CMP4266: Software Development UG1
Lab Session 3: Lists, Tuples, Repetition Structure

(Based on the Lab sheet prepared by Richard Kay)

1. Objectives

a. Experiment with the range function to generate lists.
b. Experiment with list operations, creating, inserting, appending and slicing lists.
c. Process every item in a list using a for loop.
d. Use while loops controlled by Boolean expressions.
e. Carry out simple programming exercises involving functions, lists and loops. More advanced students have the opportunity to carry out a more difficult programming exercise involving prime numbers.

Note: Although most of the instructions in this sheet are written for command line interpreter, you are advised to save your python codes as scripts. To do that, go to your Z: drive and under the folder CMP4266 that you created in Lab-1 session create a folder called Lab3. You should save all your today's Python scripts under this Z:\CMP4266\Lab3 folder.

2. Tuples, Collection Types and Lists

2.1 Tuples, type evaluation and comparison

We've used tuples as an ordered collection of items which can't be changed e.g. (10.5,"hello") . These are usually surrounded by round () brackets, and items are separated using commas (,). We can also have empty tuples, or tuples with just one item, but in that case if we put a comma after the item we'll avoid confusing the interpreter. If we want to know, we can find out how an object's type will be evaluated, by passing any object as a parameter to the type() function:

```
>>> type(("a",10.5))
<class 'tuple'>
>>> type((0))
<class 'int'>
>>> type((0,))
<class 'tuple'>
>>> type(())
<class 'tuple'>
>>> type("hello")
<class 'str'>
>>> type(10.5)
<class 'float'>
```

In Python it sometimes makes sense to compare types for equality:

```
>>> type(math.pi) == type(10.5)
True
```

```
>>> type(10.5) == type("hello")
False
```

So applying what we learned last week, we can then branch code within a function, based upon the type of a function parameter.

2.2 Student exercise

Develop a Python function which either returns the float square of its parameter x if the parameter is a number, or prints the string "Sorry Dave, I'm afraid I can't do that" if the parameter is a string, and then returns 0.0.

2.3 Lists

A list is an ordered collection of objects which we can change, surrounded by square brackets.

```
>>> type([])
<class 'list'>
>>> type([2,4])
<class 'list'>
>>> type(['hello', 10.5])
<class 'list'>
```

A list can be empty too. Sometimes we can construct a list using a loop, starting with the empty list: []. Mostly we'll want lists where all the items are of the same type, but they don't have to be.

2.4 Manipulating a list

```
>>> l=[2,3]
>>> l.append(5)
>>> l
[2, 3, 5]
```

Append method inserts an item at the end of a list.

```
>>> l.insert(0,1)
>>> l
[1, 2, 3, 5]
>>> l.insert(2,4)
>>> l
[1, 2, 4, 3, 5]
```

The insert method inserts its 2nd parameter at the index position given by the first parameter. Index positions start counting at 0, so a list with 5 items will have indices in the range 0 to 4 inclusive.

```
>>> l[1]
2
>>> l[4]
```

```
5
```

2.5 Measuring length, sorting and reversing a list

We can get the length of any sequence object (e.g. list or tuple) using the len() function:

```
>>> len(l)
5
>>> len("this")
4
>>> len((0,))
1
```

Lists can be reversed and sorted. These operations don't return the changed list, but they do change the order of the items within it.

```
>>> l.reverse()
>>> l
[5, 3, 4, 2, 1]
>>> l.sort()
>>> l
[1, 2, 3, 4, 5]
```

The del command removes the indexed item, and shifts higher indices down by one.

```
>>> del l[1]
>>> l
[1, 3, 4, 5]
>>> del l[3]
>>> l
[1, 3, 4]
```

2.6 Generating a list using the list() and range() function

```
>>> list(range(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The first parameter to range() is the inclusive start of the range. The second parameter is the exclusive end of the range. We can subtract the start parameter from the end parameter to get the length.

We can go up in 2's or 3's, or any gap or increment using a 3rd parameter.

```
>>> list(range(1,20,2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> list(range(3,20,3))
[3, 6, 9, 12, 15, 18]
```

Sometimes we want to count down so we use a negative increment.

```
>>>list( range(20,0,-1))
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

2.7 Slicing a list

```
>>> a=list(range(1,11))
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[3:5]
[4, 5]
>>> a[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[:6]
[1, 2, 3, 4, 5, 6]
>>> a[6:]
[7, 8, 9, 10]
```

If you do these experiments, you'll understand the rest of this explanation better. We can obtain a slice from the list, or a part of it, by using the slice operator. This is a pair of square brackets, with a colon, with indices before and after the colon, the first being inclusive and the second exclusive. If a start or end index isn't present, the start or end of the slice will be at the start or end of the list. A slice of part or of the whole list will take a copy of the list, and make this into a new list. Change the old list, and the cloned copy won't be changed. However, if you assign a list to another reference variable, these 2 references will still point to the same list.

```
>>> b=a
>>> c=a[:]
>>> a.remove(3)
>>> a
[1, 2, 4, 5, 6, 7, 8, 9, 10]
>>> b
[1, 2, 4, 5, 6, 7, 8, 9, 10]
>>> c
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

3. Repeating a block of statements using a for loop.

The following example defines a function using a for loop which prints a times table. The times table to be printed (e.g. 7) is passed as a parameter when we call the function. Copy and save the following code into a python file. Name the file times_table.py

```
def timestable(x):
        for i in range(1, 13):
                print("%d times %d is %d" %(i, x, i*x))


timestable(7) # calling  timestable while providing a parameter value 7
```

The output will look like the following:

```
1 times 7 is 7
2 times 7 is 14
3 times 7 is 21
4 times 7 is 28
5 times 7 is 35
6 times 7 is 42
7 times 7 is 49
8 times 7 is 56
9 times 7 is 63
10 times 7 is 70
11 times 7 is 77
12 times 7 is 84
```

3.1 Some Student Exercises

1. Develop a Python program which prompts the user to input a number between 1 and 12, and which also calls this function to print out the times table requested.

2. Adapt this software to print out all 12 times tables, each table on the same line. Note, the input number will be provided by a for-loop ranged between 1 - 13. To be able to print the results on the same line, the print statement will need to have *end*=" " as the last argument like this:

   ```
   for i in range(1, 13):
       print(i, end=" ")
   ```

   ```
   1 2 3 4 5 6 7 8 9 10 11 12
   ```

   You won't get a new line after each item to be printed. To output just a new line, at the end of each line output in your table, you could use the print command on its own. When you have completed your program the output should look like this:

   ```
     1   2   3   4   5   6   7   8   9  10  11  12
     2   4   6   8  10  12  14  16  18  20  22  24
     3   6   9  12  15  18  21  24  27  30  33  36
     4   8  12  16  20  24  28  32  36  40  44  48
     5  10  15  20  25  30  35  40  45  50  55  60
     6  12  18  24  30  36  42  48  54  60  66  72
     7  14  21  28  35  42  49  56  63  70  77  84
     8  16  24  32  40  48  56  64  72  80  88  96
     9  18  27  36  45  54  63  72  81  90  99 108
    10  20  30  40  50  60  70  80  90 100 110 120
    11  22  33  44  55  66  77  88  99 110 121 132
    12  24  36  48  60  72  84  96 108 120 132 144
   ```

You can output an integer x right justified within a field of fixed width 4 using "%4d" % x

4. Using a while loop controlled by a Boolean condition

The for loop is ideal for processing data which comes within a range of values which can be computed before we start the loop. It can also be used whenever we have a list of objects and need to run the same code on each item in the list in turn. A while loop uses a Boolean condition to continue or stop the loop.

4.1 Can a loop be infinite?

Sometimes, we can only stop the loop when we encounter some condition inside the loop which enables us to know that the job the loop does is completed. A loop which hasn't got a way of stopping itself will repeat forever, or until the program is interrupted or the machine shuts down.

We can interrupt a program in an endless loop on the Python command line by pressing <CTRL> and <c> keys together e.g. :

```
>>> while(True):
        print(1, end=" ")


1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1Traceback (most recent call last):
  File "<pyshell#5>", line 2, in <module>
    print(1, end=" ")
  File "C:\Python34\lib\idlelib\PyShell.py", line 1342, in write
    return self.shell.write(s, self.tags)
KeyboardInterrupt
>>>
```

If we have a Python program saved as a file and running which accidentally suffers the same fault, we'll be able to interrupt it if it stops for input. If it doesn't, on Windows we can press <ctrl> <alt> and <del> keys and kill the running process identified using the task manager. On Linux we can use a kill command line after identifying the process number using the ps command, e.g. kill -SIGKILL 1234 (assuming 1234 is the identified PID or process identifier).

Some long running server programs are designed to continue in their main loop e.g. processing email deliveries or requests for web pages until the machine they are running on shuts down. Much more commonly, we will want to exit a loop once a job it is doing is complete.

4.2 Testing for loop exit at the top

Copy the following code into a python file, save the file with appropriate name, and then run the python module:

```
def getrangedint():
        min, max = 1, 12
        x=0
        while x < min or x > max:
                x = int(input("enter a number between 1 and 12: "))
        return x

getrangedint()
```

The output will looks like the following:

```
enter a number between 1 and 12: 13
enter a number between 1 and 12: 0
enter a number between 1 and 12: 5
>>>
```

In this example, an initial invalid value x=0 was set as this would force the loop to execute at least once. Within the body of the loop the user is prompted to input a number and this is converted to an integer. The loop control test: x < min or x > max gives a True result if the value for x is invalid, causing the user to be prompted again. When a valid input is given, the loop can exit, and x is returned to the caller.

4.3 Exiting at the loop bottom

The above approach is clumsy and difficult to read. We have to set x to something invalid first, to ensure the body of the loop is entered. That's messy. Also the loop continuation test checks for something invalid. It's easier to follow the logic of the program when reading source code if the test deciding the loop exit checks for valid input, and not an invalid one. The next approach demonstrates a different way of exiting a while loop, this time at the loop bottom not the top.

```
def getrangedint():
        min, max = 1, 12
        while True:
                x = int(input("enter a number between 1 and 12: "))
                if x >= min and x <= max:
                        return x
getrangedint()
```

```
enter a number between 1 and 12: 0
enter a number between 1 and 12: 13
enter a number between 1 and 12: 6
>>>
```

This is neater. If we had wanted to stay within the same function when exiting a loop, we could have used a break statement to exit the loop instead of a return statement. The return exited both the loop and the function. But it's better programming practice to compose our program using smaller functions which do one thing well, encapsulating self-contained units of behaviour. We can improve it a bit further, by making our min and max parameters, rather than hard coding them.

```
def getrangedint(min, max):
        while True:
                x = int(input("enter a number between 1 and 12: "))
                if x >= min and x <= max:
                        return x



getrangedint(1,12)
enter a number between 1 and 12: 0
enter a number between 1 and 12: 13
enter a number between 1 and 12: 2
>>>
```

That's even better because our function is more reusable. A good reason for exiting a loop at the top might be a loop which does something 0 or more times. An example is a program which reads a text file which can have 0 or more records, and which does the same thing for each one.

4.4 Exiting in the middle of a loop

The next pair of functions prompts a user for a list, using a middle of loop exit, whence the list is returned. Then a total() function is used to add up the items in the list provided as a parameter. The function returns the total or zero if the list is empty.

```
def getlist():
        set=[]
        while True:
                s=input("Enter a number or * to quit: ")
                if s == '*':
                        return set
                n=float(s)
                set.append(n)

def total(set):
        totl=0.0
        for i in set:
                totl += i
        return totl

set=getlist()
print(total(set))
```

```
Enter a number or * to quit: 2
Enter a number or * to quit: 5
Enter a number or * to quit: 7
Enter a number or * to quit: *
14.0
```

4.5 Further looping examples

The following example involves a pair of functions. The first returns True or False if its parameter is a prime or a compound number. The second function loops the first, and prints all compound numbers equal to or greater than its parameter. If its parameter is a prime, it won't output anything.

```
def isprime(x):
	for i in range(2,x):
		if x % i == 0:
			return False
	return True

def listcompounds_ge(y):
	while not isprime(y):
		print(y)
		y += 1


listcompounds_ge(13)
listcompounds_ge(98)
>>>
98
99
100
>>>

listcompounds_ge(999)
>>>
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
>>>
```

4.6 Further Student exercises

1. Develop a standalone Python program (i.e. as a saved file) which obtains an integer in the range 1-12, and which then outputs a times table using this integer. Save the program as anytimestable.py The program must contain 2 separate functions, one which prints a times table and the other which obtains and returns the ranged integer. In the main part of your program, you can assign a value returned by a function to a variable like this:

   table=getrangedint(1,12)

   And then the variable: table can be used as a parameter when calling the function with prints out that times table.

2. Develop a program which prompts the user to input a whole number greater than 2. All invalid input must be rejected, including whole numbers 2 or less. The program then outputs all prime numbers starting at 2, which are less than the number input by the user.

3. Advanced exercise (Optional): Redesign and rewrite this program, saving it using a different filename, e.g. fastprimes.py. Make it run faster by having it not perform checks for factors which it doesn't need to test. The idea is that it will only look for factors for each number which it is testing for primality, based upon the primes it has already discovered, and which will be stored in a list. Only prime numbers less than or equal to the square root of the number being tested for primality need to be tried as potential factors.

   Your optimised isprime() function within this new program version must make use of a list called primes, which can start as the list [2], e.g. primes = [2] assigned before the isprime() function definition . Whenever your optimised isprime() discovers another prime e.g. 3, 5, 7 etc. then it appends it to the stored list of primes. When you have completed this exercise, time the speed of this and the previous slower but simpler program, to compare how quickly each version computes and outputs all of the prime numbers less than 100,000.

5. Moodle submission

At the end of this lab session, zip the python files that you created in Sections **2.2**, **3.1.1**, **3.1.2**, **4.6.1**, **4.6.2 and 4.6.3 (optional)**, and then upload the zip file in the Moodle. **Note that, this submission will be used only to analyse your progress/performance and it will not be considered for the assessment of this module.**