Python Data Structures

LIST

Lists are mutable sequences i.e.; you can change elements of a list in place.
Example

```
>>> l = [1, 2, 3]
>>> l[0]
1
>>> l.append(1)
>>> l
[1, 2, 3, 1]
```

Creating lists

```
L = []
L = [value, …]
L = list()
```

Accessing lists

The length of a list can be obtained by using len() method.

```
>>> len(L)          #length of List will be returned
```

The elements of a list can be accessed by their indices.

The **index** is the numbered position of a letter in the string. In Python, indices begin 0 onwards to (len(L) – 1) in the forward direction and -1,-2, …, -len(L) in the backward direction.

```
>>> print(L[0:n])
```

This will print all the elements from the first element to (n-1)th index element.

Traversing a List

```
L = ['P', 'y', 't', 'h', 'o', 'n']
for a in L:
    print(a)
```

Output:

```
P
y
t
h
o
n
```

If we want t use the indexes of elements to access them we can use functions range() and len() as per following syntax:

```
for index in range(len(L)):
    print(L[index])
```

List Methods

Difference between append() and extend()

Lists have several methods amongst which the **append()** and **extend()** methods. The former appends an object to the end of the list (e.g., another list) while the later appends each element of the iterable object (e.g., anothee list) to the end of the list.

For example, we can append an object (here the character 'c') to the end of a simple list as follows:

```
>>> stack = ['a','b']
>>> stack.append('c')
>>> stack
['a', 'b', 'c']
```

However, if we want to append several objects contained in a list, the result as expected (or not...) is:

```
>>> stack.append(['d', 'e', 'f'])
>>> stack
['a', 'b', 'c', ['d', 'e', 'f']]
```

The object `['d', 'e', 'f']` has been appended to the exiistng list. However, it happens that sometimes what we want is to append the elements one by one of a given list rather the list itself. You can do that manually of course, but a better solution is to use the **extend()** method as follows:

```
>>> stack = ['a', 'b', 'c']
>>> stack.extend(['d', 'e','f'])
>>> stack
['a', 'b', 'c', 'd', 'e', 'f']
```

index

The **`index()`** methods searches for an element in a list. For instance:

```
>>> my_list = ['a','b','c','b', 'a']
>>> my_list.index('b')
1
```

It returns the index of the first and only occurrence of 'b'. If you want to specify a range of valid index, you can indicate the start and stop indices:

```
>>> my_list = ['a','b','c','b', 'a']
>>> my_list.index('b', 2)
3
```

Warning

if the element is not found, an error is raised

insert

You can remove element but also insert element wherever you want in a list:

```
>>> my_list.insert(2, 'a')
>>> my_list
['b', 'c', 'a', 'b']
```

The `insert()` methods insert an object before the index provided.

remove

Similarly, you can remove the first occurence of an element as follows:

```
>>> my_list.remove('a')
>>> my_list
['b', 'c', 'b', 'a']
```

pop

Or remove the last element of a list by using:

```
>>> my_list.pop()
'a'
>>> my_list
['b', 'c', 'b']
```

which also returns the value that has been removed.

count

You can count the number of element of a kind:

```
>>> my_list.count('b')
2
```

sort

There is a `sort()` method that performs an in-place sorting:

```
>>> my_list.sort()
>>> my_list
['a', 'b', 'b', 'c']
```

Here, it is quite simple since the elements are all characters. For standard types, the sorting works well. Imagine now that you have some non-standard types. You can overwrite the function used to perform the comparison as the first argument of the `sort()` method.

There is also the possiblity to sort in the reverse order:

```
>>> my_list.sort(reverse=True)
>>> my_list
['c', 'b', 'b', 'a']
```

reverse

Finally, you can reverse the element in-place:

```
>>> my_list = ['a', 'c' ,'b']
>>> my_list.reverse()
>>> my_list
['b', 'c', 'a']
```

Operators

The + operator can be used to **extend** a list:

```
>>> my_list = [1]
>>> my_list += [2]
>>> my_list
[1, 2]
>>> my_list += [3, 4]
>>> my_list
[1, 2, 3, 4]
```

The * operator ease the creation of list with similar values

```
>>> my_list = [1, 2]
>>> my_list = my_list * 2
>>> my_list
[1, 2, 1, 2]
```

Slicing

Slicing uses the symbol: to access to part of a list:

```
>>> list[first index:last index:step]
>>> list[:]
>>> a = [0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5]
>>> a[2:]
[2, 3, 4, 5]
>>> a[:2]
[0, 1]
>>> a[2:-1]
[2, 3, 4]
```

By default, the first index is 0, the last index is the last one..., and the step is 1. The step is optional. So, the following slicing are equivalent:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8]
>>> a[:]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> a[::1]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> a[0::1]
[1, 2, 3, 4, 5, 6, 7, 8]
```

List comprehension

Traditionally, a piece of code that loops over a sequence could be written as follows:

```
>>> evens = []
>>> for i in range(10):
...     if i % 2 == 0:
...         evens.append(i)
>>> evens
[0, 2, 4, 6, 8]
```

This may work, but it actually makes things slower for Python because the interpreter works on each loop to determine what part of the sequence has to be changed.

A **list comprehension** is the correct answer:

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

Besides the fact that it is more efficient, it is also shorter and involves fewer elements.

Filtering Lists

```
>>> li = [1, 2]
>>> [elem*2 for elem in li if elem>1]
[4]
```

Lists as Stacks

The **Python documentation** gives an example of how to use lists as stacks, that is a last-in, first-out data structures (**LIFO**).

An item can be added to a list by using the `append()` method. The last item can be removed from the list by using the `pop()` method without passing any index to it.

```
>>> stack = ['a','b','c','d']
>>> stack.append('e')
>>> stack.append('f')
>>> stack
['a', 'b', 'c', 'd', 'e', 'f']
>>> stack.pop()
'f'
>>> stack
['a, 'b', 'c', 'd', 'e']
```

Lists as Queues

Another usage of list, again presented in **Python documentation** is to use lists as queues, that is a first in - first out (**FIFO**).

```
>>> queue = ['a', 'b', 'c', 'd']
>>> queue.append('e')
>>> queue.append('f')
>>> queue
['a', 'b', 'c', 'd', 'e', 'f']
>>> queue.pop(0)
'a'
```

How to copy a list

There are three ways to copy a list:

```
>>> l2 = list(l)
>>> l2 = l[:]
>>> import copy
>>> l2 = copy.copy(l)
```

Warning

Don't do l2 = l, which is a reference, not a copy.

The preceding techniques for copying a list create ***shallow copies***. IT means that nested objects will not be copied. Consider this example:

```
>>> a = [1, 2, [3, 4]]
>>> b = a[:]
>>> a[2][0] = 10
>>> a
[1, 2, [10, 4]]
>>> b
[1, 2, [10, 4]]
```

To get around this problem, you must perform a deep copy:

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
>>> a[2][0] = 10
>>> a
[1, 2, [10, 4]]
>>> b
[1, 2, [3, 4]]
```

Inserting items into a sorted list

The **`bisect`** module provides tools to manipulate sorted lists.

```
>>> x = [4, 1]
>>> x.sort()
>>> import bisect
>>> bisect.insort(x, 2)
>>> x
[1, 2, 4]
```

To know where the index where the value would have been inserted, you could have use:

```
>>> x = [4, 1]
>>> x.sort()
>>> import bisect
>>> bisect.bisect(x, 2)
2
```

TUPLES

Tuples are immutable sequences i.e.; you cannot change of elements of a tuple in place.

Constructing tuples

To create a tuple, place values within brackets:

```
>>> l = (1, 2, 3)
>>> l[0]
1
```

It is also possible to create a tuple without parentheses, by using commas:

```
>>> l = 1, 2
>>> l
(1, 2)
```

If you want to create a tuple with a single element, you must use the comma:

```
>>> singleton = (1, )
```

You can repeat a tuples by multiplying a tuple by a number:

```
>>> (1,) * 5
(1, 1, 1, 1, 1)
```

Note that you can concatenate tuples and use augmented assignment (*=, +=):

```
>>> s1 = (1,0)
>>> s1 += (1,)
>>> s1
(1, 0, 1)
```

Tuple methods

Tuples are optimised, which makes them very simple objects. There are two methods available only:

- *index, to find occurrence of a value*
- *count, to count the number of occurrence of a value*

```
>>> l = (1,2,3,1)
>>> l.count(1)
2
>>> l.index(2)
1
```

Interests of tuples

So, Tuples are useful because there are

- *faster than lists*
- *protect the data, which is immutable*
- *tuples can be used as keys on dictionaries*

In addition, it can be used in different useful ways:

Tuples as key/value pairs to build dictionaries

```
>>> d = dict([('jan', 1), ('feb', 2), ('march', 3)])
>>> d['feb']
2
```

signing multiple values

```
>>> (x,y,z) = ('a','b','c')
>>> x
'a'
>>> (x,y,z) = range(3)
>>> x
0
```

Tuple Unpacking

Tuple unpacking allows to extract tuple elements automatically is the list of variables on the left has the same number of elements as the length of the tuple

```
>>> data  = (1,2,3)
>>> x, y, z = data
>>> x
1
```

Tuple can be use as swap function

This code reverses the contents of 2 variables x and y:

```
>>> (x,y) = (y,x)
```

Warning

Consider the following function:

```
def swap(a, b):
    (b, a) = (a, b)
```

then:

```
a = 2
b = 3
swap(a, b)
#a is still 2 and b still 3 !! a and b are indeed passed by value not reference.
```

Misc

length

To find the length of a tuple, you can use the `len()` function:

```
>>> t= (1,2)
>>> len(t)
2
```

Slicing (extracting a segment)

```
>>> t = (1,2,3,4,5)
>>> t[2:]
(3, 4, 5)
```

Copy a tuple

To copy a tuple, just use the assignement:

```
>>> t = (1, 2, 3, 4, 5)
>>> newt = t
>>> t[0] = 5
>>> newt
(1, 2, 3, 4, 5)
```

Warning

You cannot copy a list with the = sign because lists are mutable. The = sign creates a reference not a copy. Tuples are immutable therefore a = sign does not create a reference but a copy as expected.

Tuple are not fully immutable!!

If a value within a tuple is mutable, then you can change it:

```
>>> t = (1, 2, [3, 10])
>>> t[2][0] = 9
>>> t
(1, 2, [9, 10])
```

Convert a tuple to a string

You can convert a tuple to a string with either:

```
>>> str(t)
```

or

```
>>> `t`
```

math and comparison

comparison operators and mathematical functions can be used on tuples. Here are some examples:

```
>>> t = (1, 2, 3)
>>> max(t)
3
```

Dictionary

Dictionaries are mutable, unordered collections with elements in the form of a key:value pairs that associate keys to values.

Quick example

A dictionary is a sequence of items. Each item is a pair made of a key and a value. Dictionaries are not sorted. You can access to the list of keys or values independently.

```
>>> d = {'first':'string value', 'second':[1,2]}
>>> d.keys()
['first', 'second']
>>> d.values()
['string value', [1, 2]]
```

You can access to the value of a given key as follows:

```
>>> d['first']
'string value'
```

Warning

You can not have duplicate keys in a dictionary

Warning

dictionary have no concept of order among elements

Methods to query information

In addition to **keys** and **values** methods, there is also the **items** method that returns a list of items of the form (key, value). The items are not returned in any particular order:

```
>>> d = {'first':'string value', 'second':[1,2]}
>>> d.items()
[('a', 'string value'), ('b', [1, 2])]
```

The **iteritems** method works in much the same way, but returns an iterator instead of a list. See **iterators** section for an example.

You can check for the existence of a specific key with **has_key**:

```
>>> d.has_key('first')
True
```

The expression **d.has_key(k)** is equivalent to **k in d**. The choice of which to use is largely a matter of taste.

In order to get the value corresponding to a specific key, use **get** or **pop**:

```
>>> d.get('first')  # this method can set an optional value, if the key is not found
'string value'
```

It is useful for things like adding up numbers:

```
sum[value] = sum.get(value, 0) + 1
```

The difference between **get** and **pop** is that **pop** also removes the corresponding item from the dictionary:

```
>>> d.pop('first')
'string value'
>>> d
{'second': [1, 2]}
```

Finally, **popitem** removes and returns a pair (key, value); you do not choose which one because a dictionary is not sorted

```
>>> d.popitem()
('a', 'string value')
>>> d
{'second': [1, 2]}
Methods to create new dictionary
```

Since dictionaries (like other sequences) are objects, you should be careful when using the affectation sign:

```
>>> d1 = {'a': [1,2]}
>>> d2 = d1
>>> d2['a'] = [1,2,3,4]
>>> d1['a]
[1,2,3,4]
```

To create a new object, use the **copy** method (shallow copy):

```
>>> d2 = d1.copy()
```

`copy.deepcopy()`

You can clear a dictionary (i.e., remove all its items) using the **`clear()`** method:

```
>>> d2.clear()
{}
```

The **`clear()`** method deletes all items whereas **`del()`** deletes just one:

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> del d['a']
>>> d.clear()
```

Create a new item with default value (if not provided, None is the default):

```
>>> d2.setdefault('third', '')
>>> d2['third']
''
```

Create a dictionary given a set of keys:

```
>>> d2.fromkeys(['first', 'second'])
```

another syntax is to start from an empty dictionary:

```
>>> {}.fromkeys(['first', 'second'])
{'first': None, 'second': None}
```

Just keep in ,ind thqt the **`fromkeys()`** method creates a new dictionary with the given keys, each with a default corresponding value of None.

Combining dictionaries

Given 2 dictionaries d1 and d2, you can add all pairs of key/value from d2 into d1 by using the update method (instead of looping and assigning each pair yourself:

```
>>> d1 = {'a':1}
>>> d2 = {'a':2; 'b':2}
>>> d1.update(d2)
>>> d1['a']
2
>>> d2['b']
2
```

The items in the supplied dictionary are added to the old one, overwriting any items there with the same keys.

iterators

Dictionary provides iterators over values, keys or items:

```
>>> [x for x in t.itervalues()]
['string value', [1, 2]]
>>>
>>> [x for x in t.iterkeys()]
['first', 'csecond']
```

```
>>> [x for x in t.iteritems()]
[('a', 'string value'), ('b', [1, 2])]
```

Views

viewkeys, viewvalues, viewitems are set-like objects providing a view on D's keys, values or items.

comparison

you can compare dictionaries! Python first compares the sorted key-value pairs. It first sorts dictionaries by key and comapre their initial items. If the items hae different values, Python figures out the comparison between them. Otherwise, the second items are compared and so on.

STRINGS

Strings are an immutable sequence of characters

Creating a string (and special characters)

Single and double quotes are special characters. There are used to defined strings. There are actually 3 ways to define a string using either single, double or triple quotes:

```
text = 'The surface of the circle is 2 pi R = '
text = "The surface of the circle is 2 pi R = "
text = '''The surface of the circle is 2 pi R = '''
```

In fact the latest is generally written using triple double quotes:

```
text = """The surface of the circle is 2 pi R = """
```

Strings in double quotes work exactly the same as in single quotes but allow to insert single quote character inside them.

The interest of the triple quotes (''' or """) is that you can specify multi-line strings. Moreover, single quotes and double quotes can be used freely within the triple quotes:

```
text = """ a string with special character " and ' inside """
```

The " and ' characters are part of the Python language; they are special characters. To insert them in a string, you have to escape them (i.e., with a \ chracter in front of them to indicate the special nature of the character). For instance:

```
text = "a string with escaped special character \", \' inside "
```

There are a few other special characters that must be escaped to be included in a string.
To include unicode, you must precede the string with the **u** character:

```
>>> u"\u0041"
A
```

Note

unicode is a single character set used to represent 65536 different characters.

Similarly, you may see strings preceded by the **r** character to indicate that the string has to be interpreted as it is without interpreting the special character ****.
This is useful for docstrings that contain latex code for instance:

```
r" \textbf{this is bold text in LaTeX} "
```

Strings are immutable

You can access to any character using slicing:

```
text[0]
text[-1]
text[0:]
```

However, you cannot change any character:

```
text[0] = 'a' #this is incorrect.
```

Formatter

In Python, the % sign lets you produce formatted output. A quick example will illustrate how to print a formatted string:

```
>>> print("%s" % "some text")
"some text"
```

The syntax is simply:

```
string % values
```

If you have more than one value, they should be placed within brackets:

```
>>> print("%s %s" % ("a", "b"))
```

The string contains characters and *conversion specifiers* (here %s)

To escape the sign %, just double it:

```
>>> print "This is a percent sign: %%"
This is a percent sign: %
```

There are different ways of formatting a string with arguments. The one based on a string method called **format()** is more and more common:

```
>>> "{a}!={b}".format(a=2, b=1)
2!=1
```

Operators

The mathematical operators `+` and `*` can be used to create new strings:

```
t = 'This is a test'
t2 = t+t
t3 = t*3
```

and comparison operators `>`, `>=`, `==`, `<=`, `<` and `!=` can be used to compare strings.

Methods

The string methods are numerous, however, many of them are similar (as you will see in this page).

Methods to query information

There are a few methods to check the type of alpha numeric characters present in a string:

`isdigit()`, **`isalpha()`**, **`islower()`**, **`isupper()`**, **`istitle()`**, **`isspace()`**, **`str.isalnum()`**

```
>>> "44".isdigit()  # is the string made of digits only ?
True
>>> "44".isalpha()  # is the string made of alphabetic characters
only ?
False
>>> "44".isalnum()  # is the string made of alphabetic characters
or digits only ?
True
>>> "Aa".isupper()  # is it made of upper cases only ?
False
>>> "aa".islower()  # or lower cases only ?
True
>>> "Aa".istitle()  # does the string start with a capital letter
?
True
>>> text = "There are spaces but not only"
>>> text.isspace() # is the string made of spaces only ?
False
```

You can count the occurence of a character with **`count()`** or get the length of a string with **`len()`**:

```
>>> mystr = "This is a string"
>>> mystr.count('i')
3
>>> len(mystr)
16
```

Methods that return a modified version of the string

The following methods return modified copy of the original string, which is immutable.

First, you can modify the cases
using **`title()`**, **`capitalize()`**, **`lower()`**, **`upper()`** and **`swapcase()`**:

```
>>> mystr = "this is a dummy string"
>>> mystr.title()          # return a titlecase version of the
string
'This Is A Dummy String'
>>> mystr.capitalize()    # return a string with first letter
capitalised only.
'This is a dummy string'
>>> mystr.upper()         # return a capitalised version of the
string
'THIS IS A DUMMY STRING'
>>> mystr.lower()        # return a copy of the string converted
to lower case
'this is a dummy string'
>>> mystr.swapcase()     # replace lower case by upper case and
vice versa
'THIS IS A DUMMY STRING'
```

Second, you can add trailing characters with **`center()`** and **`just()`** methods:

```
>>> mystr = "this is a dummy string"
>>> mystr.center(40)             # center the string in a string
of length 40
'         this is a dummy string         '
>>> mystr.ljust(30)                # justify the string to the
left (width of 20)
'this is a dummy string        '
>>> mystr.rjust(30, '-')           # justify the string to the
right (width of 20)
'--------this is a dummy string'
```

There is also a **`zfill()`** methods that adds zero to the left, which is equivalent to `.rjust(width, '0')`:

```
>>> mystr.zfill(30)
'00000000this is a dummy string'
```

or remove trailing spaces with the **`strip()`** methods:

```
>>> mystr = "  string with left and right spaces   "
>>> mystr.strip()
'string with left and right spaces'
>>> mystr.rstrip()
'  string with left and right spaces'
```

```
>>> mystr.lstrip()
'string with left and right spaces   '
```

or expand tabs with **expandtabs()**:

```
>>> 'this is a \t tab'.expandtabs()
'this is a     tab'
```

You can remove some specific characters with **translate()** or replace words with **replace()**:

```
>>> mystr = "this is a dummy string"
>>> mystr.replace('dummy', 'great', 1)   # the 1 means replace only once
'this is a great string'
>>> mystr.translate(None, 'aeiou')
ths s dmmy strng
```

Finally, you can separate a string with respect to a single separator with **partition()**:

```
>>> mystr = "this is a dummy string"
>>> t.partition('is')
('th', 'is', ' is a line')
>>> t.rpartition('is')
('this ', 'is', ' a line')
```

Methods to find position of substrings

There are methods such as **endswith()**, **startswith()**, **find()** and **index()** that allow to search for substrings in a string.

```
>>> mystr = "This is a dummy string"
>>> mystr.endswith('ing')       # may provide optional start and end indices
True
>>> mystr.startswith('This')      # may provide start and end indices
True
>>> mystr.find('is')              # returns start index of 'is' first occurence
2
>>> mystr.find('is', 4)          # starting at index 4, returns start index of 'is' first occurence
5
>>> mystr.rfind('is')             # returns start index of 'is' last occurence
5
>>> mystr.index('is')             # like find but raises error when substring is not found
2
>>> mystr.rindex('is')            # like rfind but raises error when substring is not found
5
```

Methods to build or decompose a string

A useful function is the **split()** methods that splits a string according to a character. The inverse function exist and is called **join()**.

```
>>> message = ' '.join(['this' ,'is', 'a', 'useful', 'method'])
>>> message
'this is a useful method'
>>> message.split(' ')
['this', 'is', 'a', 'useful', 'method']
```

The **split()** function can be applied to a limited number of times if needed. However, it starts from the left. If you want to start from the right, use **rsplit()** instead

```
>>> message = ' '.join(['this' ,'is', 'a', 'useful', 'method'])
>>> message.rsplit(' ', 2)
['this is a', 'useful', 'method']
```

If a string is multi-lines, you can split it with **splitlines()**:

```
>>> 'this is an example\n of\ndummy sentences'.splitlines()
['this is an example', ' of', 'dummy sentences']
```

you can keep the endline character by giving True as an optional argument.

Finally, note that **split()** removes the splitter:

```
>>> "this is an exemple".split(" is ")
['this', 'an exemple']
```

If you want to keep the splitter as well, use **partition()**

```
>>> "this is an exemple".partition(" is ")
```

('this', ' is ', 'an exemple')

Encoding/Decoding/Unicode

We've seen how to create a unicode by adding the letter **u** in front of a string:

```
s = u"\u0041"
```

The function **unicode()** converts a standard string to unicode string using the encoding specified as an argument (default is the default string encoding):

```
s = unicode("text", "ascii")
```

In order to figure out the default encoding, type:

```
>>> import sys
>>> sys.getdefaultencoding()
'ascii'
```

Here are some encodings:

```
ascii, utf-8, iso-8859-1, latin-1, utf-16, unicode-escape.
```

The unicode function takes also a third argument set to: ‘strict’, ‘ignore’ or ‘replace’.

Let us take another example with accents:

```
>>> # Let us start wil a special character.
>>> text = u"π"
>>> # to obtain its code (in utf-8), let us use the encode function
>>> encoded = text.encode("utf-8")
>>> decoded = text.decode("utf-8")
```

SETS

Sets are constructed from a sequence (or some other iterable object). Since sets cannot have duplicated, there are usually used to build sequence of unique items (e.g., set of identifiers).

Quick example

```
>>> a = set([1, 2, 3, 4])
>>> b = set([3, 4, 5, 6])
>>> a | b # Union
{1, 2, 3, 4, 5, 6}
>>> a & b # Intersection
{3, 4}
>>> a < b # Subset
False
>>> a - b # Difference
{1, 2}
>>> a ^ b # Symmetric Difference
{1, 2, 5, 6}
```

Note

the intersection, subset, difference and symmetric difference can be be called with method rather that symbols. See below for examples.

Ordering

Just as with dictionaries, the ordering of set elements is quite arbitrary, and shouldn’t be relied on.

Operators

As mentionned in the quick example section, each operator is associated to a symbol (e.g., &) and a method name (e.g. union).

```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])
>>> c = a.intersection(b) # equivalent to c = a & b
>>> a.intersection(b)
set([2, 3])
>>> c.issubset(a)
True
>>> c <= a
True
>>> c.issuperset(a)
False
>>> c >= a
False
>>> a.difference(b)
set([1])
>>> a - b
set([1])
>>> a.symmetric_difference(b)
set([1, 4])
>>> a ^ b
set([1, 4])
```

You can also copy a set using the copy method:

```
>>> a.copy()
set([1, 2, 3])
```