CS 133 - Introduction to Computational and Data Science

Instructor: Renzhi Cao
Computer Science Department
Pacific Lutheran University
Spring 2017

Introduction to Python II

- Quiz 1
  - Ave 17.21
  - Max 20

People by frequency	
Warren Buffett	2
Tom Brady	2
Barack/Michelle Obama	2
Aberaham Lincoln	1

Food by frequency	
Steak	2
Nothing	2
Pasta	1
Seafood	2

- In the previous class, you have learned string and list. Today we are going to learn tuples, dictionary and function!

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (“a”,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>> z = [1,2,3]
>>> z[0] = 1
>>> x[0] = 1
```

Dictionaries

- A set of key-value pairs. Like a list, but indices don't have to be a sequence of integers.
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries

- The function dict() creates a new dictionary with no items

```
>>> newDic = dict()
```

- Use [] to initialize new items

```
>>> newDic[‘one’] = ‘Hello’
>>> newDic = {‘one’:’Hello’,  ‘two’:’Great’,
‘3’:’CS133’}
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Functions

- Functions are “magic boxes” that will return values based on the input. There is an endless number of functions already created for you. Some examples:
- int(’32’) float(22) str(21)

Not all functions are included by default. You need to call the module that include them. To do that, you need to type the word import followed by the name of the module.

- **import math**
- You can rename the module by using
- **import math as m**

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

functionbasics.py

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Adding new functions

Order is important!!!

- Always declare your function **before** you try to use it

- Functions can be of two types:
- void
- Non-void

- Void functions are just like the functions we just created: They don't return any value.

```
def test(n,m,r):
    sol = n + m + r
    print sol
```

- This type of function usually **shows** the result internally

Non-void functions

A non-void function **returns** a value to the caller.

- This is very important since the function might just calculate one value of the “main” calculation
- We need to use the word return at the end of the function

```
def test(x,n,m):
        sol = x + n + m
        return sol
```

sol is a value that now is available to be used later.

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

funcasparam.py

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

funcinfunc.py

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

funcreturnfunc.py

```
~: python funcreturnfunc.py
<function bar at 0x612b0>
5
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Practices

1. Create multiple void functions that:
    1. Print the word “Hello” 3 times
    2. Print the word “Hello name!” in which name is replaced by an input given by the user. Example: If input is Cao, it will print “Hello Cao!”
    3. Calculate the multiplication of the 3 inputs received by this function and print the result
2. Create multiple non-void functions that:
    1. Return the word “Hello” 3 times
    2. Return the word “Hello name!” in which name is replaced by an input given by the user. Example: If input is Cao, it will return “Hello Cao!”
    3. Calculate the multiplication of the 3 inputs received by this function and return the result

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Control flow

Things that are False

- The boolean value False
- The numbers 0 (integer), 0.0 (float) and 0j (complex).
- The empty string "".
- The empty list [], empty dictionary {} and empty set set().

Things that are True

- The boolean value True
- All non-zero numbers.
- Any string containing at least one character.
- A non-empty data structure.

Control flow

There are cases that you want specific block of code to be functional when some condition is true.

- User type ‘yes’, do calculation, type ‘no’, quit program
- When temperature is higher than 100 degree, print ‘hot’.
- When your bank account has 0 balance, user cannot withdraw any money.

If statement

The code we have seen before is “always” executed. How would we create cases in which only some code is executed?

- **if expression:** # expression is boolean type
  do something when expression is True
  [else:] # this is optional

If statement

```
>>> smiles = "BrC1=CC=C(C=C1)NN.Cl"
>>> bool(smiles)
True
>>> not bool(smiles)
False
>>> if not smiles:
...       print "The SMILES string is empty"
...
```

The “else” case is always optional

If statement

```
>if x% 2 = = 0:
        print 'x is even'
else:
        print 'x is odd‘
```

What is the % doing here?

If statement

```
>if x = = y:
        print 'x and y are equal'
else:
        if x < y:
                print 'x is less than y'
        else:
                print 'x is greater than y‘
```

Observe the use of indentation

“elif”

```
>>> mode = "absolute"
>>> if mode == "canonical":
...                     smiles = "canonical"
...     elif mode == "isomeric":
...                     smiles = "isomeric”
...             elif mode == "absolute":
...                     smiles = "absolute"
...     else:
...                     raise TypeError("unknown mode")
...
>>> smiles
' absolute '
>>>
```

“raise” is the Python way to raise exceptions

Boolean logic

Python expressions can have “and”, “or”:

```
if(a <= 10 and b >= 10 or a == 100 and b!= 5):
    print “Hello”


if( 3 <= a <= 100):
     print “great!”
```

Practices

1. Get user's score, save it as variable score.
2. print 'A' for score in [90,100], 'B' for [80,90), 'C' for [70,80), 'D' for rest of scores.

After class

1. Practice and get familiar with Atom, command prompt
2. Try examples using python, such as string, list, tuples, if statement.

Additional functions as reference

Higher-Order Functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

Higher-Order Functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

Higher-Order Functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

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

```
>>> from highorder import *
>>> lst = ['h','e','l','l','o']
>>> reduce(plus,lst)
'hello'
```

highorder.py