Chapter 5 Notes

Tuples

Tuples are immutable ordered sequences of elements (like strings). The individual elements of a tuple can be of any type and need not be of the same type as each other.

To denote a singleton tuple, we must include a comma after the single element in that tuple. Repetition can be used on tuples. A tuple can be multiplied by an integer to give a repetitive tuple. Tuple can also be indexed and sliced just like strings. The “for” statement can be used to iterate over the elements of a tuple, along with the “in” keyword.

We can assign multiple variables at the same time using tuple: x, y, z = (1, 2, 3) will bound x to 1, y to 2 and z to 3.

Range

Like strings and tuples, ranges are immutable. **The range function returns an object of type range**. All the operations on tuples are also available for ranges, except for concatenation and repetition.

When the == operator is used to compare objects of the type range, it returns True if the two ranges represent the same sequence of integers. However, the two sequences must be in the same order.

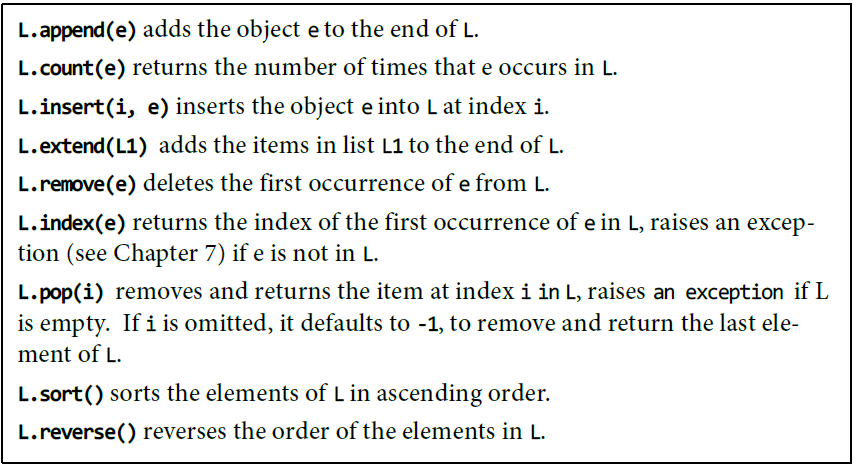
Lists and Mutability

Like a tuple a **list is an ordered sequence of values**, where each value is identified by an index. Lists are represented using square brackets, and there is no need to include a comma for a singleton list. Lists are mutable, which means that their elements can be modified after they are created.

In python, a variable is merely a name assigned to an object. An object, for example a list, can have multiple names. In order to verify an object, we can use the function **id()** to return a unique integer identifier for an object. This function allows us to test for **object equality.**

**Aliasing** is the process of using multiple names on a single object. When aliasing is applied, one can mutate the object via either path, and the effect of the mutation will be visible through all paths.

We can add additional items at the end of a list by using the **.append()** method for a list. We can also expand a list by appending one list to another using the .**extend()** method. You can’t simply append a list into another list because that will result in a nested list. List concatenation with the (+) sign will create a new list, while the .extend() method will mutate the original list.



**Cloning** can be used to help solve problems regarding aliasing. To clone a list, simply use the “everything” slice method **list\_name[:]**. You can also use the **list()** function to return a copy of an existing list.

List Comprehension

**List comprehension** provides a concise way to apply an operation to the values in a sequence. A list comprehension can be used to generate a list of characters using the “for”, “in” and “range” keywords. List comprehension syntax is:

list\_name = [{x operation} for x in range (a, b) if]

The “if” keyword can be use after the range method to modify the sequence of values generated by the first for clause.

Functions as Objects

In Python, functions are **first-class objects**. This means that they can be treated like objects of any other type, e.g., int or list. Using functions as arguments allows a style of coding called higher-order programming.

**The higher-order function** **map()** is used as a tool to map functions into certain non-function arguments, and works in combination with a for loop. When used in a for loop, map behaves like the range function in that it returns one value for each iteration of the loop. The non-function argument is very often a list.

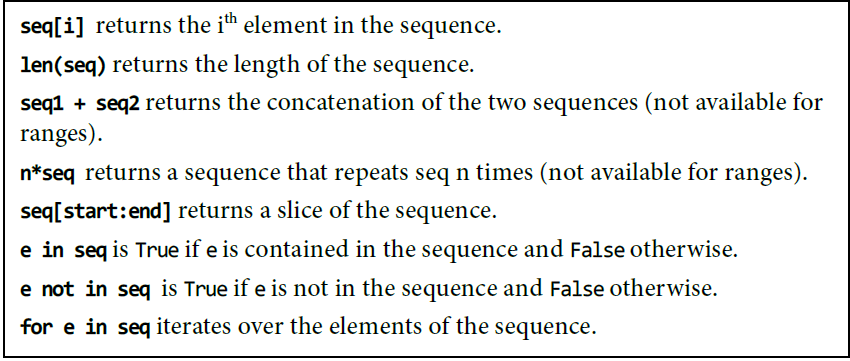
More generally, the first argument to map can be a function of n arguments, in which case it must be followed by n subsequent ordered collections (each of the same length).

**Lambda expression**: This is a special anonymous function (a function that are not bound to a name) and can be created using the keyword lambda. The syntax is:

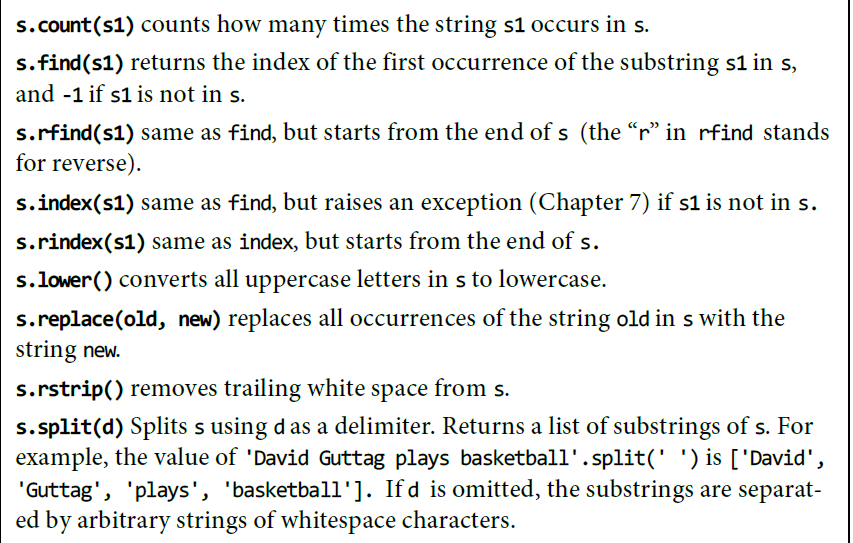
lambda <sequence of variable names>: <expression>

Sequence Type

In Python, there are four different sequence type: str, tuple, range, and list. Some of their shared operators are shown below:



For a string, here are some useful string methods:



Dictionaries

Objects of type **dict** (short for dictionary) are like lists except that we index them using keys. Dictionary is a set of key/value pairs. Literals of type dict are enclosed in curly braces, and each elemt is written as a key followed by a colon followed by a value. The entries in a dict are unordered and cannot be accessed with an index. Dictionaries are mutable. We can change dictionary values by using the key.

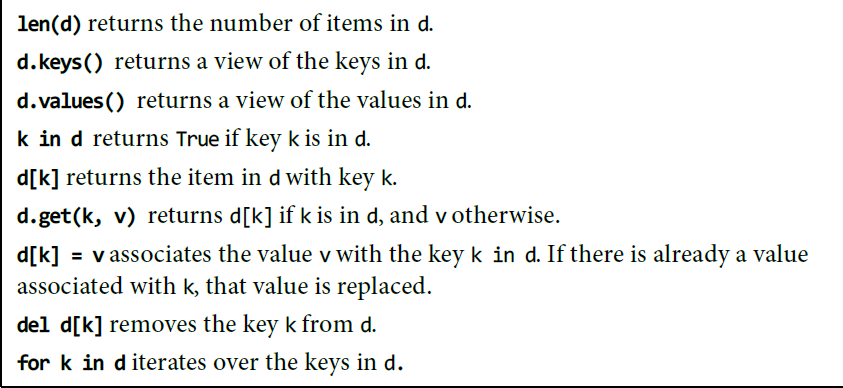
dict\_name = {‘key\_1’:”value\_1”, ‘key\_2’:”value\_2”}

A for statement can be used to iterate over the entries in a dictionary. However, the value assigned to the iteration variable is a key, not a key/value pair. The method **.keys()** return an object of type **dict\_keys.** This is an example of a **view object**. The order in which the keys appear in the view is not defined.

Objects of type dict\_keys can be iterated over using “for”, and membership can be tested using “in”.

Not all types of objects can be used as keys: A key must be an object of a **hashable type**. This basically means that the object must be of a mutable type, in which its value does not change during its lifetime. A type is hashable if it has:

* A \_hash\_ method that maps an object of the type to an int , and for every object the value returned by ­\_hash\_ does not change during the lifetime of the object, and
* An \_eq\_ method that is used to compare objects for equality.

All of Python’s built-in immutable types are hashable, and none of Python’s built-in mutable type are hashable. Below are some common methods on dict: