Functional Programming

One of the most popular programming paradigms is FP. Number of programming languages support OOP. In this lecture, we show how we can use Python to carry out some basic FP.

Created by John C. S. Lui on May 31, 2018.

Important note: *If you want to use and modify this notebook file, please acknowledge the author.*

Functional programming

- Treat input as "set"
- Define operations we want to do for each element of the set

In functional programming, there are two important concepts:

```
- iterator
- passing function as parameter to another function
```

So let us go over **iterator** first.

In [ ]:
```
## Iterators: list
for x in [1,2,3,4,5]:
    print("Element is: ", x)
```

In [ ]:
```
## Iteators: character string
for c in "John is a jerk!!!":
   print ("Char. is: ", c)
```

In [ ]:
```
# Iterators: tuple
for x in (1,2,3,4,5):
    print ('x is:', x)
```

In [ ]:
```
# Iterators: dictionary

# When we use with dictionary, it loops over the keys of the dictionary
for key in {"k1": 'value1', "k2": 'value2', "k3": 15}:
    print ("key is: ", key)
```

Iterators

- list, character strings, tuple and dictionary are all **iterable objects**.
- A function *func* takes an iterable object and returns an iterator.

In [ ]:
```
iter_obj = iter([1,2,3,4,5])   #iter_obj is an iterable object
iter_obj

# iter_obj.next()   # display next item
print(next(iter_obj), next(iter_obj), next(iter_obj))
print(next(iter_obj),next(iter_obj))

# can't access the next item
# next(iter_obj)
```

Python Function

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

In [ ]:
```
import math

mySqrt = math.sqrt  # assign my math.sqrt function to the variable mySqrt
mySqrt(4.0)
```

- If a variable can be a reference to a function
- And we can pass variable as a parameter to a function
- This means we can also pass a function (or variable) as a parameter to another function !!!

For detail, please the document: http://python-history.blogspot.com/2009/02/first-class-everything.html (http://python-history.blogspot.com/2009/02/first-class-everything.html)

Why is this **important**? Because in functional programming, we need the concept of:

- iterator
- passing function as parameter

Transformation

Often, our variables (or data) are represented using list, tuple or dictionary.
We may want to carry out some *transformation* for all or a subet of elements in the variable.
This can be achieved via

- mapping
- filtering
- reduction (for Python 2.x)

Let's illustrate the *map()* function first.

The general syntax is:

    % *map(func, seq)*

where *func* is a pre-defined function which will operate on each element of the *seq* sequence (e.g., list).

Note that in Python 2.x, map returns a list, where each element of the result list was the result
of the function *func* applied on the corresponding element of the list or tuple *seq*.
With Python 3, *map()* returns an ***iterator*** .

In [ ]:
```
# we have a list of integer and we want to multiply each elements in the list by 2 and add 1 to it.

# First, define a function
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 = list (map(map_func1, range(0,11)))
print("my_variable is now : ", my_variable)

my_variable = [ x+1 for x in map(map_func1, range(0,11)) ]   # using list compreshension
print("my_variable is really :", my_variable)
```

You can put **more than one input (or sequence)** as arguments.

In [ ]:
```
# 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)
```

Let's see how *filter()* function works.

The general syntax is:

% filter (func, seq)

where each element in the *seq* will be passed to the *func* to check the boolean condition. If it is satisfied, the element will be returned. Hence, all elements will be returned as ***iterator*** .

In [ ]:
```
# define a filter function
def my_func1(x): return x % 3 == 0 or x % 5 == 0    # return True or False

def my_func2(x): return (x+1)%2 == 0                # return True or False

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(1,25))]  # use list comprehension !!!!
print("2. my_variable = ", my_variable)
```

Let's see how *reduce()* function works in Python 2.x.

The general syntax is:

% *reduce (func, seq)*

returns a single value constructed by calling the *func* function on the
first two items of the *seq*, then the result and the next item of *seq*
will again be applied to the *func* function, and so on. Let's see an example (in Python 2.x):

```
>>> def add(x,y): return x+y
>>> reduce (add, range(1,11))     # this will return 55 in Python 2.x
```

For Python 3.x, we have to first *import* the functools package of reduce:

In [ ]:
```
from functools import reduce

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

reduce(my_add, range(1,11))
```

Lambda function

The lambda operator or lambda function is a way to create small *anonymous* functions, i.e. functions without a name.

The general syntax of a *lambda* function is quite simple:

>>> lambda argument_list: expression

The argument list consists of a comma separated list of arguments, and the expression is
an arithmetic expression using these arguments. One can assign the function to a variable to give it a name.

Let's illustrate.

In [ ]:
```
# define a function
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))
```

Combining lambda and map/filter function

Let's see how can use lambda with other map/filter functions

In [ ]:
```
# use of lambda function
my_variable1 = list(map(lambda x,y: x+y, range(0,10), range(0,10)))
print ("my_variable1 = ", my_variable1)

# use lambda and list comprehension
my_variable2 = [x for x in map(lambda x,y: x+y, range(0,10), range(0,10))]
print("my_variable2 = ", my_variable2)
```

List comprehension

General syntax:

[*expression* **for** *var* **in** *iterator or list* **if** expression]

- **if** part is optional
- Each element in the list (or iterator) is examined by the **if** statement
- If elemnet passes **if** check, then *expression* is evaluated and result will be added to the list (or iterator)

Let's illustrate.

In [ ]:
```
# define a list
a_list = [1,2,3,4,5]
print("The result is: ", [x*2 for x in a_list if x<4])
```

In [ ]:
```
# define a dictionary
my_dictionary = {1:'john', 2:'jack', 3:'alice', 4:'helen'}

a_list = [my_dictionary[i].title() for i in my_dictionary.keys()]
print('a_list = ', a_list)

a_list = [my_dictionary[i].title() for i in my_dictionary.keys() if i %2 == 0]
print('a_list = ', a_list)
```

In [ ]:
```

```