# Demonstrate Understanding - First 2 weeks

1. I always forget the principle that the slice operation [n:m] does not include the m since “[]”refers to a closed set in math. After doing some exercises and making several mistakes, I can remember such principle now. However, such experience makes me think the slice operation returns the element instead of a list when there is only one element.

e.g.

```
a_list = ['a', 'b', 'c', 'd', 'e', 'f']
print a_list[1:2]
```

I think the outcome is a string. But I know it is a list after printing “type(a_list[1:2])” and see “<type 'list'>” on the console.

2. I thought the function “int()” is useless until I use the function “raw_input()” to solve the problem 4 in exercise of Simple Python Data chapter.

At first, I wrote like this :

```
now=raw_input("now")
wait=raw_input("wait")
then=(now+wait)%24
print then
```

The outcome is strange. And I then remembered that the function “raw_input()” returns only strings. I know the importance of noticing the type of values of every function and understand the meaning of function “int()”.

3. The format of turtle module

```
import turtle   #import turtle module
wn=turtle.Screen()   #create an instance named wn of Screen class
alex=turtle.Turtle()    #create an instance named alex of Turtle class
……
wn.exitonclick()        #close the turtle window when user clicks
```

Every instance has attributes and methods.
At first, I thought “wn.bgcolor” is attributes since the background color is the attribute from a human perspective. When I see this explanation: “Some of the methods of the Turtle class set attributes that affect the actions of other methods” in the textbook, I know it is a method. And I try to understand the essential difference between attribute and method. Attributes should be assigned value just like other variables and methods work like functions having input and output.

4. Accumulator--The order of variables matters.
e.g.  enter: abcdefghijk

| | |
|---|---|
| s=raw_input("Please enter")<br>reverse=""<br>for c in s:<br>  reverse=c+reverse<br>print reverse | s=raw_input("Please enter")<br>reverse=""<br>for c in s:<br>  reverse+=c<br>print reverse |
| kjihgfedcba | abcdefghijk |

9.24

The first question in the Exercise part of Dictionary helps me figure out basic methods of dictionary.

```
d = {'apples': 15, 'grapes': 12, 'bananas': 35}
print d['bananas']    #add new pair
d['oranges'] = 20    #add new pair oranges-20
print d
print len(d)          #4
print 'grapes' in d    #True
#print d['pears']       #error
print d.get('pears', 0)     #0
fruits = d.keys()       #['apples','grapes','bananas',  'oranges']
print fruits
fruits.sort()           #['apples','bananas','grapes','oranges']
print fruits
del d['apples']
print 'apples' in d      #False
```

At first, I was not sure what the “d['oranges'] = 20” returns. I thought it would cause an error because I mixed it up with “print d['oranges']” which returns False. In fact, using “d['oranges'] = 20” to add a new pair to a dictionary is the first thing I learned in the Dictionary chapter, but because of professor’s emphasizing “print d['oranges']” in the class, I forgot it. I need to review more and remember the syntax.

```
35
{'apples': 15, 'grapes': 12, 'bananas': 35, 'oranges': 20}
4
True
0
['apples', 'grapes', 'bananas', 'oranges']
['apples', 'bananas', 'grapes', 'oranges']
False
```

pirate = {}

```
pirate['sir'] = 'matey'
pirate['hotel'] = 'fleabag inn'
pirate['student'] = 'swabbie'
pirate['boy'] = 'matey'
pirate['restaurant'] = 'galley'
#and so on

sentence = input("Please enter a sentence in English")
presence=[]
words=sentence.split()
for word in words:
    if word in pirate:
        presence.append(pirate[word])
    else:
        presence.append(word)
print " ".join(presence)
```

10.2

The difference between return and print in the function definition. The way I use to distinguish them is remembering “print is for people to see” while “return is for the program to use”. Print statement is just like any other statement in the function’ code block. The expression following “return” is the real output of the function from the program’s perspective.

e.g.

```
def square(x):
    print x
    return y*y
z = square(10)
```

The function square aims to calculate the exponation of the input, so what it returns is x*x.

The print statement “print x” here is to print out x on the screen so people can clearly know what number the input is. It is what is returned(y*y) that can participate in the following part of the program.

So what people can see is 10 for the code above. Because the last line invokes the function square.

So if we add a line: print z

```
def square(x):
    print x
    return y*y
z = square(10)
print z
```

Then we now can see 10 and 100 being printed out. Because the last line invokes the function and at the same time, print the value it returns.

10.9
For the function which has optional parameters.
The optional parameters must be put behind required parameters. Otherwise we will get a SyntaxError.
e.g:

```
  def greeting(greeting = "Hello ", name, excl = "!"):
    return greeting + name + excl
```

The parameter name should be put at the first.

```
  def greeting(name, greeting = "Hello ", excl = "!"):
    return greeting + name + excl
```

The local parameter is assigned a value during definition.

```
initial = 7
def f(x, y = 3, z = initial):
   print "x, y, z are:", x, y, z
initial = 0
f(2)
```

For example, z is assigned the value of variable initial 7 which can only be override during execution. So even the value of variable initial changes to 0, z' default value is still 7. So f(2) returns 2,3,7.

10.16
Sorting a dictionary.
There are 3 ways to sort a dictionary.
e.g d={0: 2, 1: 2, 2: 1, 3: 4, 4: 2, 5: 1, 8: 2}

Sorting by items.
y = sorted (d.items(), key = lambda x: x[1], reverse=True)
The value of d.items is a list of 8 tuples: [(0, 2), (1, 2), (2, 1), (3, 4), (4, 2), (5, 1), (8, 2)]. The input of lambda function is a tuple.
If we print y, we will get a list of tuples: [(3, 4), (8, 2), (4, 2), (1, 2), (0, 2), (5, 1), (2, 1)]

Sorting by keys.
y = sorted (d.keys(), key=lambda k: d[k], reverse=True)
The value of d.keys() is a list: [0, 1, 2, 3, 4, 5, 8]. The input of lambda function is an integer.
If we print y, we will get a list: [3, 8, 4, 1, 0, 5, 2]

Sorting by the dictionary.
y=sorted (d, key=lambda k: d[k], reverse=True)
This method is the same as sorting by keys because when passing a dictionary, in fact, the keys are passed. So if we print y, we will get a list of keys [3, 8, 4, 1, 0, 5, 2] instead of a dictionary.

10.29
json turns a response object into a python object.
request.get yield Unicode datatype.
We will see pretty output if we print a Unicode string.
But we will see ‘u’ before a string if we deal with python object that saves Unicode string.
We can use encode(‘utf8’) to get rid of those ‘u’s.

e.g
import requests
import json
result = requests.get("https://services.faa.gov/airport/status/DTW", params ={'format':'json'})
d = json.loads(result.text)

| code | outcome |
|---|---|
| print type(result) | <class 'requests.models.Response'> |
| print type(result.text) | <type ‘unicode’> |
| print type(d) | <type ‘dict’> |
| | |
| print result.text | {"delay":"false","IATA":"DTW","state":"Michigan","name":"Detroit Metropolitan....."type":""}} |
| print d | {u’status’: {u ’minDelay’:u}.....u ’visibility’: 8.0}} |
| | |
| print d['city'], d['weather']['temp'] | Detroit 65.0 F (18.3 C) |
| | |
| print (d['city'], d['weather']['temp']) | (u’ Detroit’, u ’65.0 F (18.3 C)’) |

| | |
|---|---|
| print (d['city'].encode('utf8'),<br>d['weather']<br>['temp'].encode('utf8')) | (' Detroit', '65.0 F (18.3 C)') |

11.6

## Class variable and instance variable

Class variable is a part of class definition and method definitions are class variables.
Interpreter first checks the variable name in instance variables. If it doesn't find an instance variable, it will then check whether the class has a class variable.
e.g.

```
class Point():
    printed_rep = "*"
    def __init__(self, initX, initY):
        self.x = initX
        self.y = initY
        self.printed_rep='**'
p1 = Point(2, 3)
p2 = Point(3, 12)
p2.printed_rep='***'
print p1.printed_rep
print p2.printed_rep
```

```
**
***
```

The example above shows the interpret find the variable "print_rep" in instance variables instead of in class variables.

Also, an assignment statement for a class variable in a specific instance never sets the class variable, it only sets the instance variable.

e.g.

| | |
|---|---|
| class Point():<br>  printed_rep = "*"<br>  def __init__(self, initX, initY):<br>  self.x = initX<br>  self.y = initY<br>p1 = Point(2, 3)<br>p2 = Point(3, 12)<br>p2.printed_rep='***'<br>print p1.printed_rep | * |

The “p1.printed_rep” prints out “*” after p2 sets its “printed_rep” to “***”

11.12

List comprehension is a more concise way to do for-loop. And the “for……in……” clauses inside a list comprehension need to go in the same order as if they were normal for-loops.

*e.g. Q7 in Extra Exercise*
*The nested for loop given takes in a list of lists and combines the elements into a single list. Do the same thing using a list comprehension for the list L. Assign it to the variable result2.*

| |
|---|
| L = [["hi", "bye"], ["hello", "goodbye"], ["hola", "adios", "bonjour", "au revoir"]] |
| Normal for-loop:<br>def onelist(lst):<br>  result = []<br>  for each_list in lst:<br>  for item in each_list:<br>  result.append(item)<br>  return result |
| List comprehension:<br>result2=[item for each_list in L for item in each_list] |

As showed above, the order of nested for-loop is the same, or say or logic is same. It is the syntax is different. It is a good way to write in normal for-loop format when I am not used to using list comprehension.

11.20

## Cache

e.g.Line 104-118 in sample_twitter.py

```
try:
f = open("creds.txt", 'r')
(client_key, client_secret, resource_owner_key, resource_owner_secret, verifier) = json.loads(f.read())
f.close()
except:
tokens = get_tokens()
f = open("creds.txt", 'w')
f.write(json.dumps(tokens))
f.close()
    (client_key, client_secret, resource_owner_key, resource_owner_secret, verifier) = tokens
```

The sequence of “get cache data” and “save data in cache file” in try-except code block is kind of counterintuitive when the program is run for the first time. But it is reasonable if the program is run for several times – check the cache file first and decide if it is necessary to get data from website.

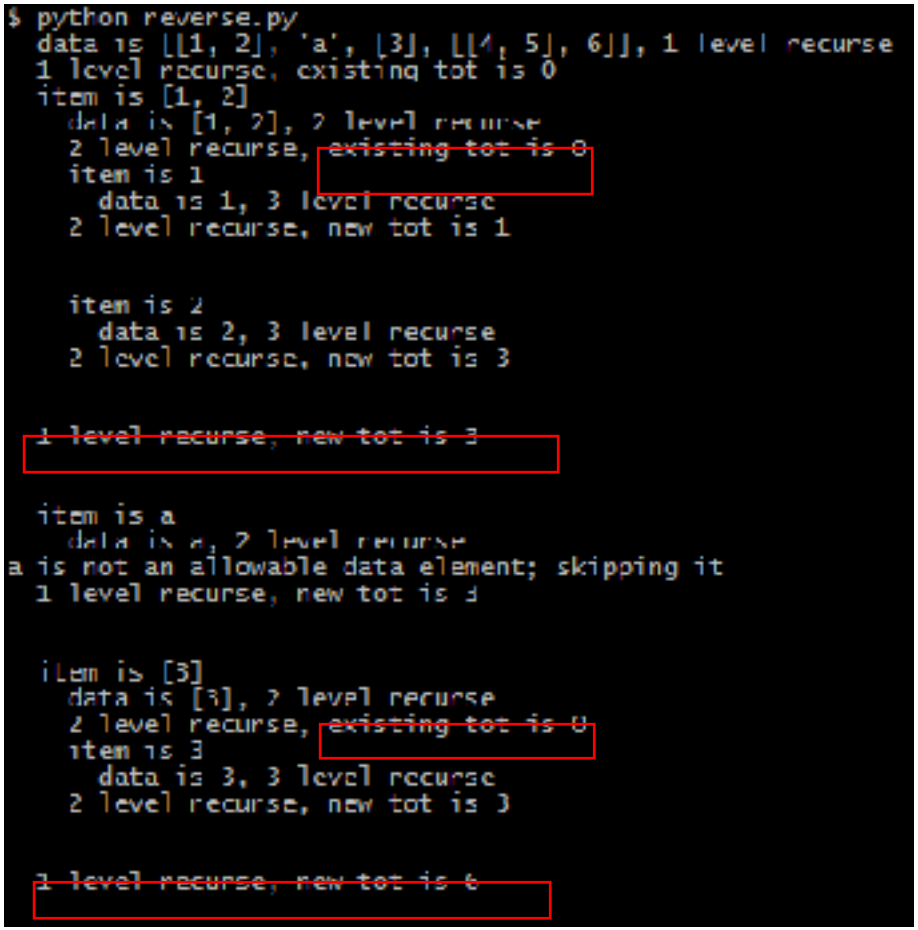
When reading data from a file to the program, f.read() returns ‘string’ object, so it is necessary to use ‘json.loads()’ to turn the ‘string’ object into the original python object. Here it is ‘tuple’.
When writing data into a file, ‘json.dumps()’ is used to turn python object (here is tuple) into ‘string’ so that it can be saved in a file.

12. 4
Recursion
Recursion is a method keeps calling itself.

```
def sum_recursive(data, recursion_level = 1):
    # data can contain integers, or other lists
    # return sum all the integers, no matter how deeply nested they are
    print '  '*recursion_level+'data is {}, {} level recurse '.format(data,recursion_level)
    #print data

    if type(data) == type([]):
        tot = 0
        print '  '*recursion_level+'{} level recurse, existing tot is {}'.format(recursion_level,tot)
        for item in data:
            print '  '*recursion_level+'item is {}'.format(item)
            tot = tot + sum_recursive(item, recursion_level + 1)
            print '  '*recursion_level+'{} level recurse, new tot is {}'.format(recursion_level,tot)
            print '\n'
        return tot
    elif type(data) == type(1):
        return data
    else:
        print "{} is not an allowable data element; skipping it".format(data)
        return 0

my_lst = [[1, 2], "a", [3], [[4, 5], 6]]
sum_recursive(my_lst) # should be 55
```




So ‘tot’ is set to 0 when there is a sub list in the list. To sum up all integers in the list, we iterate through all the items. For those sub lists, we first go into the deepest item and the add every deepest item in the deepest sub list to the ‘tot’ for this specific sub list, and then repeat such process till we go back to 1 level of list where the all the ‘tot’ add up to the final ‘tot’.