Python – S5

Contents

List

Lists

- 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.
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

[10, 20, 30, 40]
['crunchy frog', 'ram sunder', 'Hi there!']

- The first example is a list of four integers.
- The second is a list of three strings.
- The elements of a list don't have to be the same type.
- The following list contains a string, a float, an integer, and (lo!) another list:

['python', 2.0, 5, [10, 20]]

Lists

- A list within another list is nested.
- A list that contains no elements is called an empty list; you can create one with empty brackets, [].
- As you might expect, you can assign list values to variables:

```
>>> friends = [’Nirmalya', ’Subir', 'Goutam']
>>> numbers = [17, 123]
>>> empty = []
>>> print (friends, numbers, empty )
[’Nirmalya', ’Subir', 'Goutam’] [17, 123] []
```

Lists

Lists are Mutable

- 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. Remember that the indices start at 0:

```
>>> print friends[0]
Nirmalya
```

- Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list.
- When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

- The one-eth element of numbers, which used to be 123, is now 5.

Lists

Lists are Mutable

- You can think of a list as 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 you try to read or write an element that does not exist, you get an IndexError.
  - If an index has a negative value, it counts backward from the end of the list.
- The in operator also works on lists.

```
>>> friends = [’Nirmalya', ’Subir', 'Goutam']
>>> ’Subir' in friends
True
>>> 'Boss' in friends
False
```

Lists

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 friend in friends:
    print (friend)
```

- This works well if you only need to read the elements of the list.

- But if you want to write or update the elements, you need the indices.
- A common way to do that is to combine the functions range and len:

```
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.

Lists

Traversing a 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:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Lists

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]
```

- The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

Lists

List Slicing

- 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 you omit the first index, the slice starts at the beginning.
- If you omit the second, the slice goes to the end.
- So if you omit both, the slice is a copy of the whole list.

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

Lists

List Slicing

- 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']
```

Lists

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:

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

- Please note : The above example leaves t2 unmodified.

Lists

List Methods

- ‘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.

- If you accidentally write t = t.sort(), you will be disappointed with the result. [Incidentally it returns ‘None’ !!!]

Lists

- Generally, any method that mutates an object has no return value.
- Methods like append(), extend(), sort(), and reverse() have no return value.
- They adjust the structure of the list object itself.

- The append() method does not return a value.
- It mutates the list object.

- It's surprisingly common to see wrong code like this:
  a = ['some', 'data']
  a = a.append('more data')

- This is emphatically wrong. This will set a to None.
- The correct approach is a statement like this, without any additional assignment:
  a.append('more data')

Lists

Deleting list elements

- There are several ways to delete elements from a list.
- If you know the index of the element you want, you can use pop:

```
>>> 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 you don’t provide an index, it deletes and returns the last element.

Lists

Deleting list elements

- If you don't need the removed value, you can use the del operator:

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

- If you know the element you want to remove (but not the index), you can use remove:

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

- The return value from remove is None.

Lists

Deleting list elements

- To remove more than one element, you can use del 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.

Lists

Reversing list elements

- Say this is our list :

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>t.reverse()
>>>t
>>> [‘f’, ‘e’, ‘d’, ‘c’, ‘b’, ‘a’ ]
```

Note : As mentioned, reverse() acts on the original list [i.e. inplace !!]. So no new lists are created.

Lists

Lists and Functions

- There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> 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
```

- The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable. You cannot mix types.

Lists

Lists and Functions

- A small program : [to compute average]

```
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 )
```

- We make an empty list before the loop starts, and then each time we have a number, we append it to the list.
- At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

Lists

Lists and Strings

- A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string.
- To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print (t)
['s', 'p', 'a', 'm']
```

- Because list is the name of a built-in function, you should avoid using it as a variable name.
- The list function breaks a string into individual letters.
- If you want to break a string into words, you can use the split method:

```
>>> s = ’I am a good boy'
>>> t = s.split()
>>> print (t)
[’I', ’am', ’a', ’good’, ‘boy’]
```

Lists

Lists and Strings

- You can then also use:

print (t[2])

A

- Once you have used split to break the string into a list of tokens, you can use the index operator (square bracket) to look at a particular word in the list.
- You can call split with an optional argument called a delimiter specifies which characters to use as word boundaries.
- The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

Lists

Lists and Strings

- join is the inverse of split.
- It takes a list of strings and concatenates the elements.
- join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = [’clapping', 'for', 'the', ’team']
>>> delimiter = ' '
>>> delimiter.join(t)
’clapping for the team'
```

- In this case the delimiter is a space character, so join puts a space between words.
- To concatenate strings without spaces, you can use the empty string, '', as a delimiter.

- Note : these are very useful – split and join

Lists

Parsing Lines

- Usually when we are reading a file we want to do something to the lines other than just printing the whole line.
- Often we want to find the “interesting lines” and then parse the line to find some interesting *part* of the line.
- What if we wanted to print out the day of the week from those lines that start with “From ”.

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

- The split method is very effective when faced with this kind of problem.
- We can write a small program that looks for lines where the line starts with “From ” and then split those lines and then print out the third word in the line:

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

Lists

List Arguments

- When you pass a list to a function, the function gets a reference to the list.
- If the function modifies a list parameter, the caller sees the change.
- For example, delete_head removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

- Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print (letters)
['b', 'c']
```

- The parameter t and the variable letters are aliases for the same object.

Lists

List Arguments

- It is important to distinguish between operations that modify lists and operations that create new lists.
- For example, the append method modifies a list, but the + operator creates a new list:

```
>>>t1=[1 , 2]
>>> t2 = t1.append(3)
>>> print (t1)
[1, 2, 3]
>>> print (t2)
None
>>>t3 = t1+ [3]
>>> print (t3)
[1, 2, 3]
>>>t2  is  t3
False
```

- This difference is important when you write functions that are supposed to modify lists.

Dictionaries

Dictionaries

- A dictionary is like a list, but more general.
- In a list, the positions (a.k.a. indices) have to be integers; in a dictionary the indices can be (almost) any type.
- You can think of a dictionary 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.
- Dictionaries are unordered set of *key: value* pairs where keys are unique.
- We declare dictionaries using {} braces.
- We use dictionaries to store data for any particular key and then retrieve them.

- The function ‘dict’ creates a new dictionary with no items.
- Because ‘dict’ is the name of a built-in function, you should avoid using it as a variable name.

Dictionaries

```
>>> data = {'kushal':'Fedora', 'kartik':'Debian', ’raman':'Mac'}
>>> data
{'kushal': 'Fedora', ’raman': 'Mac', 'kartik': 'Debian'}
>>> data['kartik']
'Debian'
```

- To add items to the dictionary, you can use square brackets:

```
>>> data['partha'] = 'Ubuntu’
>>> data
{'kushal': 'Fedora', ’raman': 'Mac', 'partha': 'Ubuntu’, 'kartik': 'Debian’}
```

- The order of the key-value pairs is not the same.
- In fact, if you type the same example on your computer, you might get a different result.
- In general, the order of items in a dictionary is unpredictable.
- But that’s not a problem because the elements of a dictionary are never indexed with integer indices.

Dictionaries

- To check if any *key* is there in the dictionary or not you can use *in* keyword.

```
>>> 'Soumya' in data
False
```

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

```
>>> len(data)
4
```

- To delete any particular *key:value* pair

```
>>> del data['kushal']
>>> data
{’raman': 'Mac', 'kartik': 'Debian', 'partha': 'Ubuntu' }
```

Dictionaries

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

```
>>> vals = data.values()
>>> 'Ubuntu' in vals
True
```

- You must remember that no mutable object can be a *key*, that means you can not use a *list* as a *key*.
- *dict()* can create dictionaries from tuples of *key,value* pair.

Dictionaries

- If you want to loop through a dict use *items()* method.

```
>>> data
{'Kushal': 'Fedora', ’raman': 'Mac', 'kartik': 'Debian', 'partha': 'Ubuntu'}
>>> for x, y in data.items():
...         print("%s uses %s" % (x, y))
...
```

Kushal uses Fedora
raman uses Mac
kartik uses Debian
partha uses Ubuntu

Dictionaries

- 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.

- I won't explain why hash functions are so magical, but you can read more about it at wikipedia.org/wiki/Hash_table.

Dictionaries

Dictionary as a set of counters

- Suppose you are given a string and you want to count how many times each letter appears.
- There are several ways you could do it:
  - You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
  - You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function 'ord'), use the number as an index into the list, and increment the appropriate counter.
  - You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.
- Each of these options performs the same computation, but each of them implements that computation in a different way.

Dictionaries

Dictionary as a set of counters

- An implementation is a way of performing a computation; some implementations are better than others.
- 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)
```

Dictionaries

Dictionary as a set of counters

- We are effectively computing a histogram, which is a statistical term for a set of counters (or frequencies).
- The for loop traverses the string.
- Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once).
- If c is already in the dictionary we increment d[c].
- Here's the output of the program:

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

- The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.

- Also note that the output, i.e. the keys are NOT sorted

Dictionaries

Dictionary as a set of counters

- 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
```

Dictionaries

Dictionary as a set of counters

- 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)
```

- The use of the get method to simplify this counting loop ends up being a very commonly used “idiom” in Python.
- So you should take a moment and compare the loop using the if statement and in operator with the loop using the get method.
- They do exactly the same thing, but one is more succinct.

Dictionaries

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.
- Let's start with a very simple file of words taken from the text of *Romeo and Juliet* thanks to http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html.
- For the first set of examples, we will use a shortened and simplified version of the text with no punctuation.
- Later we will work with the text of the scene with punctuation included.

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

- We will 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.

Dictionaries

Dictionaries and Files

- You will see that we have two ‘for’ loops.
- The outer loop is reading the lines of the file and the inner loop is iterating 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.
- Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating “more quickly” and the outer loop as iterating more slowly.
- The combination of the two nested loops ensures that we will count every word on every line of the input file.

Dictionaries

Dictionaries and Files

```
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)
```

Dictionaries

Dictionaries and Files

- When we run the program, we see a raw dump of all of the counts in unsorted hash order. (the romeo.txt file is available at www.py4inf.com/code/romeo.txt)

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}

- The output is a little inconvenient to read, but it is correct.

Dictionaries

Looping and Dictionaries

- If you use a dictionary as the sequence in a 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.

Dictionaries

Looping and Dictionaries

- We can use this pattern to implement the various loop idioms that we have described earlier.
- For example 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 we must use the index operator to retrieve the corresponding *value* for each key.
- Here's what the output looks like:

```
jan 100
annie 42
```

- We see only the entries with a value above 10.

Dictionaries

Looping and Dictionaries

- If you want to print the keys in alphabetical order, you 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 printing out key/value pairs in sorted order as follows as follows:

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

- The output looks like:

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

Dictionaries

Advanced Text Parsing

- In the previous example using the file 'romeo.txt', we made the file as simple as possible by removing any and all punctuation by hand.
- The real text 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, 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, we would treat "who" and "Who" as different words with different counts.

Dictionaries

Advanced Text Parsing

- 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:

string.translate(s, table[, deletechars])

- *Delete all characters from s that are in deletechars (if present), and then translate the characters using table, which must be a 256-character string giving the translation for each character value, indexed by its ordinal. If table is None, then only the character deletion step is performed.*

- We will not specify the table but we will use the deletechars parameter to delete all of the punctuation.

Dictionaries

Advanced Text Parsing

- We will even let Python tell us the list of characters that it considers "punctuation":

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

- Now we can make the following modifications to our earlier program:

- Please see next slide

Dictionaries

Advanced Text Parsing

```
import string                                          # New Code
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.translate(None, string.punctuation)    # New Code
        line = line.lower()                                # New Code
        words = line.split()
        for word in words:
                if word not in counts:
                        counts[word] = 1
                else:
                        counts[word] += 1
print (counts)
```

Dictionaries - Examples

```
# reverse compliment of DNA
# A DNA sequence contains only the letters A, C, G and T. (Each letter represents a small molecule,
# and a DNA sequence is a ``macromolecular'' chain of them.) Each letter in a DNA sequence is called
# a base, basepair, or nucleotide. Normally, DNA occurs as a double strand where each A is paired with # a T and vice versa, and each C is paired with a G and vice versa. The reverse complement of a DNA
# sequence is formed by reversing the letters, interchanging A and T and interchanging C and G.
# Thus the reverse complement of ACCTGAG is CTCAGGT.

def reverse_dna (my_sequence):
    my_dictionary = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return reversed(my_sequence)
def reverse_com_dna (my_sequence):
    my_dictionary = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return "".join([my_dictionary[base] for base in reversed(my_sequence)])
seq1 = 'CATGCCGGAATT'
seq2 = 'ACCTGAG'
print('Original Seq is : ', seq1 )
print('Reverse is : ', ''.join(reverse_dna(seq1)))
print('Reversed Complement Seq is : ',''.join(reverse_com_dna(seq1)))
print('Original Seq is : ', seq2 )
print('Reverse is : ', ''.join(reverse_dna(seq2)))
print('Reversed Complement Seq is : ',''.join(reverse_com_dna(seq2)))
```

Dictionaries - Examples

```
# In this example , you have to take number of students as input , then ask marks for three subjects as
# ‘Physics’, ‘Maths’, ‘History’, if the total marks for any student is less 120 then print he failed, or else say passed.
#!/usr/bin/env python3
n = int(input("Enter the number of students : "))
data = {} # here we will store the data
languages = ('Physics', 'Maths', 'History') #all languages
 for i in range(0, n): #for the n number of students
        name = input('Enter the name of the student %d: ' % (i + 1)) #Get the name of the student
        marks = []
        for x in languages:
                marks.append(int(input('Enter marks of %s : ' % x))) #Get the marks for languages
        data[name] = marks
for x, y in data.items():
        total = sum(y)
        print("%s 's total marks %d" % (x, total))
        if total < 120:
                print("%s failed :(" % x)
        else:
                print("%s passed :)" % x)
```

Tuples

Tuples

- 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')
```

Tuples

- 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.

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

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

Tuples

- 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, you should avoid using it 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')
```

Tuples

- But if you try to modify one of the elements of the tuple, you get an error:

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

- You can't modify the elements of a tuple, but you can replace one tuple with another:

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

- NOTE : Similarly you can do this for String as well.

Say we have : mystr = 'prasun'

We can then also have complete replacement as :

mystr = 'neogy'

Tuples

- The comparison operators work 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 )  <  (0 , 1 )
False
>>> (0 , 1 ) < ( 0 , 2 )
True
>>> ( 0, 1, 2000000 ) < ( 0, 3, 4 )
True
```

Tuples

- 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.

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

- Please see next slide

Tuples

```
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)
```

- 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.

Tuples

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

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

Tuples

Tuple Assignment

- One of the unique syntactic features of the Python language is the ability to have a tuple on the left hand side of an assignment statement.
- This allows you to assign more than one variable at a time when the left hand side is a sequence.
- In this example we have a two element list (which is a sequence) and assign 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’
```

Tuples

Tuple Assignment

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

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

- Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions.
- Each value on the right side is assigned to its respective variable on the left side.
- All the expressions on the right side are evaluated before any of the assignments.
- The number of variables on the left and the number of values on the right have to be the same.
-
- Note : Be VERY careful of such Tuple assignment statements!!! It can be sometimes misleading/confusing.

Tuples

Tuple Assignment

- 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
```

Tuples

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

- As you should expect from a dictionary, the items are in no particular order.

- However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples.

Tuples

Dictionaries and Tuples

- Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

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

>>> t1 = list(t)
>>> t1.sort()
>>> t1
[('a', 10), ('b', 1), ('c', 22)]
```

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

Tuples

Multiple Assignment with Dictionaries

- Combining items, tuple assignment and for, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in 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 in hash key order (i.e. no particular order).

Tuples

Multiple Assignment with Dictionaries

- If we combine these two techniques, we can print out the contents of a dictionary sorted by the *value* stored in each key/value pair.
- To do this, we first make a list of tuples where each tuple is (value, key).
- The items method would give us a list of (key, value) tuples–but this time we want to sort by value not key.
- Once we have constructed the list with the value/key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
>>>  d = {'a':10, 'b':1, 'c':22}
>>>  l = list()
>>>  for key, val in d.items() :
...    l.append( (val, key) )
...
```

NOTE : The double parentheses in the inner statement within the ‘for’ loop!!!

Tuples

Multiple Assignment with Dictionaries

```
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

- So, by carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

Tuples

An Example program [The most common words]

- Coming back to our previous example, we can augment our program to use this 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( (None, 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 counts.items():
        lst.append( (val, key) )
lst.sort(reverse=True)
for key, val in lst[:10] :
        print (key, val)
```

Tuples

An Example program [The most common words]

- The first part of the program which 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, we construct a list of (val, key) tuples and then sort the list in reverse order.
- Since the value is first, it will be used for the comparisons and 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 we write a nice for loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (lst[:10]).

Tuples

An Example program [The most common words]

- So now the output finally looks like what we want for our word frequency analysis.

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

- The fact that this complex data parsing and analysis can be done with an easy-to-understand 19 line Python program is one reason why Python is a good choice as a language for data mining !!!.

Tuples

- I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists.
- To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.
- In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably.
- So how and why do you choose one over the others?
- To start with the obvious, strings are more limited than other sequences because the elements have to be characters.
- They are also immutable.
- If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.
- Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

Tuples

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

- Because tuples are immutable, they don't provide methods like sort and reverse, which modify existing lists.
- However Python provides the built-in functions sorted and reversed, which take any sequence as a parameter and return a new list with the same elements in a different order.

Set

Set

- A set is, perhaps the simplest possible container, since it contains objects in no particular order with no particular identification.
- Objects stand for themselves. With a sequence, objects are identified by position. With a mapping, objects are identified by some key. With a set, objects stand for themselves.
- Since each object stands for itself; elements of a set cannot be duplicated.
- A list or tuple, for example, can have any number of duplicate objects. For example, the tuple (1, 1, 2, 3) has four elements, which includes two copies of the integer 1; if we create a set from this tuple, the set will only have three elements.
- A set has large number of operations for unions, intersections, and differences.
- A common need is to examine a set to see if a particular object is a member of that set, or if one set is contained within another set.
- A set is mutable, which means that it cannot be used as a key for a dict.
- In order to use a set as a dict key, we can create a frozenset, which is an immutable copy of a set. This allows us to accumulate a set of values to create a dict key.

Set

- A set value is created by using the set() or frozenset() factory functions.
- These can be applied to any iterable container, which includes any sequence, the keys of a dict, or even a file.

```
>>> set( ("hello", "world", "of", "words", "of", "world") )
{'words', 'hello', 'of', 'world'}
```

- Note that we provided a six-tuple sequence to the set() function, and we got a set with the four unique objects. The set is shown as a list literal, to remind us that a set is mutable.
- Set operations are:
- Union : the operator is '|'. The resulting set has elements from both source sets. An element is in the result if it is one set or the other.

```
>>> fib=set( (1,1,2,3,5,8,13) )
>>> prime=set( (2,3,5,7,11,13) )
>>> fib | prime
{1, 2, 3, 5, 7, 8, 11, 13}
>>>
```

Set

- Intersection. ‘&’. The resulting set has elements that are common to both source sets. An element is in the result if it is in one set and the other.

```
>>> fib & prime
{2, 3, 5, 13}
>>>
```

- Difference. ‘-’. The resulting set has elements of the left-hand set with all elements from the right-hand set removed. An element will be in the result if it is in the left-hand set and not in the right-hand set.

```
>>> fib - prime
{8, 1}
>>> prime - fib
{11, 7}
>>>
```

Set

- Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

```
>>> cities = frozenset(["Frankfurt", "Basel","Freiburg"])
>>> cities.add("Strasbourg")
Traceback (most recent call last):
File "<stdin>", line 1, in <module> AttributeError: 'frozenset' object has no attribute 'add'
>>>
```

- We can also have some built-in functions like :

```
>>> len( set( [1,1,2,3] ) )
3
>>> max( set([1,1,2,3,5,8] ) )
8
>>> sum(set( [1,1,2,3,5,8] ) )
19    ### but not 20 so duplicates are ignored
```

Reference

- Python for Informatics – C. Severance
- Python for u and me – Kushal Das
- Think Python – A. B. Downey - [O'Reilly]
- Python Crash Course – Eric Matthes [No starch Press]
- A Byte of Python – Swaroop C H
- Introducing Python – Bill Lubanovic [O'Reilly]

- Learning Python – 5th Ed – Mark Lutz - [O'Reilly]

End of Presentation

Python – S5