Python list ,tuples

Accessing list elements and traversal

- ★ List indices work the same way as string indices.
- ★ Thus you can use the subscript operator to access the list elements.

```
>>> newnest=[10, 20, [25, 30], 40]
>>> print(newnest[0])
10
>>> print(newnest[2])
[25, 30]
>>> print(newnest[-1])
40
>>> print(newnest[4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(newnest[5-4])
20
>>> print(newnest[1.0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
>>> print(newnest[-5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

★ You could also **use an expression as an index in the subscript operator**, as **newnest[5-4]** in the example above.

★ The interpreter will evaluate the expression to determine the index.

★ **Printing the list**

- by specifying its name in the print method displays the list elements enclosed in a **pair of brackets**.

★ To **display the entire list without the brackets** themselves, you should traverse the list printing the elements one by one.

```
>>> prime=[2,3,5,7,11]
>>> for p in prime:
...     print(p,end=" ")
...
2 3 5 7 11
```

list comprehension

- ★ Creating a new list from an existing list is called list comprehension.
- ★ Assume you have a list of numbers from which you want to create another list consisting only of odd numbers in the original list.
- ★ This can be done by using a for loop to traverse the original list and copying the odd numbers to a second list.
- ★ But with list comprehension, life is far more easy! See below:
- ★

```
>>> numbers = [1, 28, 73, 4, 100, 358, 75, 208]
>>> odd = [num for num in numbers if num%2 != 0]
>>> print(odd)
[1, 73, 75]
```

The statement

```
odd = [num for num in numbers if num%2 != 0]
```

essentially directs the interpreter to iterate through the list **numbers** and copy all odd numbers to another list **odd**.

Suppose, now you need to create a list of two-digit even numbers, you could use comprehension:

```
>>> even2dig = [num for num in range(10,100) if num%2 == 0]
>>> print(even2dig)
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
    40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
    70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
```

Here, `range(10,100)` generates two digit numbers and the condition `if num%2 == 0` makes sure that only the even numbers are included in the constructed list.

Finally, assume you want to replace all multiples of 5 (less than 30) with -1, you just resort to comprehension:

```
>>> nomul5 = [num if num%5 != 0 else -1 for num in range(20)]
>>> print(nomul5)
[-1, 1, 2, 3, 4, -1, 6, 7, 8, 9, -1, 11, 12, 13, 14, -1, 16,
    17, 18, 19]
```

The interpreter generates two-digit numbers less than 20 and then puts the generated number into the constructed list if it is not a multiple of 5, otherwise puts a -1 into the list.

List operations

The + operator concatenates lists:

```
>>> prime=[2,3,5,7]
>>> composite=[4,6,8,10]
>>> numbers=prime+composite
>>> print(numbers)
```

```
[2, 3, 5, 7, 4, 6, 8, 10]
```

The * operator repeats a list a given number of times:

```
>>> binary=[0,1]
>>> bytesequence=binary*4
>>> print(bytesequence)
[0, 1, 0, 1, 0, 1, 0, 1]
```

Equality operator works well on lists. It checks if two lists have the same elements. See an example:

```
>>> even=[2,4,6,8]
>>> mul2=[2,4,6,8]
>>> print(even==mul2)
True
>>> composite=[4,6,8]
>>> print(even==composite)
False
```

Other relational operators also work with lists. Consider:

```
>>> prime=[2,3,5]
>>> even=[2,4,6]
>>> print(prime>even)
False
```

- ➔ Python starts by comparing the first element from each list.
- ➔ If they are equal,it goes on to the next element, and so on, until it finds the first pair of elements that are different and determines the relation between them.
- ➔ In the above example, prime[0] = = even[0]. Next, prime[1] and even[1] are compared.
- ➔ Thus the resulting relation is '<' and prime > even is thus False.

➔ Note that once the result is determined, the subsequent elements are skipped.

```
>>> prime=[2,3,7]
>>> even=[2,4,6]
>>> print(prime>even)
False
```

Membership operators can be applied to a list as well. See below:

```
>>> even=[2,4,6,8]
>>> composite=[4,6,8]
>>> print(2 in even)
True
>>> print(2 in composite)
False
```

```
>>> print(3 not in composite)
True
```

List slices

The slice operator on a list follows the same rules as applied to strings. See some illustrations:

```
>>>sequence=["strings","lists","tuples","bytearrays",
"bytesequences","range objects"]
>>> print(sequence[:])
['strings', 'lists', 'tuples', 'bytearrays', 'bytesequences',
    'range objects']
>>> print(sequence[1:4])
['lists', 'tuples', 'bytearrays']
>>> print(sequence[:3])
['strings', 'lists', 'tuples']
>>> print(sequence[3:])
['bytearrays', 'bytesequences', 'range objects']
```

List mutations

Unlike strings, lists are **mutable**. In other words, a list is updatable – elements can be inserted, removed, or replaced. You use the subscript operator to replace an element at a given position:

```
>>> even=[2,4,5,8]
>>> even[2]=6
>>> print(even)
[2, 4, 6, 8]
```

You can also replace a single list item with a new list. See below:

```
>>> even=[2,4,6,8]
>>> even[3]=[8,10]
>>> print(even)
[2, 4, 6, [8, 10]]
```

Slice operator and mutations

- The slice operator is a nice tool to mutate lists in terms of replacing, inserting,or removing list elements.
- Replacing elements
  - You can use the slice operator to replace a single element or multiple elements in a list.
  -

```
>>> composite=[13,17,19,23,25,27,37,41]
>>> composite[4:6]=[29,31]
>>> print(composite)
[13, 17, 19, 23, 29, 31, 37, 41]
>>> odd=[1,3,5,8]
>>> odd[3:4]=[7]
>>> print(odd)
[1, 3, 5, 7]
```

Inserting elements

The slice operator is useful for inserting new elements into a list at the desired location:

```
>>> prime=[2,3,5,7,13,17,19]
>>> prime[4:4]=[11]
>>> print(prime)
[2, 3, 5, 7, 11, 13, 17, 19]
>>> composite=[2,10,12]
>>> composite[1:1]=[4,6,8]
>>> print(composite)
[2, 4, 6, 8, 10, 12]
```

Removing elements

The slice operator can also be used to remove elements from a list by assigning the empty list to them. Examples follow:

```
>>> composite=[3,5,7,9,11]
>>> composite[3:4]=[]
>>> print(composite)
[3, 5, 7, 11]
>>> composite=[29,31,33,35,37]
>>> composite[2:4]=[]
>>> print(composite)
[29, 31, 37]
```

Using del keyword for deletion

- ★ Assigning a list element to an empty list for deletion is cumbersome.
- ★ As an alternative,Python provides the del keyword exclusively for deletion.

```
>>> composite=[3,5,7,9,11]
>>> del composite[3]
>>> print(composite)
[3, 5, 7, 11]
>>> composite=[29,31,33,35,37]
>>> del composite[2:4]
>>> print(composite)
[29, 31, 37]
>>> del composite[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

As illustrated above, `del` causes a runtime error if the index is out of range.

List methods :The list type includes several methods for inserting and removing elements. s. L denotes a list

Table 10.1: Some useful list methods. L denotes a list

List method	Purpose
`L.index(element)`	Returns the position of **element** in the list **L**. It produces an error if there is no such element.
`L.insert(position, element)`	Inserts **element** at **position** if **position** is less than the length of **L** . Otherwise, inserts **element** at the end of **L** .
`L.append(element)`	Inserts **element** at the end of **L**.
`L.extend(aList)`	Inserts the elements of the list **aList** to the end of **L**.
`L.remove(element)`	Removes **element** from the list **L**. It throws an error if there is no such element.
`L.pop()`	Removes and returns the element at the end of **L** .
`L.pop(position)`	Removes and returns the element at **position** in the list **L**.
`L.count(element)`	Returns the number of times **element** appears in the list **L**.
`L.sort(element)`	Sorts (arrange in ascending order) the elements of the list **L**.
`L.reverse()`	Reverses the elements of the list **L**.

```
>>> a = [66.25, 333, 333, 7, 1, 1234.5]
>>> print("Index of 1 is",a.index(1))
Index of 1 is 4
>>> print("Index of 333 is",a.index(333))
Index of 333 is 1
>>> a.insert(2,4587)
>>> a.insert(8,-55)
>>> print("After inserting 4587 and -55, the list is",a)
After inserting 4587 and -55, the list is [66.25, 333, 4587,
    333, 7, 1, 1234.5, -55]
>>> a.append(200)
>>> print("The list on appending 200 is",a)
The list on appending 200 is [66.25, 333, 4587, 333, 7, 1,
    1234.5, -55, 200]
>>> a.sort()
>>> print("The list after sorting is",a)
The list after sorting is [-55, 1, 7, 66.25, 200, 333, 333,
    1234.5, 4587]
>>> b=['a',25.47,'c',7,18]
```

```
>>> print("The list after sorting is",a)
The list after sorting is [-55, 1, 7, 66.25, 200, 333, 333,
    1234.5, 4587]
>>> b=['a',25.47,'c',7,18]
>>> a.extend(b)
>>> print("The list a after extending with elements of b is",a)
The list a after extending with elements of b is [-55, 1, 7,
    66.25, 200, 333, 333, 1234.5, 4587, 'a', 25.47, 'c', 7, 18]
>>> a.remove(1)
>>> print("The list a after removing 1 is",a)
The list a after removing 1 is [-55, 7, 66.25, 200, 333, 333,
    1234.5, 4587, 'a', 25.47, 'c', 7, 18]
>>> a.remove(333)
```

```
>>> print("The list a after removing 333 is",a)
The list a after removing 333 is [-55, 7, 66.25, 200, 333,
    1234.5, 4587, 'a', 25.47, 'c', 7, 18]
>>> x=a.pop()
>>> print("The element popped from the list a is",x)
The element popped from the list a is 18
>>> a.pop(6)
4587
>>> print("The list a after popping element at position 6 is",a)
The list a after popping element at position 6 is [-55, 7,
    66.25, 200, 333, 1234.5, 'a', 25.47, 'c', 7]
>>> print("The list a has",a.count(7),"occurrences of 7")
The list a has 2 occurrences of 7
>>> a.reverse()
>>> print("The list a in reverse is ",a)
The list a in reverse is  [7, 'c', 25.47, 'a', 1234.5, 333,
    200, 66.25, 7, -55]
```

Lists and functions

★ A function can be made to return multiple values using lists.

★ This is done by defining the function to return a list of values instead of a single value as in the normal case. The following code illustrates this:

```
>>> def printTop3(list):
...     list.sort()
...     list.reverse()
...     top3=[list[i] for i in range(3)]
...     return(top3)
...
>>> mylist=[23,1,78,50,100,-5]
>>> printTop3(mylist)
[100, 78, 50]
```

Strings and lists

To construct a list out of the characters from a string, you can use `list()` method as follows:

```
>>> str="two words"
>>> strlist=list(str)
>>> print(strlist)
['t', 'w', 'o', ' ', 'w', 'o', 'r', 'd', 's']
```

If you want to split a multi-word string into its constituent words and construct a list out of those words, you should use the `split()` method:

```
>>> str="two words"
>>> strwords=str.split()
>>> print(strwords)
['two', 'words']
```

★ the white space where you split the string is known as the delimiter.

★ If you want to specify a different delimiter, you can pass that character (or even a string) as an argument to the split() method.

```
>>> str="two words"
>>> wsplit=str.split('w')
>>> print(wsplit)
['t', 'o ', 'ords']
>>> wosplit=str.split('wo')
>>> print(wosplit)
['t', ' ', 'rds']
```

```
>>> str="two words"
>>> wsplit=str.split('w')
>>> print(wsplit)
['t', 'o ', 'ords']
>>> wosplit=str.split('wo')
>>> print(wosplit)
['t', ' ', 'rds']
```

Notice that the delimiter doesn't appear in the list.

The `join()` method works in the opposite sense of `split()`. It concatenates a list of strings by inserting a "separator" between them. The following code uses a blank space as the separator.

```
>>> sep=" "
>>> wordlist=["three", "word", "string"]
>>> str=sep.join(wordlist)
>>> print(str)
three word string
```

The following code uses a hyphen as a separator.

```
>>> sep="-"
>>> wordlist=["three", "word", "string"]
>>> str=sep.join(wordlist)
>>> print(str)
three-word-string
```

List of lists

All the elements in a list may be again lists. Such a list is called **list of lists**. Here is one example:

```
lol=[[1], [15, 9], [108, 778]]
```

Since the constituent elements themselves are lists, to access the items in these constituents, you need to use the subscript operator twice as illustrated below:

```
>>> lol=[[1], [15, 9], [108, 778]]
>>> celt=lol[1]
>>> elt=celt[1]
>>> print(elt)
9
```

You can even combine the above two steps:

```
>>> lol=[[1], [15, 9], [108, 778]]
>>> elt=lol[1][1]
>>> print(elt)
9
```

A list of lists is often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

To print a row, you use the subscript operator:

```
>>> print(matrix[2])
[7, 8, 9]
```

To extract an individual element from the matrix, you should use the double-index form:

```
>>> print(matrix[2][1])
8
```

List aliasing and cloning

If you assign a list variable to another, both variables refer to the same list.

Because the same list has two different names now, we say that the list is aliased. Changes made with one list affect the other.

```
>>> list = [1, 2, 3]
>>> alias=list
>>> print(alias)
[1, 2, 3]
>>> alias[1]=4
>>> print(list)
[1, 4, 3]
```

➔ If you want to modify a list and also keep a copy of the original, you need to use a technique called cloning.

➔ The easiest way to clone a list is to use the slice operator.

```
>>> list = [1, 2, 3]
>>> clone=list[:]
>>> print(clone)
[1, 2, 3]
>>> clone[1]=4
>>> print(list)
[1, 2, 3]
```

```
>>> print(clone)
[1, 4, 3]
>>> list[1]=5
>>> print(clone)
[1, 4, 3]
>>> print(list)
```

Taking a slice of any list creates a new list. Thus you are free to make changes to the new list without modifying the original or vice versa.

Equality of lists

- A list and its alias refer to the same list.
- On the other hand, a list and its clone refer to two different lists although both have the same elements.
- The equality between a list and its alias is object identity and the equality between a list

and its clone is known as structural equivalence.

- The is operator can be used to test for object identity.
- It returns True if the two operands refer to the same list, and it returns False if the operands refer to distinct lists (even if they are structurally equivalent).
- T

Equality of lists

```
>>> mylist=[1,2,3]
>>> alias=mylist
>>> clone=mylist[:]
>>> print(alias is list)
True
>>> print(clone is list)
False
```

tuples

★ A tuple is a sequence of values that resembles a list, except that a tuple is immutable.

★ You create a tuple by enclosing its elements in parentheses instead of square brackets.

★ (Strictly speaking, the enclosing parentheses are optional, but are included to improve readability.) The elements are to be separated by commas. Following are some examples:

```
t = ('a', 'b', 'c', 'd', 'e')
s = ('g', 2, 7, 8.978)
```

★ To create a tuple with a single element, we have to include a comma at the end, even though there is only one value.

★ Without the comma, Python treats the element as a string in parentheses.

```
>>> t1=('a',)
>>> t2=('a')
>>> print(type(t1))
<class 'tuple'>
>>> print(type(t2))
<class 'str'>
```

Tuple creation

a tuple can be created by enclosing its elements in parentheses.

Another way to create a tuple is to use the tuple() function.

```
>>> t1=tuple("string")
>>> print(t1)
('s', 't', 'r', 'i', 'n', 'g')
>>> t2=tuple([1,2,3])
>>> print(t2)
(1, 2, 3)
>>> t3=tuple()
>>> print(t3)
()
>>> t4=tuple([4])
>>> print(t4)
(4,)
```

In the above example, `t3` is an empty tuple. Also, notice how `t4` is printed with commma at the end.

Tuple operations

Most of the operators and functions used with lists can be used similarly with tuples. A few examples follow:

```
>>> lan=tuple("python")
>>> print(lan[0])
p
>>> print(lan[1:4])
('y', 't', 'h')
```

But, unlike lists, tuples are immutable – you are bound to get error when you try to modify the tuple contents. See below:

```
>>> lan=tuple("python")
>>> lan[0]='P'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Python provides a quick way to swap two variables `a` and `b` without using any third temporary variable:

```
>>> a,b=5,10
>>> print(a,b)
5 10
>>> a,b=b,a
>>> print(a,b)
10 5
```

This is known as **tuple assignment**.

11.3 Sets

A set is a collection of items just as tuples or lists but unlike the other two, a set is unordered; the items in a set do not have a defined order.

11.3.1 Creating a set and accessing its elements

A set is created by enclosing the items in a pair of braces. Following are some examples:

```
>>> directions={'east','west','south','north'}
>>> print(directions)
{'east', 'south', 'north', 'west'}
```

Notice that since the items are unordered in a set, each time you print the set, the output may be different concerning the order of the elements.
You can also use the **set** method to create a set from a list or a tuple as follows:

```
>>> numbers=set([1,2,3,4])
>>> print(numbers)
{1, 2, 3, 4}
>>> vowels=set(('a','e','i','o,','u'))
>>> print(vowels)
{'o', 'u', 'a', 'e', 'i'}
```

The items in a set aren't associated with index or position. This means it is impossible to print an arbitrary element of the set.

```
>>> vowels=set(('a','e','i','o,','u'))
>>> vowels[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

To traverse the set, use a `for` loop: The code

```
>>> for char in vowels:
...     print(char,end=" ")
...
o u a e i
```

Sets do not allow duplicate elements. The duplicates are automatically removed even if you supply non-unique values. See below:

```
>>> fibonacci={0,1,1,2,3,5}
>>> print(fibonacci)
{0, 1, 2, 3, 5}
```

11.3.2 Adding and removing set elements

To add a single element to a set, you use the `add()` method:

```
>>> even={2,4,8,10}
>>> even.add(6)
>>> print(even)
{2, 4, 6, 8, 10}
```

To add multiple items to a set, the `update()` method is to be used:

```
>>> odd={1,3,7,9,13,15}
>>> odd.update({5,11})
>>> print(odd)
{1, 3, 5, 7, 9, 11, 13, 15}
```

To remove an item from a set, you should use the `remove()` method:

```
>>> primes={3,5,7,9,11}
>>> primes.remove(9)
>>> print(primes)
{3, 5, 7,11}
```

If the item to be removed is not present in the set, `remove()` will raise an error:

```
>>> primes={3,5,7,11}
>>> primes.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

A second way to remove an item from a set is to use the `discard()` method which works the same way as `remove`. The difference between these two methods is that `discard()` will not raise an error if the item to be removed doesn't exist in the set unlike `remove()`. See an example:

```
>>> primes={3,5,7,9,11}
>>> primes.discard(9)
>>> print(primes)
{3, 5, 7, 11}
>>> primes.discard(2)
>>> print(primes)
{3, 5, 7, 11}
```

To remove all the set items at once, you can use the `clear()` method as illustrated below:

```
>>> primes={3,5,7,9,11}
>>> primes.clear()
```

The keyword `del` can be used to delete the set itself, that is, after the `del` operation, the set will not exist anymore. Any subsequent access to the set will throw an error. See below:

```
>>> primes={3,5,7,9,11}
>>> del primes
>>> print(primes)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primes' is not defined
```

11.3.3 Sets and relational operators

The relational operators can be used to check the relationship (subset, superset, etc) between two sets. The meaning of the relational operators in the context of sets is summarized in Table 11.1

Table 11.1: Relational operators with sets

Operator	Interpretation
<	$\subset$
>	$\supset$
<=	$\subseteq$
>=	$\supseteq$
==	checks if two sets are same
!=	checks if two sets are not same

```
>>> digits={0,1,2,3,4,5,6,7,8,9}
>>> natural={1,2,3,4,5,6,7,8,9,10}
>>> even={0,2,4,6,8}
>>> odd={1,3,5,7,9}
>>> prime={3,5,7}
>>> composite={4,6,8}
>>> print(even<natural)
False
>>> print(odd<digits)
True
>>> print(prime<=odd)
True
>>> print(digits>=even)
True
>>> print(composite==even)
False
>>> print(prime!=odd)
True
```

11.3.4 Mathematical set operations on Python sets

Python supports all the mathematical set operations (union, intersection, etc.). These are summarized in Table 11.2. See some illustrative examples below:

```
>>> positive={1,2,3,4,5}
>>> negative={-5,-4,-3,-2,-1}
>>> numbers=positive.union(negative)
>>> print(numbers)
{1, 2, 3, 4, 5, -2, -5, -4, -3, -1}
>>> even={2,4,6,8,10}
>>> mul3={3,6,9,12}
>>> even={2,4,6,8,10,12}
>>> mul6=even.intersection(mul3)
>>> print(mul6)
{12, 6}
>>> mul15={0,15,30,45}
>>> mul5={0,5,10,15,20,25,30,25,40,45}
>>> mul5.intersection_update(mul15)
>>> print(mul5)
{0, 45, 30, 15}
>>> mul4={4,8,12,16,-4,-12,-20}
>>> mul8={8,16,-40,-120}
>>> mul4only=mul4.difference(mul8)
>>> print(mul4only)
{4, 12, -20, -12, -4}
```

```
>>> mul4.difference_update(mul8)
>>> print(mul4)
{-4, 4, -12, -20, 12}
>>> numbers={-3,-2,-1,0,1,2,3}
>>> natural={1,2,3,4,5}
>>> integers=numbers.symmetric_difference(natural)
>>> print(integers)
{0, 4, 5, -1, -3, -2}
>>> numbers.symmetric_difference(natural)
>>> print(numbers)
{0, 4, 5, -1, -3, -2}
```

Set method	Meaning
`A.union(B)`	returns a new set `S` = `A` ∪ `B`
`A.intersection(B)`	returns a new set `S` = `A` ∩ `B`
`A.intersection_update(B)`	changes the set `A` to `A` ∩ `B`
`A.difference(B)`	returns a new set `S` = `A` − `B`
`A.difference_update(B)`	changes the set `A` to `A` − `B`
`A.symmetric_difference(B)`	returns a new set `S` = (`A` − `B`) ∪ (`B` − `A`)
`A.symmetric_difference_update(B)`	changes the set `A` to (`A` − `B`) ∪ (`B` − `A`)

DICTIONARY

A dictionary associates a set of **keys** with data **values**. For example, the words in a standard English Dictionary comprise a set of keys, whereas their associated meanings are the data values. Thus a dictionary is a mapping between a set of 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**, sometimes called an **item** or an **entry**.

A Python dictionary is written as a sequence of key/value pairs separated by commas and the entire sequence is enclosed in a pair of braces. Each key is separated from its value by a colon (:). Such a list of key-value pairs enclosed in a pair of braces is known as a **dictionary literal**. Following is an example dictionary:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(stud)
{'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
```

Here `Name`, `Age` and `Class` are keys whereas `Ram`, `18` and `S1` are the corresponding values. An empty dictionary without any items is written as `{}`.

Keys are unique within a dictionary although values need not be. Although the entries may appear to be ordered in a dictionary, this ordering is not the same, each time we print the entries. In general, the order of items in a dictionary is unpredictable.

11.4.1 Dictionary operations

We now move on to the discussion of some common operations on dictionaries.

11.4.1.1 Accessing values

The subscript operator is used to obtain the value associated with a key. See below:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print("I am",stud['Name'],"aged",stud['Age'],"and I am
    studying in",stud['Class'])
I am Ram aged 18 and I am studying in S1
```

However, if the key is not present in the dictionary, Python raises an error:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(stud['College'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'College'
```

11.4.1.2 Traversing a dictionary

A simple `for` loop can be used to traverse a dictionary as follows:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> for key in stud:
...     print(key,"-",stud[key])
...
Name - Ram
Age - 18
Class - S1
```

The key-value pairs need not be always printed in the order you gave.

11.4.1.3 Inserting keys and updating key values

You should use the subscript operator to add a new key/value pair to the dictionary. The following code illustrates this:

```
>>> stud['College']="Engineering college"
>>> print(stud)
{'Name': 'Ram', 'Age': 18, 'Class': 'S1', 'College':
    'Engineering college'}
```

The subscript is also used to replace the value of an existing key:

```
stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> stud['Age']=19
>>> print(stud)
{'Name': 'Ram', 'Age': 19, 'Class': 'S1'}
```

11.4.1.4 Removing keys

To remove a key from a dictionary, use the `del` operator. The corresponding key-value pair will be removed. This is illustrated below:

```
stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> del stud['Age']
>>> print(stud)
{'Name': 'Ram', 'Class': 'S1'}
```

11.4.1.5 Miscellaneous operations

The `len()` function returns the number of key-value pairs in a dictionary:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(len(stud))
3
```

The `in` operator can be used to know if a key exists in the dictionary. See below:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print('Age' in stud)
True
>>> print('College' in stud)
False
```

11.4.2 Dictionary methods

Python provides several methods to manipulate the dictionary elements. Table 11.3 summarizes the commonly used dictionary methods, where `dict` refers to a dictionary.

Table 11.3: Some dictionary methods, with `dict` denoting the dictionary

Dictionary method	Purpose
`dict.get(key,default)`	Returns the value if the key exists or returns the `default` if the key does not exist. Displays "`None`" if the default is omitted and the key does not exist.
`dict.pop(key,default)`	Removes the key and returns the value if the key exists or returns the text `default` if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`dict.keys()`	Returns a list of the keys.
`dict.values()`	Returns a list of values.
`dict.items()`	Returns a list of tuples containing the key and value pairs.
`dict.clear()`	Removes all the items from the dictionary.

Some illustrations follow now:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(stud.get('Name'))
Ram
>>> print(stud.get('College'))
None
>>> print(stud.get('College',"Oops!!No such key!"))
Oops!!No such key!
>>> stud.pop('College')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'College'
>>> stud.pop('Age')
18
>>> print(stud)
{'Name': 'Ram', 'Class': 'S1'}
>>> print(stud.keys())
dict_keys(['Name', 'Class'])
>>> print(stud.values())
dict_values(['Ram', 'S1'])
>>> print(stud.items())
dict_items([('Name', 'Ram'), ('Class', 'S1')])
>>> stud.clear()
>>> print(stud)
{}
```

11.4.3 Dictionary aliasing and copying

Assigning a dictionary literal to another creates an alias of the original dictionary. In this case, the changes made to one will affect the other. If you want to modify a dictionary and keep a copy of the original, you need to use the `copy()` method. With a copy, you can make changes to it without affecting the original. The following illustrates this:

```
>>> directions={'N':'North','E':'East','S':'South','W':'West'}
>>> alias=directions
>>> copy=directions.copy()
>>> copy['S']='Sit down students!'
>>> print(copy)
{'N': 'North', 'E': 'East', 'S': 'Sit down students!', 'W':
    'West'}
>>> print(directions)
{'N': 'North', 'E': 'East', 'S': 'South', 'W': 'West'}
>>> alias['S']='Sit down students!'
>>> print(directions)
{'N': 'North', 'E': 'East', 'S': 'Sit down students!', 'W':
    'West'}
>>> print(alias)
{'N': 'North', 'E': 'East', 'S': 'Sit down students!', 'W':
    'West'}
```

Program 11.56. To create a histogram for a string.

Solution:

Histogram for a string shows how many times each letter appears in the string. A dictionary is very apt for representing a histogram. The keys represent the letters in the string and values represent the corresponding counts.

```
histogram = {}
str=input("Enter a string")
for letter in str:
    histogram[letter] = histogram.get(letter, 0) + 1
print(histogram)
```

Program 11.57. To create a dictionary out of the keys and values input from the user.

```
dict={}
keys=[]
values=[]
n=int(input("How many entries you want?"))
print("Input the keys")
for i in range(n):
   keys.append(int(input()))
print("Input the corresponding values")
for i in range(n):
   values.append(int(input()))
dict={keys[i]:values[i] for i in range(n)}
print(dict)
```

Program 11.58. To initialize a dictionary and then print its contents in the ascending order of keys.

```
dict={'Name':'XYZ','Class':'S1','Age':18,'College':'Engineering
    college'}
keys=list(dict.keys())
keys.sort()
for key in keys:
    print(key,"-",dict[key])
```