CS 133 - Introduction to Computational and Data Science

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

Introduction to Python II

- In the previous class, you have learned string and list.

- Question left: How to print backward of a string?

In python, we use the following to get subset of a string:

s[start:end:step], by default step is 1, and it’s optional.

```
>> s = “abcdef\n”
>> s[0:5:2]
>> s[-1:-4:-1]
>> s[-1:-4]
>> s[:]                # use default start end index and step
```

Introduction to Python II

- Today we are going to learn Lists, and tuples, dictionaries, and functions.

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Content

- The method **append** modifies the list and returns **None**
- List addition (+) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Lists: examples

```
>[10,20,30,40]
>['spam', 20.0, 5, [10,20]]
>cheeses = [' Cheddar', 'Edam', 'Gouda']
>numbers=[17,123]
```

Traverse a list

```
>for cheese in cheeses:
        print cheese
>for i in range( len( numbers)):
        numbers[ i] = numbers[ i] * 2
> numbers.extend([1,2,3])        # another way to append elements
```

Lists: examples

Delete element

```
>t = [' a', 'b', 'c']
>x = t.pop( 1)
```

OR

```
>del t[ 1]
```

OR

```
>t.remove(' b‘)
```

Lists: Practice

1. Create CS133_Lists.py using Atom
2. Create String type ‘str’, the value is “CS133”
3. Assign 2017 to a variable ‘year’
4. Create a List type ‘newList’, and assign variable ‘year’ to it
5. Add ‘str’ to the ‘newList’
6. Add first two characters of ‘year’ to the end of ‘newList’
7. Delete first element in ‘newList’
8. Append [1,2,3] to ‘newList’, and print out ‘newList’ and it’s length

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
```

functionbasics.py

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

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
```

funcasparam.py

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

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)
```

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

funcreturnfunc.py

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 print “Hello Cao!”
   3. Calculate the multiplication of the 3 inputs received by this function and return the result