Functional Programming

Computer Science & Engineering Department

The Chinese University of Hong Kong

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

Content

- Functional Programming Paradigm
- Map, Filter, Reduce and Lambda
- Supplementary: Iterator class

Python Function

- Name of a function is a reference to an *object* representing that function
- We can assign that function to another variable

```
import math
mySqrt = math.sqrt
mySqrt(4)   # return 2.0
```

- Function can therefore also be passed as parameter

Reference : http://python-history.blogspot.com/2009/02/first-class-everything.html

Functional Programming

- Variables are represented as lists, tuple or dictionaries
- Transformation to all or part of these variables are made
- Three most common forms of transformation
  1. Mapping
  2. Filtering
  3. Reduction (from functools)

List Comprehensions

- A list characterized by a process
  [*expr* **for** *iter* **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']
```

Functional Programming

- Mapping
  - One-to-one transformation for each member
  - map needs a function which is to be applied to all data
  - Syntax
    `map(func, seq)`
  - func : function to be applied
    seq : sequence of data eg. List, dictionary, etc

  - 2.X returns a list whereas 3.X returns an iterator

Functional Programming

- Mapping
  - [1,2,3,4,5] => f(2*x+1) => [3,5,7,9,11]

```
def map_func1(x): return x*2 + 1

my_variable = [1,2,3,4,5] # define a list

# apply map_func1 to my_variable, and convert the result to a list
my_variable = list (map(map_func1, my_variable))
print("my_variable is : ", my_variable)
```

```
my_variable is :  [3, 5, 7, 9, 11]
```

Functional Programming

- More than one arguments possible

```
# define function
def my_add (x,y): return x+y

my_variable = list(map(my_add, range(0,10), range(0,10)))
print(my_variable)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Functional Programming

- Filtering
  - Testing and retain member which pass a function e.g.
  - [1,2,3,4,5] test for odd => [1,3,5]

```
def my_func1(x): return x % 3 == 0 or x % 5 == 0
def my_func2(x): return (x+1)%2 == 0

my_variable = list(filter(my_func1, range(1,25)))
print("1. my_variable = ", my_variable)
my_variable = [ x for x in filter(my_func2, range(1,6))]     # use list comprehension
print("2. my_variable = ", my_variable)
```

```
1. my_variable =  [3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
2. my_variable =  [1, 3, 5]
```

Functional Programming

- Reduction
  - Applying a binary function to member in cumulative manner e.g.
  - [1,2,3,4,5] => ((((1+2)+3)+4)+5)=15
  - To use it in Python 3.x, we need to import it

```
from functools import reduce
#define a reduce function
def my_add(x,y): return x+y

print(reduce(my_add, range(1,6)))
```

```
15
```

Lambda function

- Passing function to filter

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

Lambda function

- lambda is used to pass simple function

`lambda argument_list : expression`

```
def sum(x,y): return x + y
# an alternate way to write this small function is via lambda
sum1 = lambda x, y : x + y

print (sum (3,4))
print (sum1(3,4))
```

```
7
7
```

Lambda

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

```
a = [1,2,3,4,5]
print (list(map(lambda x : x *2 + 1, a)))
>>> [3, 5, 7, 9, 11]
print (list(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*

Revisit

Using stack to build a calculator

```python
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 doAddition (self ):
        right = self.dataStack.pop()
        left = self.dataStack.pop()
        self.dataStack.push(left + right)

# …

# similarly for doSubtraction, doMultiplication and doDivision
```

```python
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
```

Revisit

Using stack to build a calculator

```
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
```

Using stack to build a calculator (version 2)

```
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 performBinaryOp (self, fun ):  # generalized method
        right = self.dataStack.pop()
        left = self.dataStack.pop()
        self.dataStack.push( fun(left, right))

    def doAddition (self ):
        self.performBinaryOp(add)

# ...

# similarly for doSubtraction, doMultiplication and doDivision
```

```
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

def mul(x, y):
    return x * y

def div(x, y):
    return x / y
```

Using stack to build a calculator (version 3)

```
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 performBinaryOp (self, fun ):
        right = self.dataStack.pop()
        left = self.dataStack.pop()
        self.dataStack.push( fun(left, right))

    def doAddition (self ):
        self.performBinaryOp(lambda x, y: x + y)    # replaced with a lambda expression
```

Using stack to build a calculator (version 3)

```
class CalculatorEngine(object):
          :
    def doAddition (self):
        self.performBinaryOp(lambda x, y: x + y)

    def doSubtraction (self):
        self.performBinaryOp(lambda x, y: x - y)

    def doMultiplication (self):
        self.performBinaryOp(lambda x, y: x * y)

    def doDivision (self):
        self.performBinaryOp(lambda x, y: x / y)
```

Appendix

Iterators

- for statement loops over many data structures
```
>>> for i in [1 , 2, 3, 4]:
        print (i)
1
2
3
4
```
-
```
>>> for c in "CSCI":
        print (c)
C
S
C
I
```

Iterators

- When use with dictionary, it loop over its keys
```
>>> for k in {"x": 1, "y": 3}:
            print (k)
x
y
```
- All these are iterable objects
- Built-in function *iter* takes an iterable object and returns an iterator

```
>>> x = iter([6, 8, 9])

>>> x

<listiterator object at 0x1E547774>
```

Iterators

```
>>> x
<listiterator object at 0x1E547774>
>>> x.__next__()
6
>>> next(x)
8
>>> x.__next__()
9
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterators

- Iterators are implemented as classes
- We can implement our own iterator

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n
    def __iter__(self):      # this makes an object iterable
        return self
    def __next__(self):          # this method implement element extraction
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()   # exception when no more data
```

Iterators

- Returned iterator will then return next element through next() method

```
>>> y = yrange(3)
>>> next(y)
0
>>> y.__next__()
1
>>> y.__next__()
2
>>> next(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in next
StopIteration
```

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