Dictionaries, OOP, Functional Programming

HANDS-ON INTRODUCTION TO PYTHON

Dictionaries

- Indexed data structure - uses also square bracket notation
- Any immutable type can be used as index
- Braces create dictionary

```
>>> dct = { } # create new dictionary
>>> dct['name'] = 'Donald Duck‘
>>> dct['age'] = 63
>>> dct['eyes'] = 'black‘
```

- Index is called a key (LHS)
- Element stored that associated with key is called a Value (RHS)

Dictionaries

- Also called maps, hashes or associative arrays

```
>>> print dct['name']
Donald Duck
>>> print dct.get('age')
63
>>> print dct['weight']
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
KeyError: 'weight'
>>> print dct.get('weight', 0) # 0 is default value
0
>>> dct['age'] = 21
>>> dct['age']
21
```

- Index is called a key (LHS)
- Element stored that associated with key is called a Value (RHS)

Dictionaries

- Del used to delete an element from list

```
>>> del dct['age']
>>> print dct['age']
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
KeyError: 'age`
>>>
```

- Can be initialized also

```
>>> info = {'name':'Batman', 'age':53, 'weight':200}
>>> print info['name']
Batman
```

Dictionary operations

Operation	Description
len(d)	number of elements in d
d[k]	item in d with key k
d[k]=v	set item in d with key k to v
d.clear()	remove all items from dictionary d
d.copy	make a shallow copy of d
d.has_key(k)	return 1 if d has key k, 0 otherwise
d.items()	return a list of (key,value) pair
d.keys()	return a list of keys in d
d.values	return a list of values in d
d.get(k)	same as d[k]
d.get(k,v)	return d[k] if k is valid, otherwise return v

Return list in Python 2.7 or earlier, but return view in 3.1 or later

Inheritance

- New class has all the functionalities of parent class

Specialized variations on an abstract concept

```
class CheckAccount( BankAccount):
    def __init__(self, initBal):
        BankAccount.__init__(self, initBal)
        self.checkRecord = { }
    def processCheck(self, number, toWho, amount):
        self.withdraw(amount)
        self.checkRecord[number] = (toWho, amount)
    def checkInfo(self, number):
        if self.checkRecord.has_key(number):
            return self.checkRecord [ number ]

ca = CheckAccount( 1000 )
ca.processCheck(100, 'town Gas', 328.)
ca.processCheck(101, 'HK Electric', 452.)
print ca.checkInfo(101)
>>>('HK Electric', 452.0)
print ca.getBalance()
ca.deposit(100)
print ca.getBalance()
```

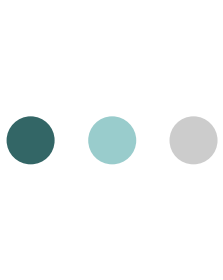
Inherit methods from parent

Inheritance

- Exception handling still in place, and flow control is halted

```
print ca.getBalance()
220.0
ca.processCheck(101, 'mortgage', 15000)
Traceback (most recent call last):
:
File "1040\test1.py", line 28, in <module>    ca.processCheck(101,
   'mortgage', 15000)
File "1040\test1.py", line 18, in processCheck
   self.withdraw(amount)
File "1040\test1.py", line 8, in withdraw
    raise ValueError, 'insufficient funds`
ValueError: insufficient funds
```

- Worst situation happens when the exception is being handled silently
- processCheck has not finish => record was not updated (inconsistency)



Software Reuse

- Inheritance will reuse code from the parent class
- Saving in development time as the reused code is debugged
- A class in Python can be distributed as a library and further reused worldwide

Overriding

- Sometimes necessary for child class to modify or replace the behavior inherited from parent class
- Child class redefines the function using the same name and arguments
- To invoke original parent class function, class name must be explicitly provided

```
class CheckAccount( BankAccount):
    :
    def withdraw(self, amount):
        print 'withdrawing ', amount
        BankAccount.withdraw(self, amount)
```

Types & Tests

- Each class definition creates a new type

```
>>> print type(myAccount)
<class '__main__.BankAccount'>
>>> print type(BankAccount)
<type 'type'>
```

- Test for membership in a class

```
>>> newAccount = CheckAccount(4000)
>>> sndAccount = BankAccount(100)
>>> print isinstance(newAccount, BankAccount)
True
>>> print isinstance(newAccount, CheckAccount)
True
>>> print isinstance(sndAccount, CheckAccount)
False
```

Types & Tests

- issubclass(A,B) returns true if class A is a subclass of B

```
>>>print issubclass(CheckAccount, BankAccount)
True
```

- **Can also perform type checking for built-in types**

```
>>>import types
>>>isinstance(3, types.IntType)
True
>>>isinstance(3, types.FloatType)
False
```

Multiple Inheritance

- Class definition specify inheritance from more than one class
- Not recommended

```
class A(object):
    def doa(self):
        print "I'm a"
class B (object):
    def dob(self):
        print "I'm b"
class C(A,B):
    def doc(self):
        print "I'm c"
>>> v=C()
>>> v.doc()
I'm c
>>> v.doa()
I'm a
>>> v.dob()
I'm b
```

Classes as Dynamic Record

- Class data field should in general not be used outside class definition
- ```
  class EmTee(object):
      pass
  ```
- A null class definition

- Create data structures that have only a single instance
  ```
  >>>myData = Emtee()
  >>>myData.name = 'Donald Tsang'
  >>>myData.age = 63
  ```
- Allow a number of values collected under a name
  ```

Programming Paradigm

- Mental model a programmer envisions as creating program
- Imperative paradigm
  - computer is a combination of processor and memory
  - Instructions have the effect of making changes to memory
  - Desired results produced by arranging sequence of instructions to transform the memory
- C, BASIC all belongs to this school of languages

Functional Programming

- Values are represented as lists, or dictionaries
- Transformation to the lists/dictionaries are made
- Works on larger scale to achieve the objective
- Three most common forms of transformation
  1. Mapping
  2. Filtering
  3. reduction

Functional Programming

- Mapping
  - One-to-one transformation for each member
  - [1,2,3,4,5] f(2*x+1)=>[3,5,7,9,11]
- Filtering
  - Testing and retain member which pass a function e.g.
  - [1,2,3,4,5] test for odd => [1,3,5]
- Reduction
  - Applying a binary function to member in cumulative manner e.g.
  - [1,2,3,4,5] => ((((1+2)+3)+4)+5)=15

Lambda function

- Passing function to filter

```
def even(x):
    return x % 2 == 0
a = [1,2,3,4,5]
print filter(even, a)
>>> [2, 4]
```

- But since functions being passed to map, filter & reduction are usually very simple
- using *def* function becomes quite cumbersome
- lambda is used to pass simple function

```
lambda x, y : x + y
```

Lambda

- A nameless function lambda is passed to map, filter, reduce

```
print map(lambda x : x *2 + 1, a)
[3, 5, 7, 9, 11]
print filter( lambda x : x % 2 == 0, a)
[2, 4]
print reduce( lambda x, y: x + y, a)
15
```

- Filter requires a function that takes only one argument and return a Boolean value – called *predicate*

List Comprehensions

- A list characterized by a process
  [*expr* **for** *var* **in** *list* **if** *expr*]
- If part optional
- Each element in list is examined
- If element pass ‘if expr’, ‘expr’ is evaluated and result add to new list

```
a = [1,2,3,4,5]
print [x*2 for x in a if x < 4]
[2, 4, 6]
```

List Comprehensions

- Used as body of a function

```
def ListofSquares( a):
    return [x*x for x in a]
>>> ListofSquares([1,2,3])
[1, 4, 9]
```

- Operations on dictionaries performed by selecting values from range of keys, then returning items with selected keys

```
d = {1:'fred', 7:'sam', 8:'alice', 22:'helen'}
>>>[d[i] for i in d.keys() if i%2==0]
['alice', 'helen']
```

Objected Oriented Programming

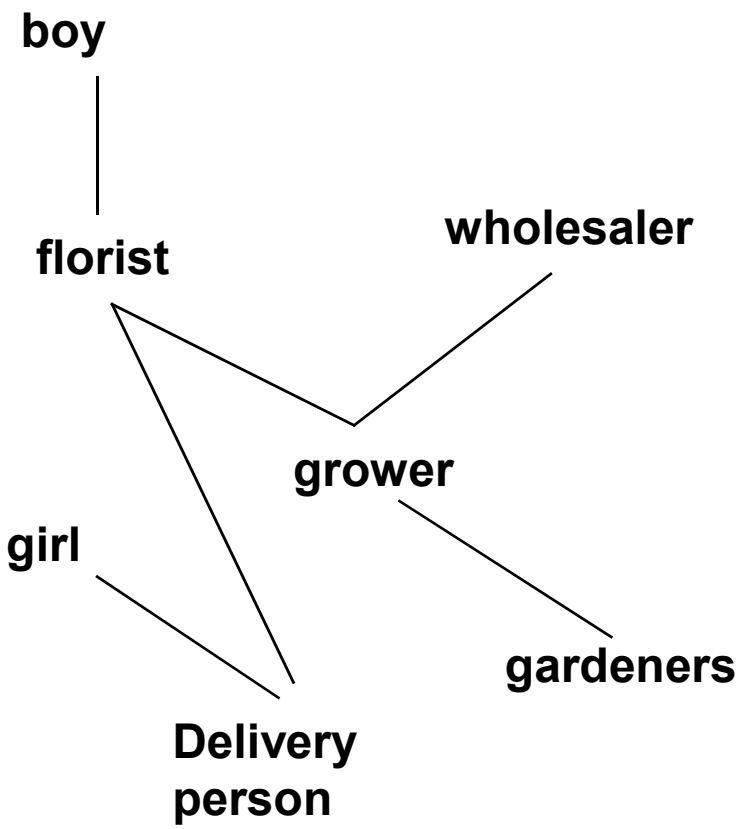
- A program is viewed as a collection of computing agents (objects)
- each of which provide a service that can be used by other
- Objects interact with each other by invoking functions defined within class – message passing
- Objects cooperated together to achieve a task

Objected Oriented Programming

- Message passing is comparable to solving problem in real world
- Consider the task of sending flowers on Feb 14 to the girl friend of a boy
- He pass a message to a florist requesting for this service
- The florist acquired flowers by dealing with a wholesaler
- The wholesaler interact with growers, delivery persons, and so on.

Objected Oriented Programming

- Intuition and skills from life experiences can readily be applied to object-oriented programs
- One of reasons of OOP becoming dominant paradigm in recent years



Stack in OOP

```
class Stack(object):
    def __init__(self):
        self.storage = []
    def push (self, newValue):
        self.storage.append( newValue )
    def top( self ):
        return
   self.storage[len(self.storage) - 1]
    def pop( self ):
        result = self.top()
        self.storage.pop()
        return result
    def isEmpty(self):
        return len(self.storage) == 0
```

```
stackOne = Stack()
stackTwo = Stack()
stackOne.push( 12 )
stackTwo.push( 'abc' )
stackOne.push( 23 )
print stackOne.top()
>>> 23
stackOne.pop()
print stackOne.top()
>>>12
print stackTwo.top()
>>>'abc'
```

Using stack to build a calculator

```
class CalculatorEngine(object):
    def __init__(self):
        self.dataStack = Stack()
    def pushOperand (self, value):
        self.dataStack.push( value )
    def currentOperand ( self ):
        return self.dataStack.top()
    def performBinary (self, fun ):
        right = self.dataStack.pop()
        left = self.dataStack.top()
        self.dataStack.push( fun(left, right))
    def doAddition (self):
        self.performBinary(lambda x, y: x + y)
    def doSubtraction (self):
        self.performBinary(lambda x, y: x - y)
    def doMultiplication (self):
        self.performBinary(lambda x, y: x * y)
    def doDivision (self):
        self.performBinary(lambda x, y: x / y)
```

```
def doTextOp (self, op):
        if (op == '+'):
   self.doAddition()
        elif (op == '-'):
   self.doSubtraction()
        elif (op == '*'):
   self.doMultiplication()
        elif (op == '/'):
   self.doDivision()


calc = CalculatorEngine()
calc.pushOperand( 3 )
calc.pushOperand( 4 )
calc.doTextOp ( '*' )
print calc.currentOperand()

>>> 12
```

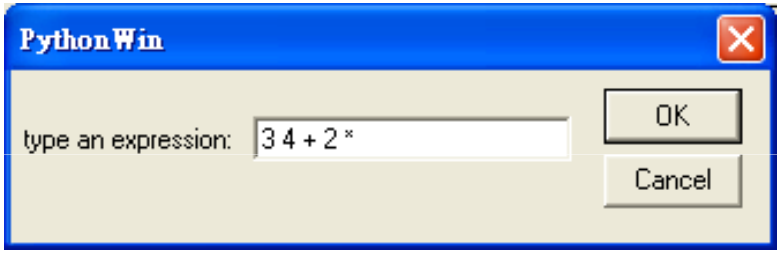
Calculator Interface

```
class RPNCalculator(object):
    def __init__(self):
        self.calcEngine = CalculatorEngine()
    def eval (self, line):
        words = line.split(" ")
        for item in words:
            if item in '+-*/':

    self.calcEngine.doTextOp( item )
            else:

    self.calcEngine.pushOperand( int (item))
        return
    self.calcEngine.currentOperand()
    def run(self):
        while True:
            line = raw_input("type an
    expression: ")
            if len(line) == 0:
                break
            print self.eval( line )
```

```
calc = RPNCalculator()
calc.run()
```




```
>>>14
```

Separating Model from View

- The calculator is constructed from 3 classes – stack, calculator, interface
- considered independent of each other
- Calculator engine encapsulates logic of using stack to perform evaluation
- But calculator knows nothing about interface

- Advantages:

1. Make program easier to understand
2. Enable software reuse e.g. stack
3. Division of calculator engine and interface
    1. View is the object that interacts with end user
    2. Model is the logic that actually implements the tasks being performed

- Can also change the interface to others without need to rewritten the calculator engine

Suggested Readings

- Ch. 5, Ch. 7 p.118 – 123, Ch. 8, Ch. 9, in Exploring Python – Timothy
- Section 4.75, 5.1, 5.2, 5.5, 9.5-9.8, in Python tutorial (official 2.7.6 doc)
- Section 9.9 if you want to learn also the use of iterators