Python CheatSheet (For Linux)
Beginning Python: From Novice to Professional
Chapter 2(Lists and Tuples)

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number-its position, or index. Python has six built-in types of sequences. This chapter concentrates on two of the most common ones: lists and tuples. The main difference between lists and tuples is that you can change a list, but you can't change a tuple.

Written as a list (**the items of a list are separated by commas and enclosed in square brackets**), that would look like this:

```
>>> edward = ['Edward Gumby', 42]
```

But sequences can contain other sequences, too, so you could make a list of such persons, which would be your database:

```
>>> edward = ['Edward Gumby', 42]
>>> john = ['John Smith', 50]
>>> database = [edward, john]
>>> database
[['Edward Gumby', 42], ['John Smith', 50]]
```

Python has a basic notion of a kind of data structure called a **container**, which is basically any object that can contain other objects. The two main kinds of **containers are sequences (such as lists and tuples) and mappings (such as dictionaries).**

Common Sequence Operations

There are certain things you can do with all sequence types. These operations include **indexing, slicing, adding, multiplying, and checking** for membership. In addition, Python has built-in functions for **finding the length of a sequence, and for finding its largest and smallest elements**.

Indexing

All elements in a sequence are numbered **from zero and upwards**. You can access them individually with a number, like this:

```
>>> greeting = 'Hello'
>>> greeting[0]
'H'
```

A string is just a sequence of characters. The index **0 refers to the first element**, in this case the letter H. This is indexing. You use an index to fetch an element. All sequences can be indexed in this way.
When you use a negative index, Python counts from the right; that is, from the last element. The last element is at position –1 (not –0, as that would be the same as the first element):

```
>>> greeting[-1]
'o'
```

String literals (and other sequence literals, for that matter) may be indexed directly, without using a variable to refer to them. The effect is exactly the same:

```
>>> 'Hello'[1]
'e'
```

If a function call returns a sequence, you can index it directly. For instance, if you are simply interested in the fourth digit in a year entered by the user, you could do something like this:

```
>>> fourth = raw_input('Year: ')[3]
Year: 2005
>>> fourth
'5'
```

Slicing

Use slicing to access ranges of elements. You can do this by using two indices, separated by a colon. Slicing is very useful for extracting parts of a sequence. The numbering here is very important. The first index is the number of the first element you want to include. However, the last index is the number of the first element after your slice.

```
>>>numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>numbers[3:6]
[4, 5, 6]
>>>numbers[0:1]
>>>[1]
```

In short, you supply two indices as limits for your slice, where the first is inclusive and the second is exclusive. Let’s say you want to access the last three elements of numbers (from the previous example). You could do it explicitly, of course:

```
>>> numbers[7:10]
[8, 9, 10]
```

Now, the index 10 refers to element 11—which does not exist, but is one step after the last element you want. Got it? This is fine, but what if you want to count from the end?

```
>>> numbers[-3:-1]
[8, 9]
```

It seems you cannot access the last element this way. How about using 0 as the element “one step beyond” the end?

```
>>> numbers[-3:0]
[]
```

Not exactly the desired result. In fact, any time the leftmost index in a slice comes later in the sequence than the second one (in this case, the third-to-last coming later than the first), the result is always an empty sequence. Luckily, you can use a shortcut: if the slice continues to the end of the sequence, you may simply leave out the last index:

```
>>> numbers[-3:]
[8, 9, 10]
```

The same thing works from the beginning:

```
>>> numbers[:3]
[1, 2, 3]
```

In fact, if you want to copy the entire sequence, you may leave out both indices:

```
>>> numbers[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Longer Steps

When slicing, you specify (either explicitly or implicitly) the start and end points of the slice.
Another parameter (added to the built-in types in Python 2.3), which normally is left implicit,
is the step length.

```
>>> numbers[0:10:1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[0:10:2]
[1, 3, 5, 7, 9]
numbers[3:6:3]
[4]
```

Adding Sequences

Sequences can be concatenated with the addition (plus) operator:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello, ' + 'world!'
'Hello, world!'
>>> [1, 2, 3] + 'world!'
```

```
Traceback (innermost last):
File "<pyshell#2>", line 1, in ?
[1, 2, 3] + 'world!'
```
TypeError: can only concatenate list (not "string") to list.As you can see from the error message, you can't concatenate a list and a string, although both are sequences. In general, you cannot concatenate sequences of different types.

Multiplication

Multiplying a sequence by a number x creates a new sequence where the original sequence is repeated x times:

```
>>> 'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

None is a Python value and means exactly that—"nothing here.

Membership

To check whether a value can be found in a sequence, you use the **in** operator. It checks whether something is true and returns a value accordingly: True for true and False for false.

```
>>> permissions = 'rw'
>>> 'w' in permissions
True
```

Length, Minimum, and Maximum

The built-in functions len, min, and max can be quite useful. The function len returns the number of elements a sequence contains. min and max return the smallest and largest element of the sequence, respectively.

Lists are mutable—that is, you can change their contents—and they have many useful specialized methods.

The list Function

Because strings can't be modified in the same way as lists, sometimes it can be useful to create
a list from a string. You can do this with the list function:

```
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

Note that list works with all kinds of sequences, not just strings.

Changing Lists: Item Assignments

Changing a list is easy.

```
>>>x = [1, 1, 1]
>>>x[1] = 2
>>>x
[1, 2, 1]
```

Deleting Elements

Deleting elements from a list is easy, too. You can simply use the del statement:

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
>>> del names[2]
>>> names
['Alice', 'Beth', 'Dee-Dee', 'Earl']
```

Assigning to Slices

Slicing is a very powerful feature, and it is made even more powerful by the fact that you can assign to slices:

```
>>> name = list('Perl')
>>> name[1:] = list('ython')
>>> name
['P', 'y', 't', 'h', 'o', 'n']
```

List Methods

A method is a function that is tightly coupled to some object, be it a list, a number, a string, or whatever.

Append

The append method is used to append an object to the end of a list:

```
>>>lst = [1, 2, 3]
>>>lst.append(4)
>>>lst
[1, 2, 3, 4]
```

Count

The count method counts the occurrences of an element in a list:

```
>>>x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
>>>x.count(1)
2
>>x.count([1, 2])
1
```

Extend

The extend method allows you to append several values at once by supplying a sequence of the values you want to append. In other words, your original list has been extended by the other one:

```
>>>a = [1, 2, 3]
>>>b = [4, 5, 6]
>>>a.extend(b)
>>>a
[1, 2, 3, 4, 5, 6]
```

This may seem similar to concatenation, but the important difference is that the extended sequence (in this case, a) is modified. This is not the case in ordinary concatenation, in which a completely new sequence is returned:

```
>>>a = [1, 2, 3]
>>>b = [4, 5, 6]
>>>a + b
[1, 2, 3, 4, 5, 6]
>>>a
[1, 2, 3]
```

Index

The index method is used for searching lists to find the index of the first occurrence of a value:

```
>>> knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']
>>> knights.index('who')
4
```

Insert

The insert method is used to insert an object into a list:

```
>>>numbers = [1, 2, 3, 5, 6, 7]
>>>numbers.insert(3, 'four')
>>>numbers
[1, 2, 3, 'four', 5, 6, 7]
```

As with extend, you can implement insert with slice assignments:

```
>>>numbers[3:3] = ['four']
>>>numbers
[1, 2, 3, 'four', 5, 6, 7]
```

Pop

The pop method removes an element (by default, the last one) from the list and returns it:

```
>>>x = [1, 2, 3]
```

```
>>>x.pop()
3
>>>x
[1, 2]
>>>x.pop(0)
1
>>>x
[2]
```

The pop method is the only list method that both modifies the list and returns a value (other than None). Using pop, you can implement a common data structure called a stack.

Remove

The remove method is used to remove the first occurrence of a value:

```
>>> x = ['to', 'be', 'or', 'not', 'to', 'be']
>>> x.remove('be')
>>> x
```

It's important to note that this is one of the "nonreturning in-place changing" methods. It modifies the list, but returns nothing (as opposed to pop).

Reverse

The reverse method reverses the elements in the list.

```
>>>x = [1, 2, 3]
>>>x.reverse()
>>>x
[3, 2, 1]
```

Note that reverse changes the list and does not return anything (just like remove and sort).

Sort

The sort method is used to sort lists in place. 3 Sorting "in place" means changing the original list so its elements are in sorted order, rather than simply returning a sorted copy of the list:

```
>>>x = [4, 6, 2, 1, 7, 9]
>>>x.sort()
>>>x
[1, 2, 4, 6, 7, 9]
>>> y = x.sort() # Don't do this!
>>>y = x[:]
>>>y.sort()
>>>x
[4, 6, 2, 1, 7, 9]
>>>y
[1, 2, 4, 6, 7, 9]
```

This function can actually be used on any sequence, but will always return a list. The sort method has two other optional arguments: key and reverse.

Cmp

cmp stands for comparision.

```
>>>cmp(42, 32)
1
>>>cmp(99, 100)
-1
>>>cmp(10, 10)
0
```

Tuples

Tuples are sequences, just like lists. The only difference is that tuples can't be changed. (As you may have noticed, this is also true of strings.) The tuple syntax is simple—if you separate some values with commas, you automatically have a tuple:

```
>>> 1, 2, 3
(1, 2, 3)
```

As you can see, tuples may also be (and often are) enclosed in parentheses. So, you may wonder how to write a tuple containing a single value. This is a bit peculiar—you have to include a comma, even though there is only one value. **One lonely comma**, however, can change the value of an expression completely:

```
>>> 3*(40+2)
126
>>> 3*(40+2,)
(42, 42, 42)
```

The tuple Function

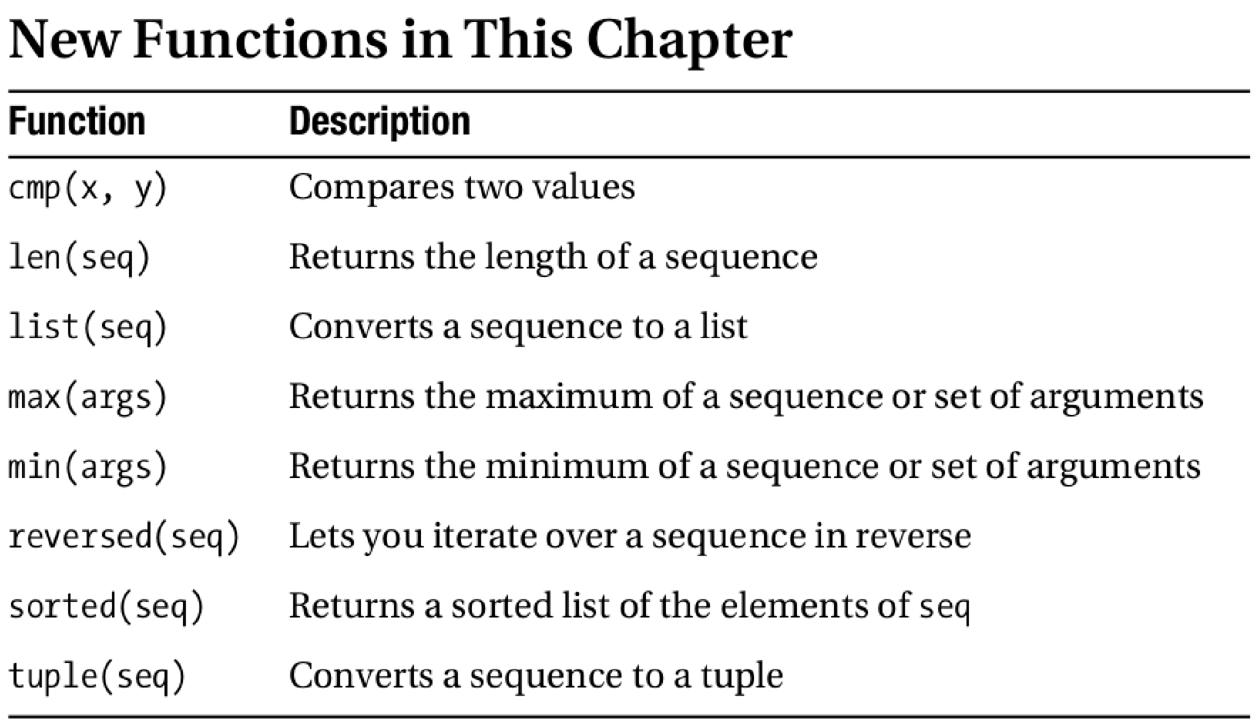
The tuple function works in pretty much the same way as list: it takes one sequence argument and converts it to a tuple. 6 If the argument is already a tuple, it is returned unchanged:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

Tuples operations

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

As you can see, slices of a tuple are also tuples, just as list slices are themselves lists.



New Functions in This Chapter

Function	Description
`cmp(x, y)`	Compares two values
`len(seq)`	Returns the length of a sequence
`list(seq)`	Converts a sequence to a list
`max(args)`	Returns the maximum of a sequence or set of arguments
`min(args)`	Returns the minimum of a sequence or set of arguments
`reversed(seq)`	Lets you iterate over a sequence in reverse
`sorted(seq)`	Returns a sorted list of the elements of `seq`
`tuple(seq)`	Converts a sequence to a tuple