global: the scope for variables defined at the top level of a program

THE LEGB RULE

- Enclosing occurs when one function is defined inside another
  - Each function definition creates a new scope
  - A lambda function also creates its own local scope
- Built-in scope contains built-in functions
  - Many functions, see: http://docs.python.org/2/library/functions.html
  - The final scope

EXAMPLES

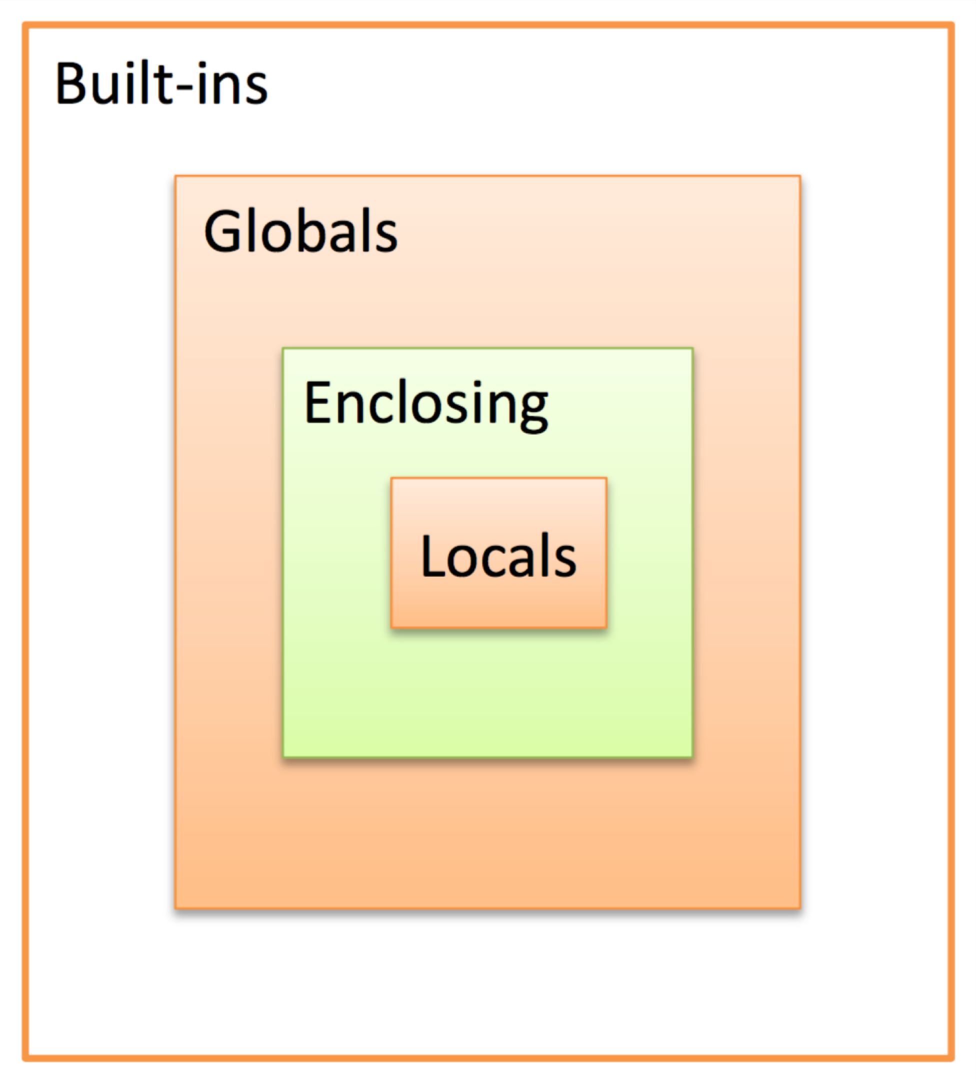
- **Local vs Global**
- ```
  >>> x = 42
  def afun():
      x = 12
      print(x)
  >>> afun()
  12
  >>> print(x)
  42
  >>>
  ```

- **Enclosing**
- ```
  def a(x):
      def b(y):
          print(x + y)
      y = 11
      b(3)
      print(y)
  >>> a(4)
  ```

EXAMPLES

- Enclosing a lambda function

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



MODULES

- A module is simply a Python file
- The import statement e.g., import foo scans a file, executes each statement in the program
  - It constructs a dictionary for all names within the module foo
- foo.bar() at execution time invokes two run-time lookups:
  - One finds the meaning for foo in the current name space
  - One finds the data field bar in foo's name dictionary

FROM FOO IMPORT BAR

- It imports a single function, not a module
- Python first constructs the module dictionary for foo, then copy the given attribute bar from this dictionary into the caller's local dictionary
- The attribute can then be used without qualifications
- bar() invokes just one look-up, in the local scope
- Be careful with "from math import *"
  - Any name in the current scope that matches a name in the imported scope will be overridden

FROM EXAMPLES

- Local is overridden
- ```
  >>> e = 42
  >>> from math import *
  >>> print(e)
  2.71828182846
  ```

- A remedy
- ```
  >>> e = 42
  >>> from math import e as e_const
  >>> print(e)
  42
  >>> print(e_const)
  2.71828182846
  ```

CREATING ONE'S OWN MODULES

- A Python module is just a Python program with a list of Python statements in it, saved as a file
  - The name of the module derived from the name of the file
- Normally files that are used as modules contain only class and function definitions
- See example mycollection.py and use-it.py

MODULE VS PACKAGE

- Package is a generalization of the module concept for larger projects
- One can group a set of modules into a directory or a directory tree, then import and hierarchically refer to them using *package.subpackage.module* syntax
- To create packages, some simple special files are needed besides the modules
  - https://docs.python.org/3/tutorial/modules.html#packages

FIN!