Module 3
Lists, Dictionaries, Tuples, Regular Expressions

Python Lists

A List is a kind of Collection

- A collection allows us to put many values in a single “variable”
- A collection can carry many values around in one convenient package.

Syntax:

```
list_variable=[ value1, value2,… ]
```

Eg:

```
friends = [ 'Joseph', 'Glenn', 'Sally' ]
marks = [ 75, 88, 96 ]
```

What is not a “Collection”

- Most of the variables have one value in them - when we put a new value in the variable - the old value is over written

  Eg:

```
>>> x = 2
>>> x = 4
>>> print (x)
4
```

List Constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas.
- A list element can be any Python object - even another list
- A list can be empty

  Eg:

```
>>> print ([1, 24, 76])
[1, 24, 76] # List elements
>>> print (['red', 'yellow', 'blue'])
['red', 'yellow', 'blue']                 # List elements
>>> print (['red', 24, 98.6])
['red', 24, 98.599999999999994]  # List elements
>>> print ([ 1, [5, 6], 7])
[1, [5, 6], 7]                    #Another list as list element
>>> print ([ ])
[ ]              #Empty list
```

We already used lists!

```
for i in [5, 4, 3, 2, 1] :
    print (i )
print (‘Done!’)

Output:
5
4
3
2
1
Done!
```

Lists and definite loops - best pals

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

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

Looking Inside Lists

- Just like strings, we can get at any single element in a list using an index specified in square brackets
  Eg:

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print (friends[1])
Glenn
```

Lists are Mutable

- Strings are "immutable" - we *cannot* change the contents of a string - we must make a new string to make any change
- Lists are "mutable" - we *can* change an element of a list using the index operator
  Eg:

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
```

```
Traceback TypeError: 'str' object does not
support item assignment
```

```
>>> x = fruit.lower()
>>> print (x)
Banana
>>> lotto = [2, 14, 26, 41, 63]
>>> print (lotto)
[2, 14, 26, 41, 63]
```

```
>>> lotto[2] = 28
>>> print (lotto)
[2, 14, 28, 41, 63]
```

How Long is a List?

- The **len**() function takes a list as a parameter and returns the number of *elements* in the list
  Eg:
  ```
  >>> greet = 'Hello Bob
  '>>> print (len(greet))
  9
  ```
- Actually **len**() tells us the number of elements of *any* set or sequence (i.e. such as a string...)
  Eg:
  ```
  >>> x = [ 1, 2, 'joe', 99]
  >>> print (len(x))
  4
  ```

Using the range function

- The **range** function returns a list of numbers that range from zero to one less than the parameter
  Eg:

```
>>> print (range(4))
[0, 1, 2, 3]
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print (len(friends))
3
```

```
>>> print (range(len(friends)))
[0, 1, 2]
```

- We can construct an index loop using **for** and an integer iterator

A tale of two loops...

Eg:

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

```
for i in range(len(friends)) :
    friend = friends[i]
    print ('Happy New Year:',  friend)
```

Output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
```

```
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print (len(friends))
3
>>> print (range(len(friends)))
[0, 1, 2]
>>>
```

Concatenating lists using '+'

- We can create a new list by adding two existing lists together

```
>>> a=[1,2,3]
>>> b=[2,4,6]
>>> c=a+b
>>> c
[1, 2, 3, 2, 4, 6]
```

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

Lists can be sliced using ':'

```
>>>a=[1,2,3]; b=[2,4,6]; c=a+b
>>> c
[1, 2, 3, 2, 4, 6]
>>> d=c[2:5]
>>> d
[3, 2, 4]
>>> c[1:]
[2, 3, 2, 4, 6]
```

```
>>> c[0:]
[1, 2, 3, 2, 4, 6]
>>> c[:4]
[1, 2, 3, 2]
>>> d[0:]
[3, 2, 4]
>>> c[:]
[1, 2, 3, 2, 4, 6]
```

Remember: Just like in strings, the second number is "up to but not including"

List Methods

Eg:

```
>> x = list()
>>> type(x)
<class 'list'>
>>> dir(x)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Building a list from scratch

- We can create an empty list and then add elements using the **append** method

```
>>> aList=[ ]
>>> bList=list()
```

```
>>> type(aList)
<class 'list'>
```

```
>>> type(bList)
<class 'list'>
>>> aList
[ ]
>>> bList
[ ]
```

```
>>> aList.append(12)
>>> aList
[12]

>>> bList.append('book')
>>> bList
['book']
```

- The list stays in order and new elements are added at the end of the list

Is Something in a List?

- Python provides two operators that let you check if an item is in a list: **in**, **not in**
- These are logical operators that return **True** or **False**
- They do not modify the list

Eg:

```
>>> some = [1, 9, 21, 10, 16]
>>> 9 in some
True
>>> 15 in some
False
>>> 20 not in some
True
```

A List is an Ordered Sequence

- A **list** can hold many items and keeps those items in the order until we do something to change the order
- A **list** can be **sorted** (i.e. change its order)
- The **sort** method (unlike in strings) means "sort yourself"

Eg:

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> friends.sort()
>>> print (friends)
['Glenn', 'Joseph', 'Sally']
>>> print (friends[1])
Joseph
```

Built in Functions and Lists

- There are a number of functions built into Python that take lists as parameters

Eg 1:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
```

```
>>> print max(nums)
74
```

```
>>> print min(nums)
3
>>> print sum(nums)
154
```

```
>>> print sum(nums)/len(nums)
25
```

Eg 2:

```
numlist = list()
while True :
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print( 'Average:', average)
```

Output:

```
Enter a number: 3
Enter a number: 9
Enter a number: 5
Enter a number: done
Average: 5.66666666667
```

Strings and Lists

```
>>> abc = 'With three words'
>>> stuff = abc.split()
>>> print (stuff)
['With', 'three', 'words']
>>> print (len(stuff))
3
```

```
>>> print (stuff[0])
With
>>> print (stuff)
['With', 'three', 'words']
```

```
>>> for w in stuff :
...             print (w)
...
With
Three
Words
```

Split breaks a string into parts and produces a list of strings. We think of these as words. We can access a particular word or loop through all the words.

```
>>> line = 'A lot               of spaces'
>>> etc = line.split()
>>> print (etc)
['A', 'lot', 'of', 'spaces']
>>>
>>> line = 'first;second;third'
>>> thing = line.split()
>>> print (thing)
['first;second;third']
```

```
>>> print (len(thing))
1
>>> thing = line.split(';')
>>> print (thing)
['first', 'second', 'third']
>>> print (len(thing))
3
>>>
```

- When you do not specify a **delimiter**, multiple spaces are treated like one delimiter
- You can specify what **delimiter** character to use in the **splitting**

Eg:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if not line.startswith('From ') : continue
        words = line.split()
        print (words[2])
```

Output:

```
Sat
Fri
Fri
Fri
…
```

```
>>> line=’ From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008’
>>> words = line.split()
>>> print (words[2])
['From', 'stephen.marquard@uct.ac.za', 'Sat', 'Jan', '5', '09:14:16', '2008']
```

The Double Split Pattern

- Sometimes we split a line one way and then grab one of the pieces of the line and split that piece again

List Summary

- Concept of a collection
- Lists and definite loops
- Indexing and lookup
- List mutability
- Functions: len, min, max, sum
- Slicing lists

- List methods: append, remove
- Sorting lists
- Splitting strings into lists of words
- Using split to parse string
- Slicing lists

```
>>> a=[1,2,3]
>>> a.insert(0,0)
>>> print(a)
[0, 1, 2, 3]
>>> a.insert(2,3)
>>> print(a)
[0, 1, 3, 2, 3]
>>> a.remove(3)
>>> print(a)
[0, 1, 2, 3]
>>> a.reverse()
>>> print(a)
[3, 2, 1, 0]
>>> a.reverse()
>>> print(a)
[0, 1, 2, 3]
>>> a=[1,3, 4, 5, 62,3, 4, 5, 6]
>>> a.sort()
>>> print(a)
[1, 3, 3, 4, 4, 5, 5, 6, 62]
>>> a.count(3)
2
>>> a.index(3)
1
>>> a.index(62)
8
```

```
>>> a.insert(7,10)
>>> print(a)
[1, 3, 3, 4, 4, 5, 5, 10, 6, 62]

>>> a=[1,2,3];b=a;print(b is a);
True
>>> b[0]=23
>>> print(a)
[23, 2, 3]
>>> print(a)
[23, 2, 3]
>>> a='apple'
>>> b='apple'
>>> a is b
True
>>> a=[1,2,3]
>>> b=[1,2,3]
>>> a is b
False
>>> a=[1,2,3];b=a;b is a;
True
>>> a=[1,2,3];b=a;a is b;
True
>>> a=[1,2,3];b=[1,2,3];b is a;
False
```

Example:

#Python program that uses 2D list

```
elements = [ ]          # Create a list.
# Append empty lists in first two indexes.
elements.append([ ])
elements.append([ ])
# Add elements to empty lists.
elements[0].append(1)
elements[0].append(2)
```

```
elements[1].append(3)
elements[1].append(4)

print(elements[0][0]) # Display top-left element.
print(elements) # Display entire list.

# Loop over rows.
for row in elements:
    # Loop over columns.
    for column in row:
        print(column, end="")
    print(end="\n")
```

Output:

```
1
[[1, 2], [3, 4]]
12
34
```

Jagged lists: The term "jagged" implies that sub lists have uneven lengths. Here we create a list of two lists—one of length 2, the other of length 5. We display their lengths.

#Python program that uses jagged lists

# A jagged list.

```
values = [[10, 20], [30, 40, 50, 60, 70]]

for value in values:
    # Print each row's length and its elements.
    print(len(value), value)
```

Output:

```
2 [10, 20]
5 [30, 40, 50, 60, 70]
```

Solve the following exercise questions:

1. Write a function called chop that takes a list and modifies it, removing the first and last elements, and returns None. Then write a function called middle that takes a list and returns a new list that contains all but the first and last elements.

```
def chop(lst):
    lst.pop(0)
    lst.pop(len(lst)-1)
    return
```

```
def middle(lst):
    return lst[1:len(lst)-1]

n=int(input("Enter a limit value: "))
oldList=[ ]
i=0
while i < n:
    oldList.append(input())
    i+=1
print(oldList)
ch=chop(oldList)
print(ch)
print(oldList)
newList=middle(oldList)
print(newList)
```

Output:

```
Enter a limit value: 7
2
2.3
I
AM
GOOD
[2,4,6,8,10,12]
22.3
['2', '2.3', 'I', 'AM', 'GOOD', '[2,4,6,8,10,12]', '22.3']
None
['2.3', 'I', 'AM', 'GOOD', '[2,4,6,8,10,12]']
['I', 'AM', 'GOOD']
```

2. Write a program to open a text file and read it line by line. For each line, split the line into a list of words using the split function. Then for each word, check to see if the word is already in a list. If the word is not in the list, add it to the list. When the program completes, sort and print the resulting words in alphabetical order.

```
fn=input("Enter the name of the file to be opened: ")
try:
    fh=open(fn)
except:
    print("Cannot open the ",fn, " file")
    exit()

lst=[]
for line in fh:
    for ln in line.split():
        if ln not in lst:
            lst.append(ln)

print("Elements of the lst:\n", lst,"\nNumber of words in the list lst: ",
len(lst))
lst.sort();print("Sorted elements of the lst:\n",lst)
```

Output:

```
Enter the name of the file to be opened: Greet.txt
Elements of the lst:
 ['Hai', 'Hello', 'and', 'Good', 'Morning', 'Have', 'A', 'Nice', 'Day', 'Welcome',
'to', 'the', 'World', 'of', 'Python', 'Programming']
Number of words in the list lst:  16
Sorted elements of the lst:
['A', 'Day', 'Good', 'Hai', 'Have', 'Hello', 'Morning', 'Nice', 'Programming',
'Python', 'Welcome', 'World', 'and', 'of', 'the', 'to']
```

3. Write a program to read through the mail box data and when you find line that starts with “From”, you will split the line into words using the split function. We are interested in who sent the message, which is the second word on the From line.

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

You will parse the From line and print out the second word for each From line, then you will also count the number of From (not From:) lines and print out a count at the end.
This is a good sample output with a few lines removed:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu
......
ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu

There were 27 lines in the file with From as the first word
```

4. Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters “done”. Write the program to store the numbers the user enters in a list and use the max() and min() functions to compute the maximum and minimum numbers after the loop completes.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

```
nNum=[ ]
print("Enter only integer values or 'done' to stop: ")

while True:
    num=input()
    if num=='done':
        break

    nNum.append(num)

print("Elements of the list are: ",nNum)
print("Maximum element in the list is: ",max(nNum))
print("Minimum element in the list is: ",min(nNum))
```

Output:

```
Enter only integer values or 'done' to stop:
Enter a number: 6
Enter a number: 5
Enter a number: 7
Enter a number: 4
Enter a number: 8
Enter a number: 3
Enter a number: 9
Enter a number: 2
Enter a number: 0
Enter a number: 1
Enter a number: done
Elements of the list are:  [6, 5, 7, 4, 8, 3, 9, 2, 0, 1]
```

```
Maximum element in the list is:  9
Minimum element in the list is:  0
```

5. Write a program to create a list of Fibonacci numbers for a given range N.

```
fibno=int(input("Enter a value for n: "))
fibLt=[ ]
a, b = 0, 1
if fibno>=0:
    fibLt.append(a)
    if fibno>=1:
        while b <= fibno:
            fibLt.append(b)
            a, b = b, a+b

print("Fibonacci numbers upto ",fibno ," are",fibLt)
```

6. Write a program to read N numbers and store them in a list and print.

```
nNum=[ ]
n=int(input("Enter a value for n: "))
print("Enter ",n, " numbers: ")
i=0
while i<n:
      num=int(input())
      nNum.append(num)
      i+=1

print("Elements of the list are: ",nNum)
```

7. Create one List of N strings and another list with M numbers. Merge both the lists. Sort the merged list and print.
8. Given a data set of airfoil self noise, find the average scaled sound pressure for a given range of angle of attack. (Big data analytics problem).
9. Create function which takes an array of numbers and returns average. Write a program to read N numbers to find average.

```
def numAvg(n):
    return sum(n)/len(n)

nNum=[]
n=int(input("Enter a value for n: "))
```

```
print("Enter ",n, " numbers: ")
i=0
while i<n:
      num=int(input())
      nNum.append(num)
      i+=1

print("Elements of the list are: ",nNum)
Avg=numAvg(nNum)
print("Average value of elements in the list is: ",Avg)
```

10. Write a swap function which swaps two array elements.

```
def swapArr(a,b):
    a,b=b,a
    return a,b

l1=[2,4,6,[8]]
l2=[1,3,5,[7,9]]

print("Before Swapping:\nL1 elements: ",l1,"\nL2 elements: ",l2)
l1,l2=swapArr(l1,l2)
print("After Swapping:\nL1 elements: ",l1,"\nL2 elements: ",l2)
```

11. Write selection sort algorithm.

```
a = [16, 19, 11, 15, 10, 12, 14]
i = 0
while i<len(a):
         #smallest element in the sublist
          smallest = min(a[i:])
         #index of smallest element
         small_index = a.index(smallest)
         #swapping
          a[i],a[small_index] = a[small_index],a[i]
          i=i+1

print (a)
```

Python Dictionaries

What is a Collection?

- A collection is nice because we can put more than one value in them and carry them all around in one convenient package.
- We have a bunch of values in a single “variable”
- We do this by having more than one place “in” the variable.
- We have ways of finding the different places in the variable

What is not a “Collection”?

- Most of our variables have one value in them - when we put a new value in the variable - the old value is over written

Eg:

```
>>> x = 2
>>> x = 4
>>> print (x )
4
```

A Story of Two Collections..

- **List** - A linear collection of values that stay in order
- **Dictionary** - A “bag” of values, each with its own label

Dictionaries

- Dictionaries are Python’s most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - ✓ Associative Arrays - Perl / Php
  - ✓ Properties or Map or HashMap - Java
  - ✓ Property Bag - C# / .Net
- Lists index their entries based on the position in the list
- Dictionaries are like bags - no order
- So we index the things we put in the dictionary with a “lookup tag”

Syntax:

dictionary_variable = {key1 : value1, key2 : value2,… }

- key is separated from value using colon (:),
- (key, value) pairs are separated using comma (,)
- (key, value) pair is called item

Eg:

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print (purse)
{'money': 12, 'tissues': 75, 'candy': 3}
>>> print (purse['candy'])
3
>>> purse['candy'] = purse['candy'] + 2'
>>> print (purse)
{'money': 12, 'tissues': 75, 'candy': 5}
```

Comparing Lists and Dictionaries

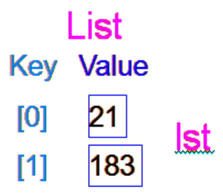
- Dictionaries are like Lists except that they use keys instead of numbers to look up values

Eg:

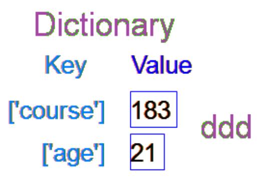
```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print (lst[21, 183])
>>> lst[0] = 23
>>> print (lst[23, 183])
>>> ddd = dict()
```

```
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print (ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print (ddd)
{'course': 182, 'age': 23}
```

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print (lst)
[21, 183]
>>> lst[0] = 23
>>> print (lst)
[23, 183]
```



```
>>> ddd = dict()
    >>> ddd['age'] = 21
    >>> ddd['course'] = 182
    >>> print (ddd)
    {'course': 182, 'age': 21}
    >>> ddd['age'] = 23
    >>> print (ddd)
    {'course': 182, 'age': 23}
```



Dictionary Literals (Constants)

- Dictionary literals use curly braces and have a list of key : value pairs
- Can also make an empty dictionary using empty curly braces

Eg:

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print (jjj)
{'jan': 100, 'chuck': 1, 'fred': 42}
>>> ooo = { }
>>> print (ooo)
{}
```

Dictionary as a Set of Counters

- One common use of dictionary is counting how often we "see" something.

To count the frequency of each letter in a given a string, using dictionary:

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

Output:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

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

Using get():

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

Output:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

```
>>> ccc = dict()
>>> ccc['csev'] = 1
>>> ccc['cwen'] = 1
>>> print (ccc)
```

```
{'csev': 1, 'cwen': 1}
>>> ccc['cwen'] = ccc['cwen'] + 1
>>> print (ccc)
{'csev': 1, 'cwen': 2}
```

Dictionary Tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the in operator to see if a key is in the dictionary

```
>>> ccc = dict()
>>> print (ccc['csev'])
Traceback (most recent call last):   File "<stdin>", line 1, in <module> KeyError: 'csev'
>>> print ('csev' in ccc)
False
```

When we see a new name

- When we encounter a new name, we need to add a new entry in the dictionary and if this the second or later time we have seen the name, we simply add one to the count in the dictionary under that name

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
    if name not in counts:
        counts[name] = 1
    else :
        counts[name] = counts[name] + 1
print (counts)
```

Output:

```
{'csev': 2, 'zqian': 1, 'cwen': 2}
```

The get method for dictionaries

- This pattern of checking to see if a key is already in a dictionary and assuming a default value if the key is not there is so common, that there is a method called **get**() that does this for us

```
if name in counts:
    x = counts[name]
else :
    x = 0
```

- Default value if key does not exist (and no Traceback).

```
x = counts.get(name, 0)
```

Output:

```
'csev': 2, 'zqian': 1, 'cwen': 2
```

Simplified counting with get()

- We can use **get**() and provide a default value of zero when the key is not yet in the dictionary - and then just add one

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
   counts[name] = counts.get(name, 0) + 1
print (counts)
```

Output:

```
'csev': 2, 'zqian': 1, 'cwen': 2
```

Dictionaries and files

- Common use of a dictionary is to count the occurrence of words in a file with some written text.

We write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary. Among the two for loops, the outer loop reads the lines of the file and the inner loop iterates through each of the words on that particular line. This is an example of a pattern called *nested loops* because one of the loops is the *outer* loop and the other loop is the *inner* loop.

The combination of the two nested loops ensures that every word on every line of the input file is being counted.

```
fname = input('Enter the file name: ')
try:
        fhand = open(fname)
except:
        print('File cannot be opened:', fname)
        exit()
counts = dict()
for line in fhand:
        words = line.split()
        for word in words:
                if word not in counts:
                        counts[word] = 1
                else:
                        counts[word] += 1
print(counts)
```

Output:

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1, 'is': 3, 'through': 1, 'pale': 1, 'yonder': 1, 'what': 1, 'sun':
2, 'Who': 1, 'But': 1, 'moon': 1, 'window': 1, 'sick': 1, 'east': 1, 'breaks': 1, 'grief': 1, 'with': 1, 'light':
1, 'It': 1, 'Arise': 1, 'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

Counting Pattern

- The general pattern to count the words in a line of text is to split the line into words, then loop through the words and use a dictionary to track the count of each word independently.

Counting Words

```
counts = dict()
print ('Enter a line of text:’)
line = input(' ')
words = line.split()
print ('Words:', words)
print ('Counting...’ )
for word in words:
   counts[word] = counts.get(word,0) + 1
print ('Counts', counts)
```

Output:

```
python wordcount.py
Enter a line of text:the clown ran after the car and the car ran into the tent and the tent fell down
on the clown and the car
Words: ['the', 'clown', 'ran', 'after', 'the', 'car', 'and', 'the', 'car', 'ran', 'into', 'the', 'tent', 'and', 'the',
'tent', 'fell', 'down', 'on', 'the', 'clown', 'and', 'the', 'car']
Counting...
Counts {'and': 3, 'on': 1, 'ran': 2, 'car': 3, 'into': 1, 'after': 1, 'clown': 2, 'down': 1, 'fell': 1, 'the': 7,
'tent': 2}
```

Definite Loops and Dictionaries

- Even though dictionaries are not stored in order, we can write a for loop that goes through all the entries in a dictionary - actually it goes through all of the keys in the dictionary and looks up the values

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

- If we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
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, so use the index operator to retrieve the corresponding value for each key.

```
Output:
jan 100
annie 42
```

- We see only the entries with a value above 10.
- To print the keys 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])

Output:
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

- First you see the list of keys got from the **keys** method is in unsorted order. Then the key-value pairs is in order from the for loop.

Retrieving lists of Keys and Values

- You can get a list of keys, values or items (both) from a dictionary

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print (list(jjj))
['jan', 'chuck', 'fred']
>>> print (jjj.keys())
['jan', 'chuck', 'fred']
>>> print (jjj.values())
[100, 1, 42]
>>> print (jjj.items())
```

```
[('jan', 100), ('chuck', 1), ('fred', 42)]
>>>
```

Bonus: Two Iteration Variables!

- We loop through the key-value pairs in a dictionary using *two* iteration variables
- Each iteration, the first variable is the key and the the second variable is the *corresponding* value for the key

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for aaa,bbb in jjj.items() :
...         print (aaa, bbb)
...
jan 100
chuck 1
fred 42
>>>
```

Advanced text parsing

The Python **split** function looks for spaces and treats words as tokens separated by spaces, we would treat the words "soft!" and "soft" as different words and create a separate dictionary entry for each word. Also since the file has capitalization, "who" and "Who" would be treated as different words with different counts.

Both the problems can be solved by using the string methods **lower**, **punctuation**, and **translate**. The **translate** is the most subtle of the methods.

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.

We will not specify the table but we will use the ***deletechars*** parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers "punctuation":

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

```
#Python program to find the frequency of words in a text file.
import string
```

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

Output:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2, 'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love':
24, 'cloak': 1, a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40, 'maiden': 1, 'whiteupturned':
1, 'juliet': 32, 'gentleman': 1, 'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Summary

- What is a collection?
- Lists versus Dictionaries
- Dictionary constants
- The most common word
- Using the get() method
- Hashing, and lack of order
- Writing dictionary loops
- Sneak peek: tuples
- Sorting dictionaries

Solve the following exercise questions:

1. Write a program that reads the words in words.txt and stores them as keys in a dictionary. It doesn’t matter what the values are. Use the in operator as a fast way to check whether a string is in the dictionary.
2. Program to count the occurrences of letters in a given word.
3. Program to count the frequency of words in a given text file.
4. Write a Python program to count the same values associated with key in a dictionary.

5. Given a data set of airfoil self noise, find the average scaled sound pressure for each angle of attack. (Use dictionary - Big data analytics problem).
6. Create a dictionary by merging two lists having keys and values.
7. Write a python program to merge TWO dictionaries.
8. Write a program to display the distinct words in a file with length of word and frequency.
9. Write a program that reads the words in words.txt and stores them as keys in a dictionary. It doesn’t matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.
10. Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines that start with “From”, then look for the third word and keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).
Sample Line:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Sample Execution:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

11. Write a program to read through a mail log, build a histogram using a dictionary to count how many messages have come from each email address, and print the dictionary.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

12. Add code to the above program to figure out who has the most messages in the file. After all the data has been read and the dictionary has been created, look through the dictionary using a maximum loop (see Section [maximumloop]) to find who has the most messages and print how many messages the person has.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
Enter a file name: mbox.txt
zqian@umich.edu 195
```

13. This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e., the whole email address). At the end of the program, print out the contents of your dictionary.

```
python schoolcount.py
```

```
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

Tuples

Tuples are like lists

- Tuples are another kind of sequence that function much like a list - they have elements which are indexed starting at 0

Syntax:

```
    tuple_variable = (value1, [value2,…, valuen-1 ], valuen)
```

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 also comparable and hashable so we can sort lists of them and use tuples 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')
```

Creating tuple:

Creating empty tuple:

```
>>> tup=tuple( ) # with no argument, create an empty tuple using the built-in function tuple( )
>>> type(tup)
<class 'tuple'>
>>> print(tup)
()
>>> tups=( )
>>> type(tups)
<class 'tuple'>
```

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

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

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

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

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')
```

Avoid using **tuple** as a variable name, as **tuple** is the name of a constructor.
Most list operators also work on tuples.

```
>>> t = ('a', 'b', 'c', 'd', 'e')      # bracket operator indexes an element
>>>print(t[0])
'a'
>>> print(t[1:3])      # slice operator selects a range of elements
('b', 'c')
```

Cannot modify the elements of a tuple, but can replace one tuple with another:

```
>>> t[0] = 'A'          # error if we try to modify one of the elements of the tuple
TypeError: object doesn't support item assignment

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

Example:

```
>>> x = ('Glenn', 'Sally', 'Joseph')
>>> print (x[2])
Joseph
>>> y = ( 1, 9, 2 )
>>> print (y)
(1, 9, 2)
>>> print (max(y))
9
>>> for iter in y:
...     print (iter)
...
1
9
2
```

Additional examples for Tuples are "immutable"

- Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

Example:

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print (x)
[9, 8, 6]

>>> y = 'ABC’
>>> y[2] = 'D’
Traceback:'str' object does not support item Assignment

>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple' object does not support item Assignment
```

Things not to do with tuples

Example:

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:AttributeError: 'tuple' object has no attribute 'sort’
>>> x.append(5)
Traceback:AttributeError: 'tuple' object has no attribute 'append’
>>> x.reverse()
Traceback:AttributeError: 'tuple' object has no attribute 'reverse’
```

A Tale of Two Sequences

Example:

```
>>> l = list()
>>> dir(l)[
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> t = tuple()
>>> dir(t)
['count', 'index']
```

Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- So in our program when we are making "temporary variables" we prefer tuples over lists.

Tuples and Assignment

- We can also put a tuple on the left hand side of an assignment statement
- We can even omit the parenthesis

Example:

```
>>> (x, y) = (4, 'fred')
>>> print (y)
Fred
>>> (a, b) = (99, 98)
>>> print (a)
99
```

Tuples and Dictionaries

- The items() method in dictionaries returns a list of (key, value) tuples

Example:

```
>>> d = dict()
>>> d['csev'] = 2
>>> d['cwen'] = 4
>>> for (k,v) in d.items():
...     print (k, v)
...
csev 2
cwen 4
>>> tups = d.items()
>>> print (tups)
[('csev', 2), ('cwen', 4)]
```

Tuples are Comparable

- The comparison operators work with tuples and other sequences If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

Example:

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

Sorting Lists of Tuples

- We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary
- First we sort the dictionary by the key using the items() method

Example:

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

Using sorted()

- We can do this even more directly using the built-in function sorted that takes a sequence as a parameter and returns a sorted sequence

Example:

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

```
>>> for k, v in sorted(d.items()):
...         print (k, v)
...
a 10
b 1
c 22
```

Sort by values instead of key

- If we could construct a list of tuples of the form (value, key) we could sort by value
- We do this with a for loop that creates a list of tuples

Example:

```
>>> c = {'a':10, 'b':1, 'c':22}
>>> tmp = list()
>>> for k, v in c.items() :
        ...     tmp.append( (v, k) )
        ...
>>> print (tmp)
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> tmp.sort(reverse=True)
>>> print (tmp)
[(22, 'c'), (10, 'a'), (1, 'b')]
```

```
fhand = open(LongIntro.txt')
counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0 ) + 1
lst = list()
for key, val in counts.items():
    lst.append( (val, key) )
lst.sort(reverse=True)
```

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

Output:
the 226
to 204
a 165
and 160
you 130
is 112
of 103
Python 85
that 68
in 66
```

The top 10 most common words.

Even Shorter Version (adv)

List comprehension creates a dynamic list. In this case, we make a list of reversed tuples and then sort it.

Example:

```
>>> c = {'a':10, 'b':1, 'c':22}
>>> print (sorted( [ (v,k) for k,v in c.items() ] ))
[(1, 'b'), (10, 'a'), (22, 'c')]
```

Summary

- Tuple syntax
- Mutability (not)
- Comparability
- Sortable
- Tuples in assignment statements
- Using sorted()
- Sorting dictionaries by either key or value

Solve the following exercise questions:

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 sort the list in reverse order and print out the person who has the most commits.

Sample Line:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
```

```
zqian@umich.edu 195
```

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 4
17 2
18 1
19 1
```

3. Write a program that reads a file and prints the letters in decreasing order of frequency. Your program should convert all the input to lower case and only count the letters a-z. Your program should not count spaces, digits, punctuation, or anything other than the letters a-z. Find text samples from several different languages and see how letter frequency varies between languages.

Regular Expressions

Regular Expressions

- In computing, a regular expression, also referred to as "regex" or "regexp", provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor.
- Really clever "wild card" expressions for matching and parsing strings.

Understanding Regular Expressions

- Very powerful and quite cryptic
- Fun once you understand them
- Regular expressions are a language unto themselves
- A language of "marker characters" - programming with characters
- It is kind of an "old school" language - compact

Regular Expression Quick Guide

^	Matches the beginning of a line
$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace character
*****	Repeats a character zero or more times
***?**	Repeats a character zero or more times (non-greedy)
+	Repeats a chracter one or more times
+?	Repeats a character one or more times (non-greedy)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(	Indicates where string extraction is to start
)	Indicates where string extraction is to end

The Regular Expression Module

- Before using regular expressions in your program, import the library using "**import re**"
- Use **re.search()** to see if a string matches a regular expression similar to using the find() method for strings
- Use **re.findall()** extract portions of a string that match your regular expression similar to a combination of **find()** and slicing: **var[5:10]**

Using re.search() like find()

```
hand = open('mbox-short.txt')
for line in hand:
   line = line.rstrip()
   if line.find('From:') >= 0:
      print line
```

```
import re
hand = open('mbox-short.txt')
for line in hand:
         line = line.rstrip()
         if re.search('From:', line) :
               print (line)
```

Using re.search() like startswith()

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

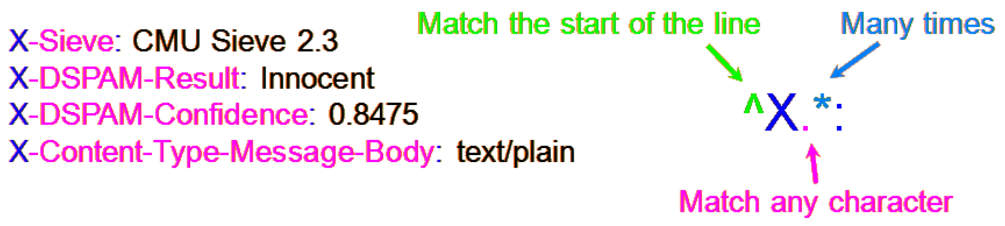
```
import re
hand = open('mbox-short.txt')
for line in hand:
   line = line.rstrip()
   if re.search('^From:', line) :
         print (line)
```

We fine-tune what is matched by adding special characters to the string.

Wild-Card Characters

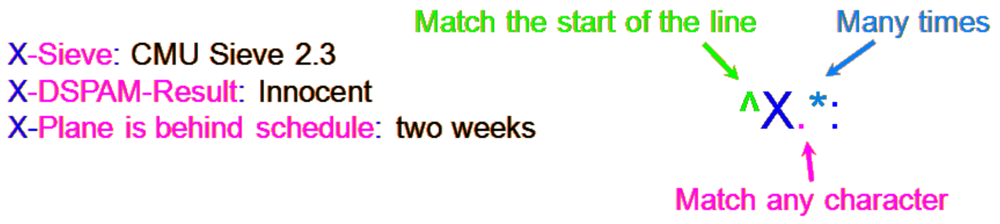
- The **dot** character matches any character
- If you add the **asterisk** character, the character is "any number of times"

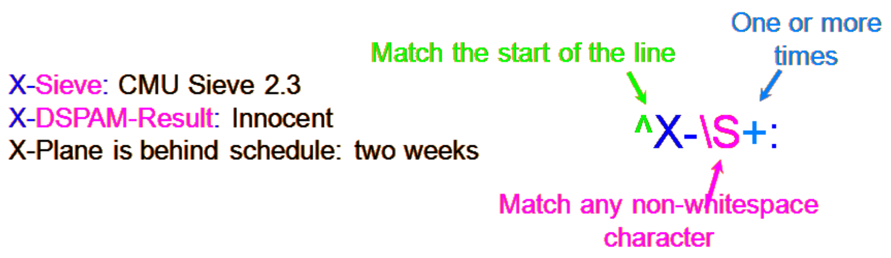
Example:


X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body: text/plain
Match the start of the line
Many times
^X.*:
Match any character

Fine-Tuning Your Match

- Depending on how "clean" your data is and the purpose of your application, you may want to narrow your match down a bit




X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-Plane is behind schedule: two weeks
Match the start of the line
One or more times
^X-\S+:
Match any non-whitespace character

Matching and Extracting Data

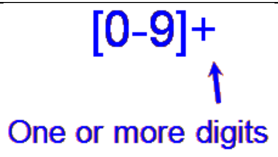
- The **re.search()** returns a True/False depending on whether the string matches the regular expression
- If we actually want the matching strings to be extracted, we use **re.findall()**

Example:

```
>>> import re
>>> x = 'My 2 favorite numbers are 51 and 42'
```

```
>>> y = re.findall('[0-9]+',x)
>>> print (y)
```

```
['2', '51', '42']
```



- When we use **re.findall()** it returns a list of zero or more sub-strings that match the regular expression

Example:

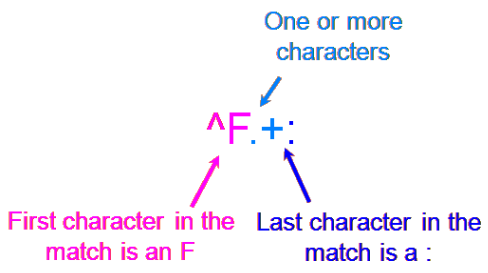
```
>>> import re
>>> x = 'My 2 favorite numbers are 51 and 42'
>>> y = re.findall('[0-9]+',x)>>> print y['2', '51', '42']
>>> y = re.findall('[AEIOU]+',x)
>>> print (y)
[ ]
```

Warning: Greedy Matching

- The repeat characters (* and +) push outward in both directions (greedy) to match the largest possible string

Example:

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+:', x)
>>> print (y)
['From: Using the :']
```

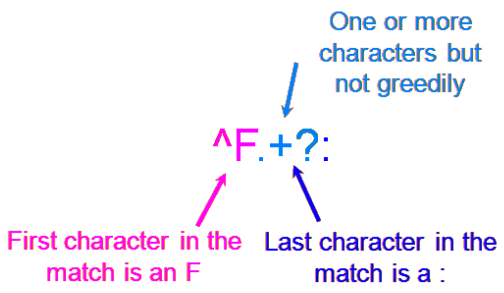

One or more characters
^F.+:
First character in the match is an F
Last character in the match is a :

Non-Greedy Matching

- Not all regular expression repeat codes are greedy!

Example:

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print (y)
['From:']
```

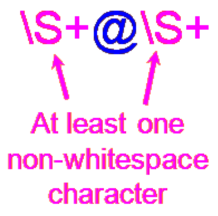

One or more characters but not greedily
^F.+?:
First character in the match is an F
Last character in the match is a :

Fine Tuning String Extraction

- You can refine the match for **re.findall()** and separately determine which portion of the match that is to be extracted using parenthesis

Example:

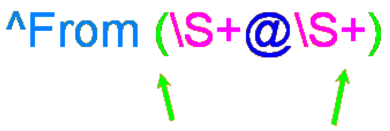
```
From stephen.marquard@uct.ac.za Sat Jan  5
09:14:16 2008
>>> y = re.findall('\S+@\S+',x)
>>> print (y)
['stephen.marquard@uct.ac.za']
>>> y = re.findall('^From:.*? (\S+@\S+)',x)
>>> print (y)
['stephen.marquard@uct.ac.za']
```



- Parenthesis are not part of the match - but they tell where to start and stop, what string to extract

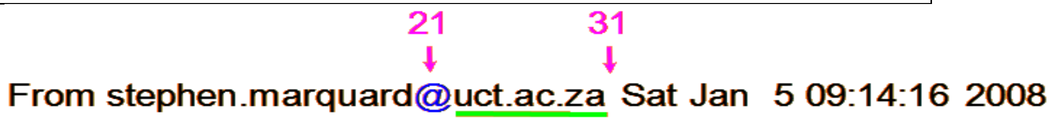
Example:

```
From stephen.marquard@uct.ac.za Sat Jan  5
09:14:16 2008
>>> y = re.findall('\S+@\S+',x)
>>> print (y)
['stephen.marquard@uct.ac.za']
>>> y = re.findall('^From (\S+@\S+)',x)
>>> print (y)
['stephen.marquard@uct.ac.za']
```



Extracting a host name - using find and string slicing.
Example:

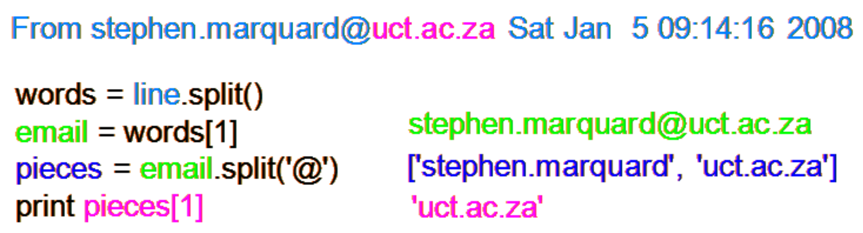
```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print (atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print (sppos)
31
>>> host = data[atpos+1 : sppos]
>>> print (host)
uct.ac.za
```



The Double Split Version

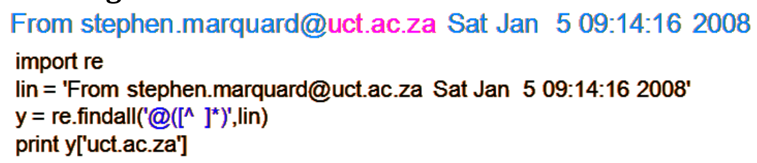
- Sometimes we split a line one way and then grab one of the pieces of the line and split that piece again

Example: From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

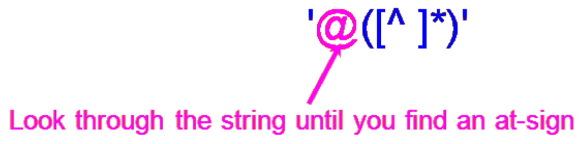


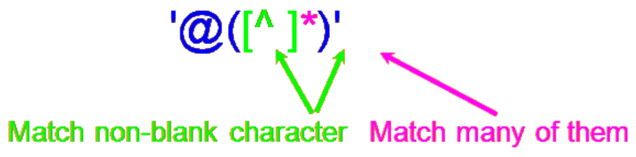
```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

words = line.split()
email = words[1]                stephen.marquard@uct.ac.za
pieces = email.split('@')       ['stephen.marquard', 'uct.ac.za']
print pieces[1]                 'uct.ac.za'
```

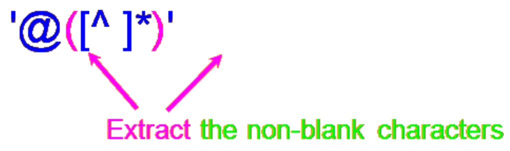
The Regex Version



```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
import re
lin = 'From stephen.marquard@uct.ac.za  Sat Jan  5 09:14:16 2008'
y = re.findall('@([^ ]*)',lin)
print y['uct.ac.za']
```







Even Cooler Regex Version

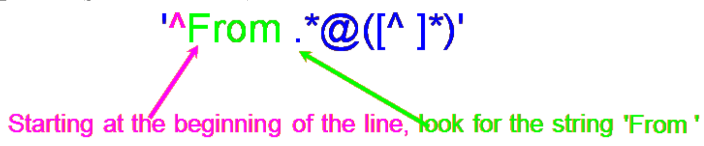
Example: From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

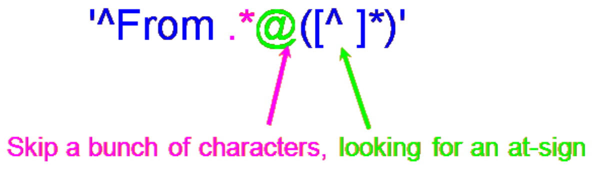
import re

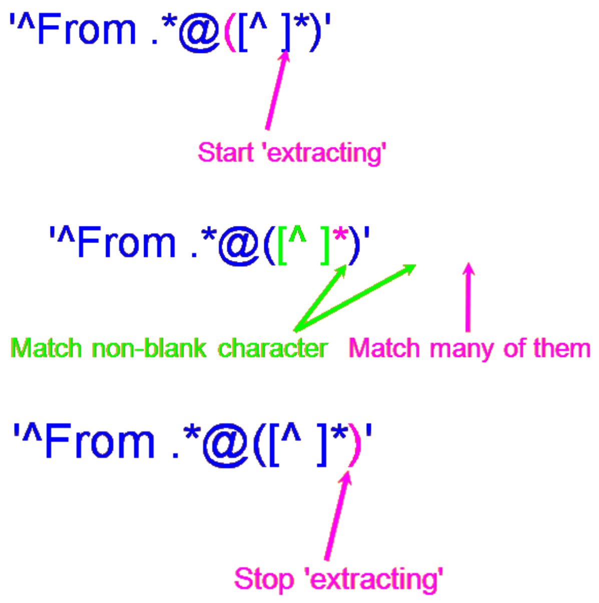
lin = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'

y = **re.findall**('^From .*@([^]*)',lin)

print (y['uct.ac.za'])


'^From .*@([^]*)'
Starting at the beginning of the line, look for the string 'From '





Spam Confidence

```
import re
hand = open('mbox-short.txt')
numlist = list()
for line in hand:
    line = line.rstrip()
    stuff = re.findall('^X-DSPAM-Confidence: ([0-9.]+)', line)
    if len(stuff) != 1 :  continue
    num = float(stuff[0])
    numlist.append(num)
print ('Maximum:', max(numlist))
```

Output:

```
python ds.py
Maximum: 0.9907
```

Regular Expression Quick Guide

^	Matches the beginning of a line
$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace character
*	Repeats a character zero or more times
*?	Repeats a character zero or more times (non-greedy)
+	Repeats a chracter one or more times

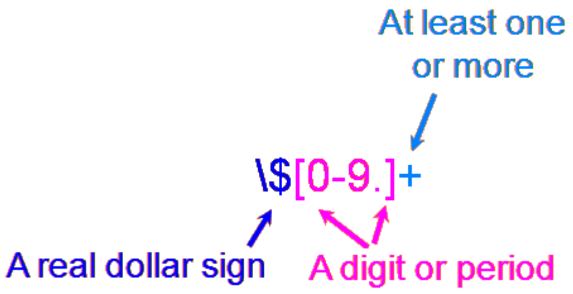
+?	Repeats a character one or more times (non-greedy)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(	Indicates where string extraction is to start
)	Indicates where string extraction is to end

Escape Character

- If you want a special regular expression character to just behave normally (most of the time) you prefix it with '\'

Example:

```
>>> import re
>>> x = 'We just received $10.00 for cookies.'
>>> y = re.findall('\$[0-9.]+',x)
>>> print (y)
['$10.00']
```



Summary

- Regular expressions are a cryptic but powerful language for matching strings and extracting elements from those strings
- Regular expressions have special characters that indicate intent

Solve the following exercise questions:

1. Write a simple program to simulate the operation of the grep command on Unix. Ask the user to enter a regular expression and count the number of lines that matched the regular expression:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^Xmbox.txt
had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

2. Write a program to look for lines of the form
`New Revision: 39772`

and extract the number from each of the lines using a regular expression and the findall() method. Compute the average of the numbers and print out the average.

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```