Objective of this File:

- Experiment with the range function to generate lists.
- 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.
- 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.

1. Converting between floats and ints:

We can use the float() and int() functions on data of the other type to convert it. We could also convert any int into a float by multiplying it by 1.0 e.g

```
int(9.6)
Output
9

float(8)
Output
8.0

8*1.0
Output
8.0
```

Note: The use of the **float()** function is preferred because the source code expresses more clearly what the programmer intended.

1.2 Converting numbers into string format

There are 2 approaches, direct conversion, or interpolation of numbers into strings. We can use the **str()** function to convert a number into string directly.

```
str(8.5)
Output
'8.5'

str(3)
Output
'3'
```

We can also interpolate numbers into strings. Python uses **%d and %f for ints and floats**.

```
s = "an int %d number" % 10
print(s)
Output
an int 10 number


s = "a float %f number" % 3.3
print(s)
Output
a float 3.300000 number
```

%f gives 6 decimal places. We can choose how many decimals we want. Formatting money amounts in a report we will typically want 2 decimal places.

If we want to interpolate more than one number into a string, we can use a tuple, which is a comma separated set of values in round brackets, to supply the data.

```
a=10
b=3.6
print("a float %f and an int %d" % (b, a))
Output
a float 3.600000 and an int 10
```

1.3 Branching alternate paths of code execution using if, elif and else

```
def isleap(y):
        if y % 4 == 0:
                return True
        else:
                return False

isleap(2000)
Output
True


isleap(2012)
Output
True


isleap(2013)
Output
False
```

NOTE:

A. we have used a single = to assign a value to a variable, e.g. x=2 and we change or overwrite the value in x (if it had a value) by doing this. But in the above isleap function definition we use a double == to compare or test 2 values of a pair of

variables, literals or expressions to see whether or not these are the same. Here we are interested in whether or not the remainder on dividing y (the year) by 4 is equal to 0. We are not going to overwrite the value of the variable or expression on the left hand side of the == operator

B. The comparison statement: expr1 == expr2 has either a True or False result. If the 2 expressions are equal we get a True result and if these are different we get a False result.
C. The block of code immediately after the if comparison: statement is executed if the comparison is True, and if it is False, the block of code immediately after the else: statement is executed instead.

Does the above function correctly identify leap years in all cases? Actually it doesn't. It gives incorrect results for 1900 and for 2100. In the Gregorian Calendar years divisible by 400 (e.g. 2000) are leap years. Other years divisible by 100 are not leap years. Other years divisible by 4 are leap years and other years are not leap years.

```
1 def isleap(y):
2     if y % 400 ==0:
3         return True
4     elif y % 100 ==0:
5         return False
6     elif y % 4 == 0:
7         return True
8     else:
9         return False
```

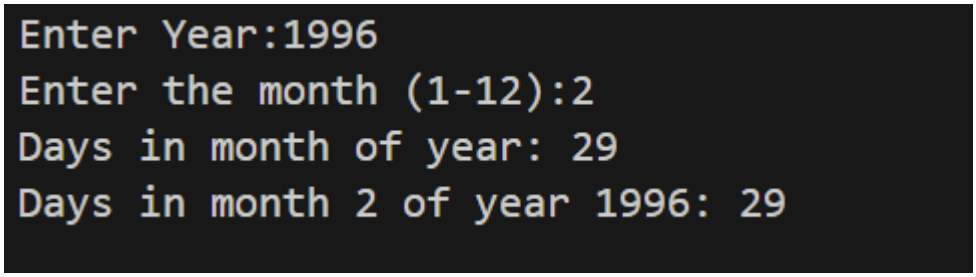
Note:

- One and only 1 branch of a nested if, elif, else set of code blocks can execute
- The if and every elif (else if) statement must be followed by an expression which can be evaluated as True or False before the colon (:) at the end of the line.
- These are tested starting at the top until a match is found
- If none of these tests after the if and elif's are True, then the else activates, which controls the default branch if all the above tests fail

Challenge:

Extend leap.py by adding a function daysinmonth() to which a month and year are passed as integer parameters. The function must branch based on month and return an integer, being the number 28, 29, 30 or 31. The month will be passed as a number between 1 and 12. If the month number is 2, the function isleap() must be called to

work out whether the year parameter is a leap year or not. Months with 31 days should be detected using an else: block. The number of days in the 12 months is known from this statement: "30 days has September, April, June and November. All the rest have 31, except February which has 28, and 29 for a leap year"



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:

```
print(type(("a",10.5)))
Output
<class 'tuple'>

print (type((0)))
Output
<class 'int'>

print(type((0,)))
Output
<class 'tuple'>

print(type(()))
Output
<class 'tuple'>


print(type("hello"))
Output
<class 'str'>
```

2.3 Lists

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

```
print(type([]))
```
Output
```
<class 'list'>
```

```
print(type([2,4]))
```
Output
```
<class 'list'>
```

```
print(type(['hello', 10.5]))
```
Output
```
<class 'list'>
```

2.2 Manipulating a 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

```
>>> l = [1,2,4]
>>> l.append(6)
>>> print(l)
[1, 2, 4, 6]
```

Question how to insert item to the 2nd parameter or 3th parameter?

Or how to print specific item from the list?

2.3 Measuring length, sorting and reversing a list

```
print(len(l))
```
Output
```
5
```
```
print(len("this"))
```
Output
```
4
```

```
print(len((0,)))
```
Output
```
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()
print(l)
```
Output
```
[5, 3, 4, 2, 1]
```

Try to remove item from the list.

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

```
print(list(range(1,10)))
```
Output
```
[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.

```
print(list(range(1,20,2)))
```
Output
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
print(list(range(3,20,3)))
```
Output
```
[3, 6, 9, 12, 15, 18]
```

2.4 Slicing a list

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

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.

```
>>> print(a)
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> b = a
>>> c = a[:]
>>> a.remove(3)
>>> print(a)
    [1, 2, 4, 5, 6, 7, 8, 9, 10]
>>> print(b)
    [1, 2, 4, 5, 6, 7, 8, 9, 10]
>>> print(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. 5) is passed as a parameter when we call the function. Copy and save the following code into a python file.

```
>>> def timestable(x):
...     for i in range(1,13):
...         print(" %d times %d  is %d" %(i, x, i*x))
...
...
>>>
>>> timestable(5)
 1 times 5  is 5
 2 times 5  is 10
 3 times 5  is 15
 4 times 5  is 20
 5 times 5  is 25
 6 times 5  is 30
 7 times 5  is 35
 8 times 5  is 40
 9 times 5  is 45
 10 times 5  is 50
 11 times 5  is 55
 12 times 5  is 60
```

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
```

Challenge:

Develop a Python program to print out all 10 times tables,

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

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?

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=" ")
```

Output

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
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.

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:

Output

```
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()
```

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):
    total= 0.0
    for i in set:
        total += i
    return total

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

Challenge:

Develop a program that prompts the user to input a whole number greater than 2. All invalid input, including whole numbers 2 or less,

must be rejected. The program then outputs all prime numbers starting at 2, which are less than the number input by the user.

Output

```
Enter number greater than 2 : 18
2
3
5
7
11
13
17
```

Further Challenge:

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.

Tip: 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.

```
enter  a number greater than 2:0
The number must be greater than 2. Try again.
enter  a number greater than 2:t
That's not a valid number. Please enter a whole number.
enter  a number greater than 2:50
Prime numbers less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>>
```