Unit 3
Dictionaries and Tuples

Dictionaries

(Like putting different things into a bag!)

- A "bag" of values, each with its own label (key).
- The 'most powerful' data collection types

Different names in different languages:
Associative arrays - Perl/PHP
Properties or Map or Hashmap - Java
Property Bag - C#/.NET

A dictionary is similar to a list, but:

- the order of items doesn't matter
- they aren't selected by an offset such as 0 or 1

Instead, a unique 'key' is associated with each 'value'
'key' can be any immutable data type: Boolean, float, tuple, string...but it is often a string!

Dictionaries are "Mutable" (contents can be changed)

Creating a dictionary:

1. Using {}
eg (i):

```
>>>empty_dict = {}
>>>empty_dict
{}
```

eg (ii):

```
>>>new_dict = {
. . .    "day" : 5,
. . .    "venue": "GJB",
. . .    "event": "Python Carnival!",
. . .    }

>>>new_dict
{ "day" : 5, "venue": "GJB", "event": "Python Carnival!" }
```

2. Using dict()
eg:

```
>>>new_dict = dict()
```

eg:

```
>>>purse = dict()
#lets create a dictionary named 'purse'
>>>purse
{ }
#and put some items into it.
. . .
```

We can 'add items to', 'remove items from' or 'change items in', a dictionary.

Comparing Lists and Dictionaries

eg:

```
>>>lst = list()                    >>>dd = dict()

>>>lst.append(21)                  >>>dd['age'] = 20
>>>lst.append(180)                 >>>dd['course code'] = 452

>>>print lst                       >>>print dd
  [21, 180]                        {'age':20, 'course code': 452}


>>>lst[0] = 24                     >>>dd['age'] = 23


>>>print lst                       >>>print dd
[24, 180]                          {'course code':452, 'age':23}
#ordered                           #unordered
```

List		Dictionary	
key	*value*	*key*	*value*
[0]	*24*	*['age']*	*23*
[1]	*180*	*['course code']*	*452*

Main Properties of Dictionaries in Python:

1. Accessed by key, not offset position
2. Unordered collections of arbitrary objects
3. Variable-length, heterogeneous, and arbitrarily nestable
4. Of the category "mutable mapping"
5. Tables of Object references (hash tables)

Dictionary Literals and Operations:

(a) Nested Dict

```
>>>D ={'info': {'name': 'Alice', 'age':18}}
```

To get the 'age' value,

```
>>> ?
```

(b) Alternative construction techniques:

(i) dict_var = dict(key1 = value1, key2 = value2, ...)

eg:

```
>>>D = dict(name='Alice', age=18)
```

(ii) dict_var = dict([(key1, value1),(key2, value2), ...])

eg:

```
>>>D = dict([('name','Alice'),('age',18)])
```

(iii) Using existing lists to create a dict (with zip() function)

dict_var = dict(zip(list1,list2))

eg:

>>>D = dict(zip(keyslist,valueslist))

(iv) Creating dict from keys only

dict_var = dict.fromkeys([key1, key2, ...])

eg:

>>>D = dict.fromkeys(['name','age','place'])

Note: All the values will be initialized to 'None' (not zero or blank)

>>>D
{'name': None, 'age': None, 'place': None}

(c) Indexing by key

Syntax:

dict_var['key']

eg:

>>>D['age']
18

(d) Membership operation

Syntax:

'key' in dict_var

eg:

>>>'place' in D

(e) Methods

(i) Print all keys in Dict

dict_var.keys()

(ii) Print all values in Dict

dict_var.values()

(iii) Print all key + value pairs in Dict

dict_var.items()

(iv) Make a copy

dict_var.copy()

(v) Remove all items from Dict

dict_var.clear()

(vi) Merging keys from different dict

dict_var1.update(dict_var2)

if 'key' of dict_var2 is present in dict_var1,
then 'value' is updated in dict_var1.

if 'key' of dict_var2 is not present in dict_var1,
then item is added to dict_var1.

(vii) Fetch by key, if absent default

dict_var.get(key, default?)

eg:

```
>>>D = {'one':1,'two':2, 'three':3}
>>>D.get('two','Not found!')
2
>>>D.get('five','Not found!')
'Not found!'
>>>
```

(ix) Remove by key, if absent default

dict_var.pop(key, default?)

(x) Fetch by key, if absent set default

dict_var.setdefault(key, default?)

(xi) deleting items by key

del dict_var[key]

(xii) Dictionary views

ver 2.x: dict_var.viewkeys(), dict_var.viewvalues()
ver 3.x: dict_var.keys(), dict_var.values()

(xiii) Dictionary comprehensions

- for creating dictionary using comprehension

dict_var = {var : expn for var in ...}

eg(1):

```
>>>D = {x : x**2 for x in range(5)}
>>>D
?
```

eg(2):

```
>>>D1 = {x:x+10 for x in range(2, 12, 2)}
>>>D1
?
```

Applications of Dictionary:

1. **Dictionary as a Set of Counters:**

To count the occurrence of a letter in a given string, the best data structure to use is 'dict'.

Usage of 'in' operator is handy (Membership operator)

eg:

```
# function to check if a given letter exists in the dictionary.
# If it exists, increment its value by 1,
# else insert the letter with initial value as 1.

def hist(s):     #'s' is a string
   d = dict()
   for c in s:
      if c not in d:
         d[c] = 1
      else:
         d[c] += 1
```

```
    return d
```

Note: For printing the items in dictionary, use iterations (for loop).

```
d1 = hist(input('Enter a string:'))
for key in d1:
    print key, d1[key]
   #prints a key and its corresponding value
```

Exercise:
Write a function that takes a string as argument and returns a dictionary containing words of the string (as keys) with their frequency of occurrence (as values).

2. Reverse Lookup

Typically when we need to print the all values in a dictionary, we use the keys as index (lookup).

```
>>>for key in dict1:
. . .      print dict1[key]
 #prints the value corresponding to key
```

But, if we have the value and want to find its key, there are 2 issues:
(1) There might be more than one key that maps to the same value.
(2) There is no simple syntax for accessing, the key needs to be searched.

eg:
Consider the hist() function, it returns a dictionary containing the characters as key and their occurrence count as value. To check how many keys have the same count, we need to use a loop..

```
# if 'd' is the dictionary, 'k' is key and 'v' is value, then
            for k in d:
                if d[k] == v:
                    print k
```

execute revlukup_v1.py

Dictionaries and Lists

A dictionary can have a list as values. Suppose a dictionary maps from letters to frequencies (letters are 'keys' and frequencies are 'values'), we may need to convert it.. that is.. create a list that maps from frequencies to letters

execute rev_lukup.py

Exercise

I(a). Write a function word_hist that counts the occurance of the following words in a given input file 'words.txt':

{'the', 'of', 'for', 'is', 'as', 'an'}

Display the words and their count.
[Hint: Extend the hist() function.]

I(b). Modify the function in I(a) so that the words are sorted.

I(c). Modify the function in I(a) so that the words appear in descending order of their frequency.

Use of random module

execute weather.py

Some useful functions in random:

- choice(seq) - choose a random element from a
non-empty sequence, 'seq'.
eg: random.choice([1, 2, 3, 5])

- randint(a, b) - return a random integer in the range
[a, b] (both inclusive)
eg: random.randint(1,100)

- sample(population, k) - chooses 'k' unique random elements from a 'population' sequence and returns a new list leaving the original 'population' unchanged.
eg: random.sample(range(10000000), 60)

Memos (Memorize abbreviated!!)

Consider the fibonacci series, we can generate a fibonacci series using dictionaries where the key indicates the element number and the value indicates the number in the series.

The first 2 values (0 and 1) in the series can be initialized in a dictionary while the remaining values are calculated and added to this dictionary.

Since the values 0 and 1 are required every time, they need not be recomputed..

Memo - is a computed value stored to avoid unnecessary re-computation in future.

execute memo_demo.py

Exercise
I. Run the above version of fibonacci series and the original version. Compare their run times.

Global Variables

Any variable declared outside a function belongs to '__main__' module(top level script environment). In the previous example(memo_demo.py), 'known' was a global variable.

eg(1):
```
myGlobal = 15

def func1():
   myGlobal = 45

def func2():
```

```
    print myGlobal

func1()
func2()
```

#Output: ?

eg(2):

```
myGlobal = 15

def func1():
    global myGlobal
    myGlobal = 45

def func2():
    print myGlobal

func1()
func2()
```

#Output:?

Objects and Values

Consider these assignment statements:

```
>>>a = 'purple'
>>>b = 'purple'
```

Though both refer to a string with same value, do they refer to the 'same' string?

There can be 2 possibilities here:

(i) a -> 'purple'
    b -> 'purple'

(ii) a, b -> 'purple'

Case (i): 'a' and 'b' refer to different objects that have the same value.
Case (ii): 'a' and 'b' refer to the same object.

How do we check the possibilities?

Use the 'is' operator

```
>>>a is b
True/False
```

False => Case (i)
True => Case (ii)

Does case(ii) work for all data types and data structures?

Though two lists are identical (same elements), they are not the same object!
eg:
>>>alist = [1, 2, 3, 4]
>>>blist = [1, 2, 3, 4]
>>>alist is blist
False

Note: Case (ii) ['a' and 'b' refer to the same object] is possible for all data types if we assign value of another variable instead of initializing.

eg:
>>>alist = [1, 2, 3, 4]
>>>clist = alist
>>>alist is clist
True

This concept is termed as 'Aliasing'. Any changes made thru one variable (reference) affects the other:

>>>alist.append(5)
>>>clist
[1, 2, 3, 4, 5]

Note: It is always safer to avoid 'aliasing' when working with 'mutable' objects.

Tuples

They are another kind of sequence that function much like a list - they have elements which are indexed starting at 0.
They work exactly like lists, except that tuples can't be changed in place!!

Basic Properties:

1. Ordered collections of arbitrary objects
2. Accessed by offset
3. Of the category "immutable sequence"
4. Fixed - length, heterogeneous and arbitrarily nested
5. Arrays of object references

Creating tuples

(a) Using tuple() function

eg(1):
>>>x = tuple()
>>>type(x)
<class 'tuple'>
>>>x
( )

eg(2):
>>>y = tuple('Hello')
>>>y
?

(b) Using ()

eg(3):

>>>t=()
>>>type(t)
<class 'tuple'>
>>>t
()

eg(4):
>>>x = ('Tom', 'Dick', 'Harry')
>>>type(x)
<class 'tuple'>
>>>x
('Tom', 'Dick', 'Harry')

(c) Casual way!
>>>z = 1,2,3,4
>>>type(z)
<class 'tuple'>
>>>z
(1, 2, 3, 4)

Tuple literals and Operations

(a) Tuple syntax Peculiarities: Commas and parentheses

eg(1):
>>>x = (40)
>>>type(x)
?
>>>x
?

eg(2):
>>>x=(40,)
>>>type(x)
?
>>>x
?

This is confusing!?

(b) Nested tuples
>>>T = ('Bob', ('Developer','Manager'))
How to print the message below using tuple, T?
Bob is a Developer

>>>T[0]
Bob
>>>print(T[0], 'is a ',T[1][0])
Bob is a Developer

(c) Indexing and Slicing
>>>T1 = (1, 2, 3, 4, 5)
>>>T1[0:2]
(1, 2)
>>>T1[3:]
(4, 5)

>>>T1[::-1]
(5, 4, 3, 2, 1)
>>>T1[0], T[2:4]
(1, (3, 4))

(d) Concatenate & Repetition
>>>T1 = (1, 2)
>>>T2 = (3, 4)
>>>T1 + T2
(1, 2, 3, 4)
*>>>T1 * 2*
(1, 2, 1, 2)

(e) Iteration and Membership

eg(1):
>>>T1 = (1, 2, 3, 4, 5)
>>>for ele in T1:
. . . print(ele)
. . .
1
2
3
4
5
>>>

eg(2):
>>>2 in T1
True
>>>10 in T1
False

(f) Conversions, methods and immutability

(1) Conversions & immutability
eg(1): To convert a Tuple to a List
>>>T = (1, 2, 3, 4)
>>>alist = list(T)
[1, 2, 3, 4]
 # elements of 'alist' can be modified.

eg(2): To convert a list to a Tuple
>>>blist = [4, 3, 2, 1]
>>>T1 = tuple(blist)
>>>T1
(4, 3, 2, 1)
 # T1 is immutable!

(2) Methods: count() and index()

eg:
>>>Tnew = (1, 2, 3, 5, 2, 7, 8,2)
>>>Tnew.count(2)
3

>>>Tnew.index(2) # offset of first appearance
1

>>>Tnew.index(2, 2) # offset of appearance after offset 2
4

>>>Tnew.index(2, 5) # offset of appearance after offset 5
7

Exercise:
Consider a tuple containing duplicate elements. Count the number of times each element appears in the tuple. If the appearance is more than once, then print all the indices along with the element.

Tuple Unpacking!

Consider an example tuple:
>>>T = (12, 23, 34, 45)
>>>T
(12, 23, 34, 45)

>>>a, b, c, d = T

The multiple variables in the LHS should be equal to the number of elements in the tuple

>>>a
12
>>>d
45

Tuple assignment

Whenever we need to swap two variables, we use the conventional method: Using a temporary variable,

>>>temp = a
>>>a = b
>>>b = temp

It is rather simple to perform swapping using tuple assignment (does not require 'temp' variable!)

eg(1):
>>>a=5
>>>b=10
>>>a, b = b, a
>>>a
10
>>>b
5

eg(2):
>>>A = (1, 2, 3)
>>>B = (4, 5, 6)
>>>A, B = B, A
>>>A
(4, 5, 6)
>>>B

(1, 2, 3)

Note:
This type of assignment (multiple variables in LHS) can be used with any kind of sequence (string, list or tuple) on the RHS.

eg: Splitting the user name and the domain name from an email id, will be as simple as,

```
>>>addr = 'hod.cse@nie.ac.in'
>>>uname, domain = addr.split('@')
>>>uname
hod.cse
>>>domain
nie.ac.in
```

Tuples as Return values

Usually a function returns only one value. But, if the value is a tuple, it implies that the function is returning multiple values.

(a) One such function is the 'divmod()' built-in function.
Syntax:

```
T = divmod(a, b)
```

where, 'a' is the dividend, 'b' is the divisor, and 'T' is the returned tuple containing the quotient and the remainder.

```
>>>q, r = divmod(12,4)
>>>q
3
>>>r
0
```

(b) User defined functions can be written in such a way that multiple values are returned from it.
eg:

```
>>>def min_max(t):
. . .     return min(t), max(t)
. . .
>>>T = (3, 6, 1, 8, 7, 2)
>>>min_max(T)
(1, 8)
>>>
```

Variable-length Argument Tuples

Concept of gather and scatter (thru tuples)

Conventionally, any function takes precise number of arguments. But, we can define functions which take variable-length arguments.

A parameter name that begins with a '*' gathers arguments into a tuple.

eg:

```
>>>def prints(*args):
. . .     print(args)
. . .
```

```
>>>prints('5th', 25, 45.75,'S')
('5th', 25, 45.75, 'S')
```

'scatter' is the complement of 'gather'.
Consider a sequence of values to be passed to a function as multiple arguments, the '*' operator can be used.

eg: the 'divmod()' built-in function requires 2 arguments

```
>>>T = (10, 6)
>>> divmod(*T)
(1, 4)
>>>
```

Lists and Tuples

zip() is a built-in function that takes two or more sequences and "zips" them into a list of tuples where each tuple contains one element from each sequence.

eg(1):

```
>>>a = 1, 2, 3, 4
>>>b = 'a', 'b', 'c', 'd'
>>>zip(a, b)
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

#in ver. 3.x
>>>tuple(zip(a,b))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Using for loop for traversal:

```
>>>nu = zip(a,b)
>>>for digit, letter in nu:
. . .    print(digit, letter)
. . .
>>>
```

Q: Can we combine more than 2 tuples using zip()?
eg (2):

```
>>>a = 1, 2, 3
>>>b = 'hai'
>>>c = 'Wow'
>>>res = zip(a, b, c) # can this be done??
```

Additional usage of zip():
It can be used to traverse thru two or more sequences at the same time.

eg: To compare two sequences (element by element), and return True if all the elements are matching..

```
def has_match(t1, t2):  #t1 and t2 are two sequences
   for x , y in zip(t1, t2):
      if x != y: return False
   return True
```

Note:

To traverse the elements of a sequence and their indices, a built-in function 'enumerate()' can be used:

eg:
```
>>>s = 'hello'
>>>for index, element in enumerate(s):
. . .       print(index, element)
. . .
0 h
1 e
2 l
3 l
4 o
>>>
```

Dictionaries and Tuples

(a) Converting a dictionary to a list of tuples:

Recall the method called 'items()' in dictionary..
```
>>>d = {'a':1, 'b':2, 'c':3}
>>>t = d.items()
>>>t
[('a', 1), ('b', 2), ('c', 3)]
```

it returns a list of tuples, where each tuple is a key-value pair.

Note: In Python 3.x, items() returns an iterator.

(b) Converting a list of tuples to a dictionary:

eg(1):
```
>>>t = [('a', 1), ('b', 2), ('c', 3)]
>>> d = dict(t)
>>>d
{'a': 1, 'b': 2, 'c': 3}
```

Using zip()
eg(2):
```
>>>d = dict(zip('abc', range(1,4)))
>>>d
{ 'a' : 1, 'b' : 2, 'c' : 3 }
```

Comparing Tuples

Use relational operators to compare the tuples/any sequence.

eg:
```
>>>(0, 1, 2) < (5, 1, 2)
?

>>>(0, 1, 20000) < (0, 3, 4)
?
>>>('Alice', 'Tom') < ('Alice', 'Harry')
?
```

>>>('Alice', 'Tom') < ('Bob', 'Dick')
?

Note: The comparison starts with first elements of both the sequence. If the elements are equal, the next elements are compared until they differ. The remaining elements in the sequence are not considered.

Exercises