Module 2

Iteration

Indefinite loops

The while loop: The while statement is called an *indefinite* loop because it simply loops until some condition becomes **False**.

Flow of execution

1. Evaluate the condition, yielding **True** or **False**.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

Example: Simple program that counts down from five and then says "Blastoff!".

```
n = 5
while n > 0:
        print(n)
        n = n - 1
print('Blastoff!')
```

Output:

5
4
3
2
1
Blastoff!

Note: If there is no iteration variable, the loop will repeat forever, resulting in an *infinite loop*.

Infinite loops using break and continue:

The following python code takes input from the user until he types "done".

Code illustrating usage of break:

```
while True:
        line = input('> ')
        if line == 'done':
                break
        print(line)
print('Done!')
```

Explanation:

As the loop condition is always **True**, the loop runs repeatedly until it hits the **break** statement. Each time through, it prompts the user with an angle bracket('>'). If the user types "done", the **break** statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the beginning of the loop.

Here's a sample run:

```
> hello there
hello there
> finished
finished
> done
Done!
```

Code illustrating usage of break and continue:

When in an iteration of a loop we want to finish the current iteration and immediately jump to the next iteration, we can use the "continue" statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Sample code:

```
while True:
        line = input('> ')
        if line[0] == '#':
                continue
        if line == 'done':
                break
        print(line)
print('Done!')
```

sample run

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Note: Lines starting with # symbol are not printed.

Definite loops using for:

The for loop: The **for** loop is called as definite loop as it loops through a known set of items, i.e: it iterates as many times as there are items in the set.

Code illustrating usage of for loop:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
        print('Happy New Year:', friend)
print('Done!')
```

Explanation with output:

The variable "friends" is a list with three strings in it and the "**for"** loop goes through the list and executes the body once for each of the three strings in the list.

Output:
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!

Constructing Loop Patterns

a) Initializing one or more variables before the loop starts
b) Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
c) Looking at the resulting variables when the loop completes

Performing Counting, Summing, Maximum and Minimum element using for loop:

Code for Counting:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
        count = count + 1
print('Count: ', count)
```

Output:
Count: 6

Code for Summing:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
        total = total + itervar
print('Total: ', total)
```

Output:
Total: 154

Finding Largest element:

```
largest = None
print('Before: ', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
        if largest is None or itervar > largest :
                largest = itervar
        print('Loop: ', itervar, largest)
print('Largest:', largest)
```

Output:

```
Before: None
Loop: 3     3
Loop: 41    41
Loop: 12    41
Loop: 9     41
Loop: 74    74
Loop: 15    74
Largest: 74
```

Finding Smallest element:

```
smallest = None
print('Before: ', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
        if smallest is None or itervar <smallest :
                smallest = itervar
        print('Loop: ', itervar, smallest)
print('Smallest:', smallest)
```

Output:

```
Before: None
Loop: 3    3
Loop: 41   3
Loop: 12   3
Loop: 9    3
Loop: 74   3
Loop: 15   3
Smallest:  3
```

Finding Minimum using Python built-in min() function:

```
def min(values):
smallest = None
for value in values:
        if smallest is None or value < smallest:
                smallest = value
return smallest
```

Note: In Python programming indentation is very important.

Strings

A string is a sequence:

A string is a *sequence* of characters, which can be accessed one character at a time with the bracket operator. The expression in brackets is called an *index*. The index indicates which character in the sequence is to be accessed. Index value starts from 0. Expressions, variables and operators can be used as an index, but the value of the index has to be an integer.

Example:

```
fruit = 'pineapple'
letter = fruit[4]
print(letter)
```

Output:

a

Finding the length of a string using len():

Example:

```
#len() is a built-in function used to retrieve the number of characters in a given string:
fruit = 'pineapple'
print(len(fruit) )
```

Output:

9

```
#To get the last character, you have to subtract 1 from length:
fruit = 'apple'
last = len(fruit)-1
print(fruit[last])
```

Output:

e

Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, selects each character in turn, processes it, and repeats this until the end of the string. This pattern of processing is called a *traversal*.

Traversal using while loop

```
fruit = ‘mango’
index = 0
while index < len(fruit):
        letter = fruit[index]
        print(letter)
        index = index + 1
```

This loop traverses the string and displays each letter on a line by itself.

Output:

```
m
a
n
g
o
```

String slices

A segment of a string is called a slice. Selecting a slice is similar to selecting a character

Examples:

```
s = 'Monty Python'
print(s[0:5])
```

Output:
Monty

Here alphabets whose index value ranges from 0(index of alphabet 'M') to 4(index of alphabet 'y')(one less than index 5) is selected.

```
print(s[6:12])
```

Output:
Python

Here alphabets whose index value ranges from 6 to 11(one less than index 12) is selected.

Index →	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Element →	M	o	n	t	y		P	y	t	h	o	n

If the first index (before the colon) is omitted, the slice starts at the beginning of the string. If the second index is omitted index (after the colon), the slice goes to the end of the string. If both the index is omitted then the resulting string is the string itself. But if both the index values are dame, then the resulting string is an empty string.

Examples:

```
s = 'Monty Python'
print(s[0:5])    #Monty
print(s[6:12])   #Python
print(s[:5])     #Monty
print(s[6:])     #Python
print(s[5:5])    #Empty string
print(s[:])      #Monty Python
print(s[::-1])   # nohtyP ytnoM
```

Strings are immutable

Immutable means that existing string value cannot be changed. We can only create a new string that is a variation on the original.

Example:

```
greeting = 'Hello, world!'
```

```
new_greeting = 'J' + greeting[1:]
print(new_greeting)
```

Output:
Jello, world!

Looping and Counting

Program to count the occurrence of a's using for loop

```
word = 'Pineapple'
count = 0
for letter in word:
    if letter == 'p':
        count = count + 1
print(count)
```

Output:
2

```
word = 'Pineapple'
count = 0
for letter in word.lower():
    if letter == 'p':
        count = count + 1
print(count)
```

Output:
3

The in operator

The word **in** is a boolean operator that takes two strings and returns **True** if the first appears as a substring in the second else returns **False**

Examples:

```
'a' in 'banana'
```

Output:
True

```
'seed' in 'banana'
```

Output:
False

String comparison

The comparison operators work on strings. To see if two strings are equal

```
if word == 'banana':
        print 'All right, bananas.'
```

Other comparison operations are useful for putting words in alphabetical order

```
if word < 'banana':
```

```
        print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
        print 'Your word,' + word + ', comes after banana.'
else:
        print 'All right, bananas.'
```

Note: Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

String methods

Strings are an example of Python *objects*. An object contains both data (the actual string itself) and *methods*, which are effectively functions that are built into the object and are available to any *instance* of the object. Python has a function called **dir** which lists the methods available for an object.

List of methods avalaible in strings

The **type()** function shows the type of an object and the **dir()** function shows the available methods.

```
stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode','endswith' , 'expandtabs', 'find', 'format', 'format_map',
'index' , 'isalnum', 'isalpha', 'isdecimal', 'isdigit','isidentifier' , 'islower', 'isnumeric', 'isprintable','isspace' , 'istitle', 'isupper', 'join', 'ljust', 'lower','lstrip' , 'maketrans', 'partition', 'replace', 'rfind','rindex' , 'rjust', 'rpartition', 'rsplit', 'rstrip','split' , 'splitlines', 'startswith', 'strip', 'swapcase','title' , 'translate', 'upper', 'zfill']
```

Examples:

a) String method named **upper()** takes a string and returns a new string with all uppercase letters.
Method syntax: **word.upper()**

```
word = 'banana'
new_word = word.upper()
print(new_word)
```

Output:

BANANA

Note: The method lower takes a string and returns a new string with all lowercase letters.
Method syntax: **word.lower()**

b) String method named **find()** searches for the position of one string within another.

```
word = 'banana'
index = word.find('a')
print(index)
```

Output:
1

#Different cases in find method
The find method can find substrings as well as characters

```
word.find('na')
```

Output:
2

It can take as a second argument the index where it should start

```
word.find('na', 3)
```

Output:
2

```
word.find('na', 3)
```

Output:
4

c) String method named strip which removes white space (spaces, tabs, or newlines) from the beginning and end of a string

```
line = '  Here we go '
line.strip()
```

Output:
'Here we go'

Note: lstrip() and rstrip() remove whitespace at the left or right

d) String method named **startswith** return boolean values.

```
line = 'Have a nice day'
line.startswith('Have')
```

Output:
True

line.startswith('h')
Output:
False

Note: startswith requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the lower method.

line = 'Have a nice day'
line.startswith('h')

Output:
False

>>>line.lower()
'have a nice day'

>>>line.lower().startswith('h')
Output:
True

In the last example, the method **lower()** is called with **startswith()** to see if the resulting lowercase string starts with the letter "h".

Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:
From stephen.marquard@ uct.ac.za Sat Jan 5 09:14:16 2008
and we wanted to pull out only the second half of the address (i.e., uct.ac.za) from each line, we can do this by using the find method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space *after* the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.
Illustration:
data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
atpos = data.find('@')
print(atpos)

Output:
21

sppos = data.find(' ',atpos)
print(sppos)

Output:
31

```
host = data[atpos+1:sppos]
print(host)
```

Output:
uct.ac.za

Format operator
The *format operator*, % allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator. The first operand is the *format string*, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.
Example :

```
camels = 42
'%d' % camels
```

Output:
'42'

Example:
Python program to display the list and count of vowels in a given string.

```
def vowels(s):
   vowels='aeiou'
   v=0

   for letter in s:
      if letter in vowels:
         v+=1
         print(letter,end=' ')
   print("\nNumber of vowels: ",v)


text=input("Enter a string:")
print("The list of vowels found in \"", text ,"\" are: ")
vowels(text.lower())
```

Additional examples:

```
>>> text='It's ok'
SyntaxError: invalid syntax

>>> text="It's ok"
>>> text
"It's ok"
```

```
>>> text='She said "Hello" to all'
>>> text
'She said "Hello" to all'

>>> address= 302, Skyway Grandeur
SyntaxError: invalid syntax

>>> address='''1005, Skyway Grandeur,
3 purple street,
Gravenstein Highway North
Sebastopol,
CA 95472'''
>>> address
'1005, Skyway Grandeur,\n3 purple street,\nGravenstein Highway North\nSebastopol,\nCA
95472'
>>> print(address)
1005, Skyway Grandeur,
3 purple street,
Gravenstein Highway North
Sebastopol,
CA 95472
```

Chapter 7: Files

Opening files

To read from or write to a file (on hard drive), we must first *open* the file. Opening the file communicates with the operating system, which knows where the data for each file is stored. When opening a file, the operating system finds the file by name and makes sure that the file exists. In this example, we open the file mbox.txt, which should be stored in the same folder that we are in when we start Python.

Syntax:
File handle=open('filename')

Example:

```
fhand = open('mbox.txt')
```

If the open is successful, the operating system returns us a *file handle*. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. We are given a handle if the requested file exists and we have the proper permissions to read the file. If the file does not exist, open will fail with a traceback and we will not get a handle to access the contents of the file

Text files and lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters.

Example: this is a sample of a text file which records mail activity from various individuals in an open source project development team:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
Concept of lines in files

To break the file into lines, there is a special character that represents the "end of the line" called the *newline* character.In Python, we represent the *newline* character as a backslash-n in string constants.Even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the \n in the string, but when we use print to show the string, we see the string broken into two lines by the newline character.

Example:

```
>>>stuff = 'X\nY'
>>>print(stuff)
X
Y
>>>len(stuff)
3
```

We can also see that the length of the string X\nY is *three* characters because the newline character is a single character. When we look at the lines in a file, we need to *imagine* that there is a special invisible character called the newline at the end of each line that marks the end of the line. The newline character separates the characters in the file into lines.

Reading files

Syntax:
File handle=open('filename','r')

Note: Here **r** specifies the mode which says that file is opened in **read mode.** If second argument is not specified, then file is opened in read mode by default. i.e: **File handle=open('filename')**

Example: Counting number of lines in file 'mbox-short.txt' and printing the total count

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
        count = count + 1
print('Line Count:', count) # count depends on number of lines present in file
```

Example: Counting number of characters in file 'mbox-short.txt' and printing the total count

```
fhand = open('mbox-short.txt')
inp = fhand.read()
print len(inp)
```

Output:
94626 #includes total number of characters + newline character present in the file 'mbox-short.txt'
print(inp[:20]) # read the first 20 charaters of the file using slicing concept
Output:
From stephen.marquar # the first 20 characters present in the file 'mbox-short.txt'

Searching through a file

We can combine the pattern for reading a file with string methods to build simple search mechanisms.

Example:
If we wanted to read a file and only print out lines which started with the prefix "From:", we could use the string method *startswith* to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
        if line.startswith('From:'):
        print(line)
```

Output:
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu

The above lines are the lines which starts from "From". One more thing what we observe in the output is that there are lot of blank spaces between the lines.These blank spaces are due to existence of invisible newline characters at the end of each line and then print adds *another* newline, resulting in the double spacing effect we see.in output.

To solve the prolem we have to use rstrip method which removes these extra white spaces at the right side

Modfied code:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if line.startswith('From:'):
                print(line)
```

Output:

From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...

Searching operation-skipping uninteresting lines
Code:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        # Skip 'uninteresting lines'
        if not line.startswith('From:'):
                continue
        # Process our 'interesting' line
        print(line)
```

Output:
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...

Using in to select lines

We can look for a string anywhere in a line as our selection criteria

Code:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if not '@uct.ac.za' in line : continue
        print line
```

Output:
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za

Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...

Here lines containing "@uct.ac.za" will be selected from the file 'mbox-short.txt'

User choosing the file name
It would be more usable to ask the user to enter the file name string each time the program runs so we can use our program on different files without changing the Python code.

Code:
```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
        if line.startswith('Subject:'):
                count = count + 1
print('There were', count, 'subject lines in', fname)
```

We read the file name from the user and place it in a variable named fname and open that file. Now we can run the program repeatedly on different files.

Output 1:
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

Output 2:
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt

Using try, except, and open

The QA team is responsible for finding the flaws in programs before delivering the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the try/except structure. Assuming that the open call might fail, we need to add recovery code when the open fails as follows :

Code:
```
fname = input('Enter the file name: ')
try:
```

```
        fhand = open(fname)
except:
        print('File cannot be opened:', fname)
        exit()

count = 0
for line in fhand:
        if line.startswith('Subject:'):
                count = count + 1
print('There were', count, 'subject lines in', fname)
```

Note: The exit() function terminates the program. It is a function that we call that never returns.

Output 1
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

Output 2
Enter the file name: na na boo boo
File cannot be opened: na na boo boo

Writing files

To write a file, we have to open it with mode "w" as a second parameter:

Syntax:
File handle=open('filename','w')

Example:
If the file already exists, opening it in write mode clears out the old data and starts fresh. If the file doesn't exist, a new one is created. In order to add contents to the file open the file in append mode using second parameter as 'a'.

The write method of the file handle object puts data into the file, returning the number of characters written. The file object keeps track of where the content was written last time. so if we call write again, it adds the new data to the end. We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The print statement automatically appends a newline, but the write method does not add the newline automatically. When writing is complete, close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

Example: (in Interpreter Mode)

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>

>>> line1 = "This here's the wattle,\n"
```

```
>>> fout.write(line1)
24
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
>>>fout.close()
```

Example: (in Script Mode)

```
fout = open('output.txt', 'w')          #open the file named output.txt in write mode
line1 = "This here's the wattle,\n"     #content to be written
fout.write(line1)                       #writes line1
fout.close()                            #closing the file
```

Example:

Python program to display the list and count of vowels in a given file.

```
def vowels(s):
    vowels='aeiou'
    v=0
    for letter in s:
        if letter in vowels:
            v+=1
            print(letter,end=' ')
    print("\nNumber of vowels: ",v)

fh=open("files.txt")

print("The list of vowels found in file: ", fh.name ," are: ")
for line in fh:
    vowels(line.lower())
```

Output:

```
The list of vowels found in file:  files.txt  are :
e e a o e a o i e a i e o a i e e u i o e e i e o e i e i e o u i a e i o u o e a i e i o e e e a a o e a i e i o e
e o u o e a i e o u a e a i e o e a i e o i e i e a e a a e u e e i e e i i i e a e e o e e i e o i o u e o e i e a
e o e a o u a e i e e a o
Number of vowels:  129
```

```
def vowels(s):
   vowels='aeiou'
   v=0
   for letter in s:
      if letter in vowels:
```

```
            v+=1
            print(letter,end=' ')
    print("\nNumber of vowels: ",v)

fh=open("files.txt")
print("The list of vowels found in file: ", fh.name ," are: ")
for line in fh:
    line= line.lower()
    vowels(line)
```