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 how to create a python script, get input from user, object type of number and strings.

  - What tool we used to edit Python code?
  - How to run python code?
  - Is “Print” a valid variable name?
  - Is “int” a valid variable name?
  - Is “Int” a valid variable name?

Introduction to Python II

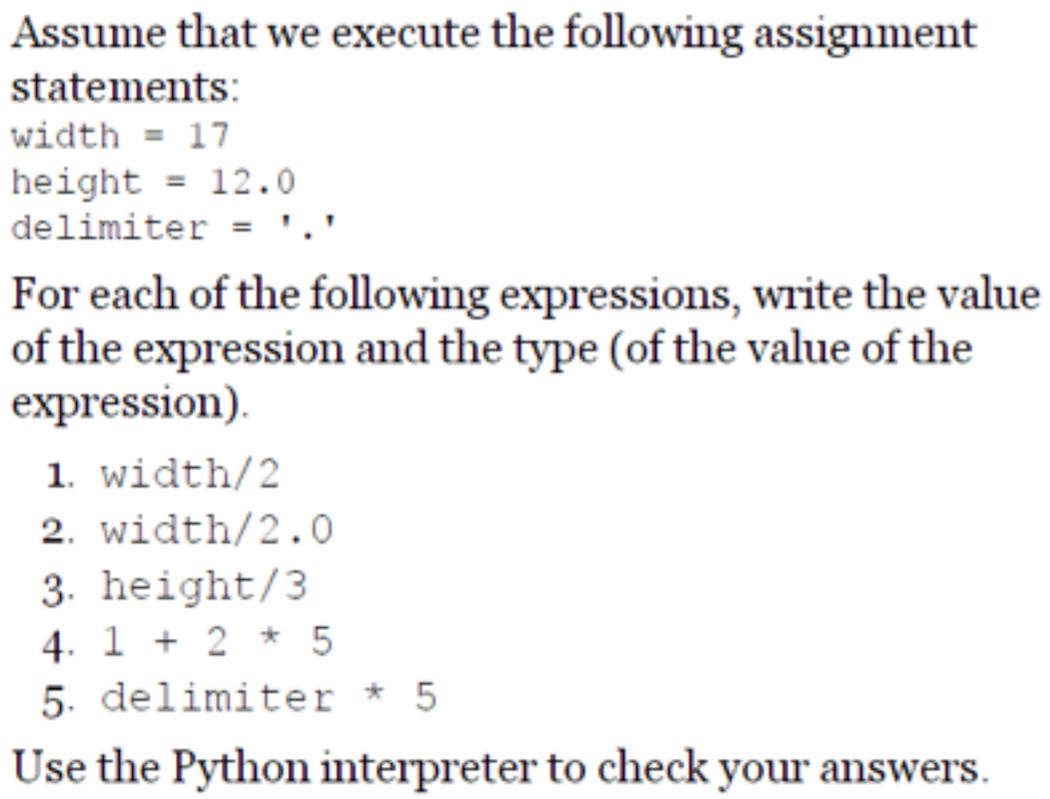
- Today we are going to learn String, Lists, and tuples, dictionaries, and functions.

Tracing variable's value

- `>>> x = 1.5`
- `>>> y = x`
- `>>> y = x + 1.5`
- `>>> x`
- `>>> y`

Exercises

- First test your program from command prompt
- Use **Atom** (text editor) to create python script - test.py, and run it



Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
delimiter = '.'
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`
5. `delimiter * 5`

Use the Python interpreter to check your answers.

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Exercises

- Try to use **raw_input** to get a score from the user, multiply it by 10, and print out the result.

Strings

- Record both textual information (your name as example) and arbitrary collections of bytes (such as image file’s contents)

- Strings are sequences of characters.

Strings

- Strings are *immutable*
- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)‘
>>> print s
And me too!
though I am much longer
than the others :)
```

Substrings and Methods

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Methods: find , split

```
smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles.find("(O)")
15
>>> smiles.find(".")
9
>>> smiles.find(".", 10)
-1
>>> smiles.split(".")
['C(=N)(N)N', 'C(=O)(O)O']
>>>
```

Use “find” to find the start of a substring.

Start looking at position 10.

Find returns -1 if it couldn’t find a match.

Split the string into parts with “.” as the delimiter

String operators: in, not in

```
if "Br" in “Brother”:
	print "contains brother“

email_address = “clin”
if "@" not in email_address:
	email_address += "@brandeis.edu“
```

String Formatting

- Similar to C’s printf (%s for string, %d for integer).
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Strings

```
>fruit = ‘banana’
>letter = fruit[1]
>len(fruit)
>fruit[-1]
>fruit[-2]
```

Traverse a string

```
>for char in fruit:
        print char
>r= fruit[0:2]
```

Strings

```
>fruit = ‘banana’
>fruit[:]               # all of fruit as a top-level copy (0:len(fruit))


> fruit + ‘xyz’         # Concatenation

> fruit * 8             # Repetition


> fruit[0] = ‘a’        # immutable objects cannot be changed

> new = ‘a’ + fruit[1:]       # this is fine
```

Strings

Strings have methods:

```
>word= “banana”
>word.find(‘a’) or word.upper() or word.replace(‘a’,’b’) or word.split(‘,’)

> S = ‘aaa,bbb,ccc, dd\n’
> S.rstrip()       # remove whitespace characters on the right side

>dir(S)            # help
```

Exercise

Create a script that:

1. Create a string with any characters in total length of 10. (you can manually assign it or asks the user - Raw_input method)
2. Prints the string letter by letter. Each letter in a different line
3. Prints the string in lower case
4. Prints the string in upper case
5. Prints the string backwards
6. Create string with “,” inside, and use split method to process it
7. Prints first three characters
8. Prints last four characters

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 = (1,2,3)
>>> 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)
```

```
~: 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

After class

1. Practice and get familiar with Atom, command prompt
2. Try examples using python, such as Integer, Strings

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)
```

highorder.py

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