Unit-5

Control Flow

Conditional execution

Conditional statements allow checking the conditions and changing the behavior of the program accordingly.
The simplest form is the if statement:

```
if x > 0 :
        print('x is positive')
```

Tab Spacing

The boolean expression after the if statement is called the *condition*. The ***if statement*** with a colon character (:) and the line(s) after the **if** statement are indented (as indicated by the arrow tab spacing).

The statement consists of a header line that ends with the colon character (:) followed by an indented block. These statements are called *compound statements* because they stretch across more than one line.

There is no limit on the number of statements that can appear in the body, but there must be at least one. If body has no statements then ***pass*** statement may be used, which does nothing.

For Example:

```
x=100
if x>99:
  x=x+500
if x < 101 and x>99:
  y=x+100
  x=x-50
  x=x+y
print(x)
```

OUTPUT:
600

Alternative execution

A second form of **if** statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0:
        print('x is even')
else:
        print('x is odd')
```

Ex. Write a program in python to accept an integer from the user and determine whether its even or odd.

Solution:

```
num=input('Input a number:')
num=int(num)
if(num!=0):
   if num%2!=0:
      print(num,'is an Odd number')
   else:
      print(num,'is an Even number')
```

Chained conditionals
If there are more than two possibilities and we need more than two branches. This can be done by *chained conditional*:

```
if x < y:
          print('x is less than y')
elif x > y:
          print('x is greater than y')
else:
          print('x and y are equal')
```

elif is an abbreviation of "**else if**". There is no limit on the number of **elif** statements.

Example:
Write a program in python to accept an integer from the user and determine whether its positive, negative or zero.

Solution:

```
num=input('Input a number:')
num=int(num)
if num >0:
      print(num,' is a positive number')
elif num<0:
           print(num,' is a negative number')
else:
           print(num,'  is zero')
```

Nested conditionals
One conditional can also be nested within another. The three-branch examples could be written as:

```
if x == y:
          print('x and y are equal')
else:
      if x < y:
              print('x is less than y')
      else:
              print('x is greater than y')
```

Example:

Write a program in python to accept three integers from the user and determine the largest of the three.

```
n1=input('Input the first number:')
n1=int(n1)
n2=input('Input the second number:')
n2=int(n2)
n3=input('Input the third number:')
n3=int(n3)
if n1>n2:
   if n1>n3:
      print(n1, ' is the largest')
   else:
      print(n3, ' is the largest')
elif n2>n3:
   print(n2, ' is the largest')
else:
   print(n3, ' is the largest')
```

Catching exceptions using try and except

```
>>>inp = input('Enter Fahrenheit Temperature: ')
>>>fahr = float(inp)
>>>cel = (fahr - 32.0) * 5.0 / 9.0
>>>print(cel)
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
pythonfahren.py

Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py

Enter Fahrenheit Temperature: amar
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
fahr = float(inp)
ValueError: could not convert string to float: 'amar'
```

The above program can be rewritten using **try** and **except** feature in Python as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
        fahr = float(inp)
        cel = (fahr - 32.0) * 5.0 / 9.0
        print(cel)
except:
        print('Please enter a number')
```

Short-circuit evaluation of logical expressions

Python processes a logical expression such as x >= 2 and (x/y) > 2, by evaluating the expression from left to right. When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical stops because the overall value is already known, it is called *short-circuiting* the evaluation.

The short-circuit behavior leads to a clever technique called the ***guardian pattern***. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Using the guardian pattern prior to evaluation:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In the first logical expression, x >= 2 is False so the evaluation stops at the **and.** In the second logical expression, x >= 2 is True but y != 0 is False so the program never reaches (x/y). In the third logical expression, the y != 0 is *after* the (x/y) calculation, so the expression fails with an error. In the **second expression**, **y != 0 acts as a *guard*** to insure that only (x/y) is executed if y is non-zero.

Debugging

Python displays an error message that contains a lot of information. The most useful parts are usually:

- What kind of error it was, and

- Where it occurred.

Syntax errors are usually easy to find, but there are few tricky errors such as ***Whitespace*** because spaces and tabs are invisible and most ignore them.

```
>>> x = 5
>>>  y = 6
File "<stdin>", line 1
y = 6
^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.
In general, error messages tell where the problem was discovered, but that is often not where it was caused.

Lists

A list is a sequence Like a string, a *list* is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called *elements* or sometimes *items*.

```
[10, 20, 30, 40]
[1,2,3.3,'Cat','Mat',5.5]
['Apple','Orange','Banana']
```

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is nested. A list that contains no elements is called an empty list; empty list can be created with empty brackets [ ].

List can be assigned to variables:

```
>>> names = ['Ram', 'Ramses', 'Ramesh']
>>> numbers = [22, 1941]
>>> empty = [ ]

>>> print(names, numbers, empty)
```

OUTPUT:

```
['Ram', 'Ramses', 'Ramesh'][22,1941][ ]
```

Lists are mutable

Unlike strings, lists are mutable because the order of items in a list can be changed or an item in a list can be reassigned.

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator.
The expression inside the brackets specifies the index.

The indices start at 0:

```
>>> print(names[0])
Ram
```

When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [22,1941]
>>> numbers[1] = 2
>>> print(numbers)
    [22, 2]
```

List has a relationship between indices and elements. This relationship is called a *mapping*; each index " maps to" one of the elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If an element is read or written that does not exist, an **IndexError** is generated.
- If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists.

```
>>> names = ['Ram', 'Ramses', 'Ramesh']
>>> 'Ramses' in names
True

>>> 'Bob' in names
False
```

Traversing a list

The most common way to traverse the elements of a list is with a **for** loop. The syntax is the same as for strings:

```
for name in names:
        print(name)

for i in range(len(numbers)):
        numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n − 1, where n is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

A for loop over an empty list never executes the body:

```
for x in empty:
        print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

['Draft', 1, ['John','Bob','Sam'], [1, 2, 3]]

List operations
The + operator concatenates lists:

```
>>a = [1, 2, 3]
>>b = [4, 5, 6]
>>c = a + b
>>print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>> [0] * 4
[0, 0, 0, 0]
>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']

>>> t[:4]
['a', 'b', 'c', 'd']

>>> t[3:]
['d', 'e', 'f']
```

If the first index s skipped, the slice starts at the beginning. If second index is kipped, the slice goes to the end. So if both indices are skipped, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods
Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements while it leaves t2 unmodified:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

Sorting a list:

Sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None.

Deleting elements

There are several ways to delete elements from a list. If the index of the element to be deleted is known, pop can be used:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

pop modifies the list and returns the element that was removed. If an index is not provided, it deletes and returns the last element.

If value removed is not desired, then the del operator can be used:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

remove():

If the element to be removed is known (but not the index), remove can be used:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

The return value from remove is None.

To remove more than one element, **del** can be used with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

Practice Questions

1. Write a Python program that accepts a list and modifies it by removing the first and last elements and then first prints those deleted items and then it prints the rest of the list.
2. Write a Python program that accepts a list and modifies it by removing the first and last elements and then copies the remaining list to a new list. Print the new list.

Tuples:

Tuples are immutable

A tuple is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are *immutable*. Tuples are *comparable* and *hashable* so can be sorted and also used as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print(t)
()
```

If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Because tuple is the name of a constructor, should be avoided to be used as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if one of the elements of the tuple is modified, it generates an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

The elements of a tuple can't be modified, but can be replaced with another tuple:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

tuple v/s list:

- Tuples are stored in a single block of memory. Tuples are immutable. It doesn't require extra space to store new objects.
- Lists are allocated in two blocks: the fixed one with all the Python object information and other with a variable sized block for the data.
- Hence coz. of this reason creating a tuple is faster than List.
- Whenever application needs a constant set of values and which can be iterated through it, then tuple is preferred than a list.
- Tuple makes the code safer if the data is to be "*write-protect*" i.e. it need not to be changed anytime later.

Comparing tuples

The comparison operators works with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on. This feature lends itself to a pattern called *DSU* for **Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence, **Sort** the list of tuples using the Python built-in sort, and **Undecorate** by extracting the sorted elements of the sequence. [DSU].

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()

for word in words:
        t.append((len(word), word))
t.sort(reverse=True)
res = list()

for length, word in t:
        res.append(word)
print(res)
```

```
['yonder', 'window', 'breaks', 'light', 'what', 'soft', 'but', 'in']
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length. sort compares the first element length, first, and only considers the second element to break ties. The keyword argument **reverse=True** tells sort to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in *reverse* alphabetical order, so “what” appears before “soft” in the following list. The output of the program is as follows:

Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows to assign more than one variable at a time when the left side is a sequence. In this example there is a two-element list (which is a sequence) and statements following it assigns the first and second elements of the sequence to the variables x and y in a single statement.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

The following is also an equally valid syntax:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
```

```
>>>
```

A particularly clever application of tuple assignment allows us to *swap* the values of two variables in a single statement:

```
>>> a, b = b, a
```

The number of variables on the left and the number of values on the right must be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list, or tuple).

For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

Dictionaries

A *dictionary* is like a list. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type. Dictionary can be visualized as a mapping between a set of indices (which are called *keys*) and a set of values. Each key maps to a value. The association of a key and a value is called a *key-value pair* or sometimes an *item*.

The function **dict** creates a new dictionary with no items. Because **dict** is the name of a built-in function, should be avoided as a variable name.

```
eng2sp = dict()
print(eng2sp)
{}
```

The curly brackets, {}, represent an empty dictionary. To add items to the dictionary, square brackets can be used:

```
eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key ’one’ to the value “uno”. If the dictionary is printed, it prints the a key-value pair with a colon between the key and value:

```
print(eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, to create a new dictionary with three items.

```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2sp)

{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. Hence, the order of items in a dictionary is unpredictable. The elements of a dictionary are never indexed with integer indices. Instead, keys are used to look up the corresponding values:

```
>>> print(eng2sp['two'])
'dos'
```

The key ’two’ always maps to the value “dos” so the order of the items doesn’t matter.
If the key isn’t in the dictionary, an exception is generated:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The **in** operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary.

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, the method values can be used, which returns the values as a list, and then use the in operator:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

The **in** operator uses different algorithms for lists and dictionaries. For **lists**, it uses a **linear search** algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list.
For dictionaries, Python uses an algorithm called a ***hash table*** that has a remarkable property: the **in** operator takes about the same amount of time no matter how many items there are in a dictionary.

Exercise 1:

Write a program that reads accepts words from the user until user type '**End'** when done. Store all those words as ***keys*** in a dictionary along with the length of each of the words with their keys respectively. Accept a search string key from the user and then check whether the key is present in the dictionary using the in operator and if found display '***Search Key Successful***' and also display the key with its value otherwise display '***Search key not found***'.

Solution:

```
Dict={}
while(True):
   word=input('Enter words when finished type \'End\':')
   if word=='End':
      break
   word=word.strip()
   Dict[word]=len(word)

for k,v in Dict.items():
   print(k,'=>',v)
srch=input('Enter the search string:')
if srch in Dict.keys():
   print(srch ,' is present and its value is: ', Dict[srch])
else:
   print(srch ,' is not present')
```

Dictionary as a set of counters

Suppose it's required to count how many times each letter appears in a given string. There are several ways it can be done:

1. By creating 26 variables, one for each letter of the alphabet. Then traverse the string for each character and incrementing the corresponding counter, probably using a chained conditional.
2. By creating a list with 26 elements. Then by converting each character to a number (using the built-in function ord), then using the number as an index into the list, and incrementing the appropriate counter.
3. By creating a dictionary with characters as keys and counters as the corresponding values. The first character, would be just added as a first item to the dictionary. After that the value of an existing item could be incremented.

For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear. Here is what the code might look like:

```
word = 'brontosaurus'
d = dict()
for c in word:
        if c not in d:
        d[c] = 1
        else:
        d[c] = d[c] + 1
print(d)
```

get Method:

Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

We can use get to write our histogram loop more concisely. Because the get method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the if statement.

```
word = 'brontosaurus'
d = dict()
for c in word:
        d[c] = d.get(c,0) + 1
print(d)
```

Looping and dictionaries

If you use a dictionary as the sequence **in** for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
        print(key, counts[key])
```

Here’s what the output looks like:

```
jan 100
chuck 1
annie 42
```

Again, the keys are in no particular order. For example to find all the entries in a dictionary with a value above ten, the following code can be written as:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
        if counts[key] > 10 :
        print(key, counts[key])
```

The for loop iterates through the *keys* of the dictionary, the index operator is used to retrieve the corresponding *value* for each key. Here’s what the output looks like:

```
jan 100
annie 42
```

If the keys needs to be printed in alphabetical order, first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
        print(key, counts[key])
```

Here’s what the output looks like:

```
['jan', 'chuck', 'annie']

annie   42
chuck   1
jan     100
```

Advanced text parsing

The actual text file romeo.txt has lots of punctuation, as shown below.

But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,

Since the Python split function looks for spaces and treats words as tokens separated by spaces, the words “soft!” and “soft” would be treated as *different* words and create a separate dictionary entry for each word.

Also, since the file has capitalization, the words “who” and “Who” will be treated as different words with different counts.

We can solve both these problems by using the string methods lower, punctuation, and translate. The translate is the most subtle of the methods. Here is the documentation for translate:

line.translate(str.maketrans(fromstr, tostr, deletestr))

*Replace the characters in **fromstr** with the character in the same position in **tostr** and delete all characters that are in **deletestr**. The **fromstr** and **tostr** can be empty strings and the **deletestr** parameter can be omitted.*

Python tell us the list of characters that it considers “punctuation” by using the following code:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

We make the following modifications to our program:

```
import string
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit(1)          #Abnormally exits the program
    counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
    if word not in counts:
        counts[word] = 1
    else:
        counts[word] += 1
print(counts)
```

Dictionaries and tuples

Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

The items are in no particular order.
However, since the list of tuples is a list, and tuples are comparable, the list of tuples can be sorted by converting a dictionary to a list of tuples, which will output the contents of a dictionary sorted by key:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

Multiple assignment with dictionaries
Combining items, tuple assignment, and **for**, the keys and values of a dictionary can be traversed in a single loop:

```
for key, val in list(d.items()):
        print(val, key)
```

This loop has two *iteration variables* because items returns a list of tuples and key, val is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).

If these two techniques are combined, the contents of a dictionary can be printed and sorted by the *value* stored in each key-value pair.

To do this, first make a list of tuples where each tuple is (value, key). The items method would give a list of (key, value) tuples, but needs to be sorted by value, not key. Hence, all that needs to be done is sort the list in reverse order and print out the new, sorted list.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> w = list()
>>> for key, val in d.items() :
...             w.append( (val, key) )
...
>>> w
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> w.sort(reverse=True)
>>> w
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

The most common words
Program to use the above technique to print the ten most common words in the text as follows:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
        line = line.translate(str.maketrans('', '', string.punctuation))
        line = line.lower()
        words = line.split()
        for word in words:
        if word not in counts:
        counts[word] = 1
        else:
        counts[word] += 1
# Sort the dictionary by value
lst = list()
for key, val in list(counts.items()):
        lst.append((val, key))
lst.sort(reverse=True)

for key, val in lst[:10]:
        print(key, val)
```

The first part of the program reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out counts and ending the program, list of (val, key) tuples is constructed and then list is sorted in reverse order. Since the value is first, it will be used for the comparisons.
If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.
At the end a **for** loop which does a multiple assignment iteration, prints out the ten most common words by iterating through a slice of the list (lst[:10]).

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

Using tuples as keys in dictionaries

Because tuples are *hashable* and lists are not, if a *composite* key needs to be created a dictionary is used where a tuple is used as the key. To create a telephone directory a composite key maps from last-name, first-name pairs to telephone numbers. A dictionary assignment statement can be written as follows:

```
directory[last,first] = number
```

The expression in brackets is a tuple. tuple assignment can be used in a for loop to traverse this dictionary.

```
for last, first in directory:
        print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

Exercise 1: Revise a previous program as follows: Read and parse the "From" lines and pull out the addresses from the line. Count the number of messages from each person using a dictionary. After all the data has been read, print the person with the most commits by creating a list of (count, email) tuples from the dictionary. Then print out the person who has the most commits.

Exercise 2: This program counts the distribution of the hour of the day for each of the messages. You can pull the hour from the "From" line by finding the time string and then splitting that string into parts using the colon character. Once you have accumulated the counts for each hour, print out the counts, one per line, sorted by hour as shown below.

```
Sample Execution: python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 2
17 2
18 1
19 1
```

Files

Stored on a Persistence storage device Ex. Secondary Memory, Pen drive, Hard Drive etc. Secondary memory is not erased when the power is turned off. The main focus will be on reading and writing text files such as those created in a text editor.

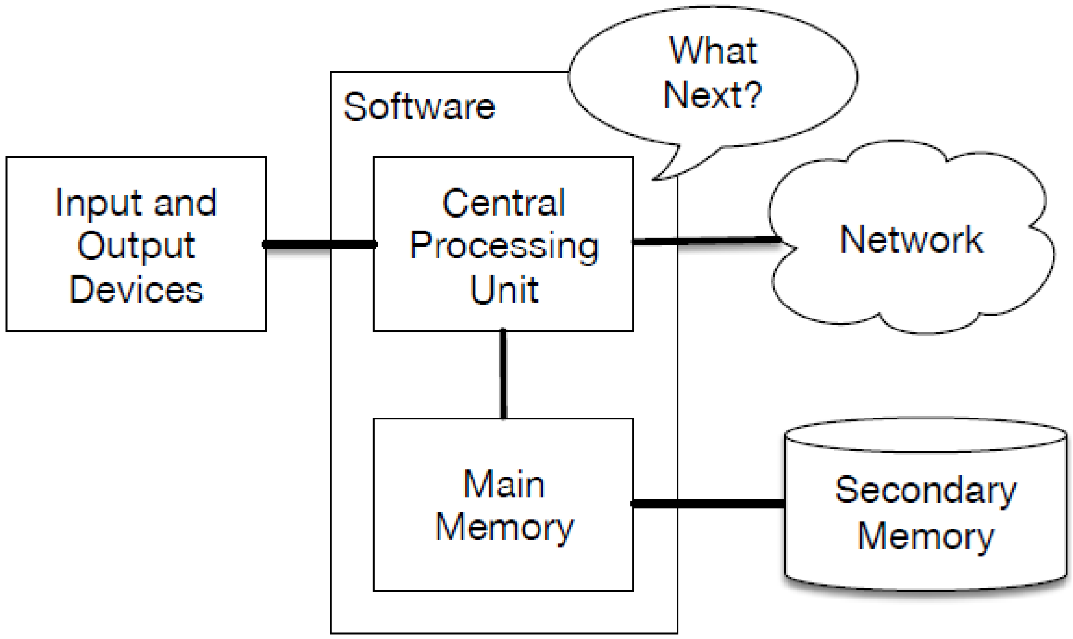


Figure 7.1: Secondary Memory

Opening files

When file has to be read or written (onto the hard drive), first it must be opened. Opening the file communicates with the operating system running on that computer, which knows where the data for each file is stored. When a file is opened, the operating system finds the file by name and makes sure the file exists.

General syntax:
filehandle= open("filename",mode)

The modes are:

'***r***' – Read mode is used when the file is only being read.

'***w***' – Write mode is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated).
'***a***' – Append mode, is used to add new data to the end of the file; that is new information is automatically appended to the end.
'***r+***' – Special read and write mode, is used to handle both actions when working with a file.
'x' – Create - will create a file, returns an error if the file exist.

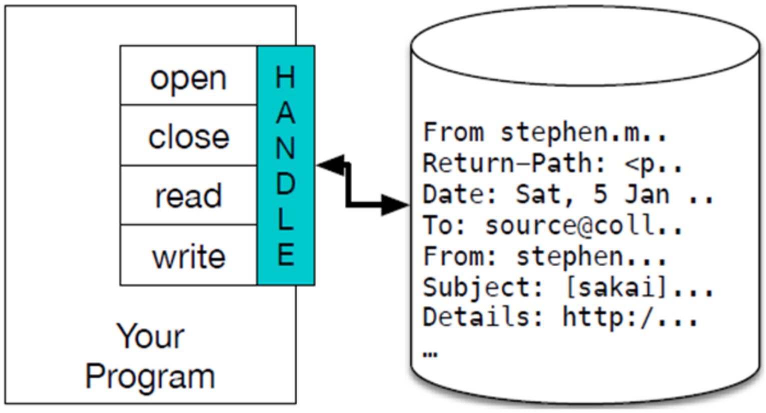
'rt': It opens in text file in read mode.

'b': This opens in binary mode. Binary format is *usually* used when dealing with images, videos, etc.
'+': This will open a file for reading and writing (updating).
'rb': Opens the file as read-only in binary format.
'wb': Opens a write-only file in binary mode.

In this example, the file ***mbox.txt*** is opened, which should be stored in the same folder as that off when Python was started.

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

If the **open** call is successful, the operating system returns a *file handle*. The file handle is not the actual data contained in the file, but instead it is a "handle" that can be used to read the data. A *handle* is successfully returned if the requested file exists and has the proper permissions to read the file.



A File Handle

If the file does not exist, **open** will fail with the message *Traceback,* and the handle is not available to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

try and **except** model can be used to handle the situation of opening a file that does not exist.

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. For 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
...
```

To break the file into lines, there is a special character that represents the “end of the line” called the *newline* character.In Python, newline character is represented as a backslash-n in string constants.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

Please Note that the files used in this document have been downloaded from **www.py4e.com/code3/**

Reading files

While the *file handle* does not contain the data for the file, it is quite easy to construct a **for** loop to read through and count each of the lines in a file:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
        count = count + 1
print('Line Count:', count)
```

The file handle can be used as the sequence in the for loop. The above **for** loop counts the number of lines in the file and prints them out. The meaning of the for loop into English is, “for each line in the file represented by the file handle, add one to the count variable.”

The reason that the open function does not read the entire file is that the file might be quite large with many gigabytes of data. The open statement takes the same amount of time regardless of the size of the file. The **for** loop actually causes the data to be read from the file. When the file is read using a **for** loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the line variable for each iteration of the **for** loop. Because the for loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If the file is relatively small compared to the size of your main memory, it can be read as the whole file into one string using the read method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable ***inp***. This form of the open function should only be used if the file data will fit comfortably in the main memory of your computer. If the file is too large to fit in main memory, one should write program to read the file in chunks using a **for** or **while** loop.

Searching through a file

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.
For 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)
```

When this program runs, we get the following output:

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

The output looks great since the only lines visible are those which start with **"From:",** but there are some extra blank lines. This is due to that invisible newline character. Each of the lines ends with a newline, so the print statement prints the string in the variable line which includes a newline and then print adds another newline, resulting in the double spacing effect we see.

The **rstrip** method which strips whitespace from the right side of a string as follows:

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

When this program runs, we get the following output:

When this program runs, we get the following output:

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

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

The loop can be modified to follow the pattern of skipping uninteresting lines as follows:

```
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)
```

The find string method can be used to simulate a text editor search that finds lines where the search string is anywhere in the line. Since find looks for an occurrence of a string within another string and either returns the position of the string or -1.

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if line.find('@uct.ac.za') == -1: continue
        print(line)
```

Letting the user choose the file name
We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code. This is quite simple to do by reading the file name from the user using input as follows:

```
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)
```

The file name is read from the user and placed in a variable named **fname** and then that file is opened. Now we can run the program repeatedly on different files.

```
python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Writing multiple lines to a file:

filehandle.writelines(list1)

list1: is a List containing multiple strings / items

For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

Example: To insert a list containing multiple strings into a file.

```
s1='An Apple a Day keeps the Doctor away.\n'
s2='Make Hay while the Sun shines.\n'
s3='Time and Money waits for none.\n'
lst=[s1,s2,s3]
fh=open('Proverb.txt','w+')
for w in lst:
   fh.writelines(w)
fh.seek(0)
```

# Reads and returns the characters until the end of the line is reached as a string.

```
print(fh.readline())
```

Binary File(s):

Binary files are files where the format isn't made up of readable characters. Binary files can be of types such as image files like JPEGs or GIFs, audio files like MP3s, video files like MP4 or binary document formats like Word or PDF.

filehandle=open(filename,'br+')

b: indicates the binary mode
r+: read and write mode

Example: To create a copy of a Binary file.

```
fh1=open('cover3.jpg','br')           #assuming cover3.jpg binary file is present in the current path
fh2=open('cover3Copy.jpg','bw')       #opens another empty file in binary write mode
byte=fh1.read(1)
while byte:                           #reads until byte is false i.e. byte=' '
   print(byte)
   fh2.write(byte)
   byte=fh1.read(8192)
fh1.close()
fh2.close()
```

OR

```
fh1=open('cover3.jpg','br')           #assuming cover3.jpg binary file is present in the current path
fh2=open('cover3Copy.jpg','bw')       #opens another empty file in binary write mode
byte=fh1.read(1)
while byte !=b'':                     #reads until byte is not equal to b'' ie empty byte
   print(byte)
   fh2.write(byte)
   byte=fh1.read(8192)
fh1.close()
fh2.close()
```

Dictionaries and files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. The combination of the two nested loops ensures that every word is counted which is on every line of the input file.

```
fname = input('Enter the file name: ')
try:
        fhand = open(fname)
except:
        print('File cannot be opened:', fname)
        exit(1)          #abnormally exits the program

counts = dict()
```

```
for line in fhand:
    words = line.split()        # splits the words separated by spaces contained in the line
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1
print(counts)
```