Session 4
Lists, Sets and Dictionarys

Introduction

If we want to have a program do anything useful we have to deal with some form of data, generally this data will have a common theme; for example we may have a program that deals with pricing information or a list of users. Instead of assigning ever value to a unique variable we can group all of these similar values under one variable by creating an **array**. An array is a generic name for a collection of data, by storing this information together we can combine it with loops to run various functions on large amounts of data.

In a previous session we discussed data types, those types are called **primitive types** due to them representing the most basic of types. There are also multiple types of arrays that store data in different forms and are called **containers**. Containers store information in two ways, as a sequence of data or as a mapping, we will learn more about these things as we look at the three main container types in Python.

All containers have an index which is an internal way of accessing a piece of data in the container (**the element**). In lists and sets each element (an item in the container) is assigned a unique index number. Even if the data is the same, each element has a unique number. This makes it simple to combine loops and containers to access all elements of a list in an **iteration**.

Note

In most programming languages index numbers start from zero, so the first item in a list is element zero and the second is element one.

Lists

The most commonly used type is a list, it is simply a list of unordered data. Similar to a shopping list - it can contain multiple copies of the same item and can be in any order. Lists are usually filled information of the same data type, for example a list of strings is better than a list of some strings and some numbers. Due to the freedom of Python it is possible to mix these primitive data types together inside lists and can cause problem later on. Let's see a small list example -

```
our_list = [1, 2, 3, 4, 5]

print our_list

for i in our_list:
    print i
```

Note

It is common programming practice to use i as the name of your iterator when looping over lists.

In this example we create a list called **our_list** and fill it with 5 separate numbers. The first print statement prints the lists in a more human readable format. The second part of the code is a for loop that we have covered in previous sessions, this loop is filling i with each element in the array l until it reaches the end. The loop effectively prints all numbers in the list.

The key parts of a list are the square angle brackets when initialising the variable, the square brackets signify that it is a special container called as **list**. The list is initialised with five elements by placing them inside the square brackets and separating them with a comma. Lists are **mutable**, by mutable we mean that they can be modified after they are created.

Note

Lists are great for storing data that has no obvious order and contains duplicate elements. There are many techniques for searching lists that are quicker than simply checking every element.

To retrieve data from a list at a specific location is not common but is possible by using the square brackets and the index position of the element you are after.

```
our_list = [1, 32, 4, 566, 22]
print our_list[0]
```

To add data to a list use the **append** function, this function is used for both lists and sets.

```
our_list = [1, 2, 3, 4]
print our_list

our_list.append(5)
print our_list
```

To remove data from a list use the **remove** function and give it the value to be removed. If the list contains more than one element that matches only the first will be removed.

```
our_list = ['python', 'strings', 'are', 'also', 'lists']
print our_list

our_list.remove('python')
print our_list
```

Exercise

In this exercise you will first fill a list of integer numbers and then iterate through the list to find the highest and the lowest values. Save it as Exercise_5.py.

Sets

Sets are a common mathematical type similar to lists in that they can contain any type of data in any order without any duplication. The common usages of sets are to remove duplicates from lists and testing for list items for membership of certain types.

In Python sets are not part of the default language and must be imported as a module. See below for an example of how to use sets taken from the Python website (https://docs.python.org/2/library/sets.html)

```
from sets import Set

engineers = Set(['John', 'Jane', 'Jack', 'Janice'])
programmers = Set(['Jack', 'Sam', 'Susan', 'Janice'])
managers = Set(['Jane', 'Jack', 'Susan', 'Zack'])
employees = engineers | programmers | managers           # union
engineering_management = engineers & managers       #intersection

print employees
print engineering_management
```

Generally lists are used in favour of sets unless the power of sets is a requirement.

Dictionaries

A dictionary is similar to a language dictionary - the way to use a dictionary is to look up the word in question to get to the meaning of that word. A programming dictionary links **dictionary keys** to **dictionary values**, to access a value the programmer uses a key to retrieve the data. Each key in a dictionary must be unique but the values can be duplicated. Dictionaries are very useful for storing large numbers of key/value pairs such as a user profile where creating a variable for each part would not be feasible. Dictionaries in Python are defined by curly braces **{}** and the keys and values can be a mixture of types but generally mixing key types should be avoided.

```
user_profile = {}

user_profile['username'] = 'Jack'
user_profile['last_login'] = 'Yesterday'
user_profile['id'] = 1000

print "User name: {0}".format(user_profile['username'])
```

As you can see in the above example we created a user profile for the user 'Jack' and stored it in a dictionary, then used the dictionary to print some information about the user.

Note

In Python it is possible to do a print statement followed by a variable name to get a full print out of the variable's contents, try this with the above dictionary.

Exercise

Create a dictionary that converts the number to a month name, e.g. given a variable that contains 1 your program should output January. Don't forget to check that the input variable is a valid month number!

To get user string input to your program and convert it into an integer use the following lines in your code:

```
user_input = raw_input("Enter a month number: ")
month = int(user_input)
```

Save your program as Exercise_6.py.

Sorting and Searching

Frequently if we have a list of values we want to search it to see if it contains a value, in Python this is extremely easy but it is important that you understand the different search algorithms that are available and which are better than others. For example if you have a list that contains a million elements it doesn't make sense to go through every element and check it, they are much quicker ways. Searching often goes hand in hand with sorting, a sorted list is a lot easier to search as we can make guesses on the position of the value in a list.

Sorting

To sort a list in Python is, as you might have guessed, extremely simple -

```
random_list = [1, 5, 21, 2, 7, 81, 4, 22]
print "Unordered: {0}".format(random_list)

random_list.sort()
print "Ordered: {0}".format(random_list)
```

In the above example we create a list of unordered numbers and print them, then we call the **sort function** that is available to all lists in Python, this sorts the elements of the list inside the variable; the output of the sort does not need to be assigned to a new variable as you can see when it is printed a second time. Additional to **sort** is the function **sorted** that will return a new sorted list.

Note

It's not just numbers that can be sorted, strings will be sorted alphabetically.

The built in sort functions are generally fast enough in any language but it is helpful to see the difference between available sorting algorithms, some are extremely slow and wasteful where as others use a bit of intelligence to massively speed up the sort. Even with modern computing power it's important to never be wasteful and always write your code to be as fast as possible.

Exercise

In this exercise you will write a very simple sorting algorithm that is designed to show you how not to search! The algorithm is extremely simple and consists of these steps -

1. *Create a new list to contain your sorted values*
2. *Iterate through your list to be sorted*
3. *Store the first value as your minimum value*
4. *Check each element of the unsorted list against the minimum value and replace as required*
5. *When you have reached the end of the list to be sorted place your minimum value in the sorted list*

6. *Repeat steps 2-6 until you have a sorted list*
7. *Output the list to the user*

To complete this exercise you will need two loops, one placed inside the other; this is called a nested loop. See below for an example -

```
unordered_list = [1, 32, 4, 566, 22]
ordered_list = []

while len(unordered_list):
    minimum = unordered_list[0]
    for i in unordered_list:
        pass
```

Len is a function that returns the length of a container. Your task is to fill in the rest of the code that follows the given rules. Save your code as Exercise_7.py.

In Exercise 7 you wrote a search function which probably completed 'instantly' on your machine, in fact this search function is the slowest search function feasible. What the function does is go through the list N times with N being the length of the list, there are much quicker ways than this example.

Searching

Searching, like sorting is very simple and fast in modern languages like Python but much like sorting there are right ways to do it and wrong ways. Searching is a very interesting problem in computing - each time you make a Google search it is looking through literally billions of websites to give you the results you want, this takes less than a second each time which simply wouldn't be possible if it was checking each website in sequence. Fast searching is still a billion dollar industry so it is an important topic to understand and something that we will come back to later.

First let's see searching in Python -

```
unordered_list = [1, 32, 4, 566, 22]

print 1 in unordered_list
print 2 in unordered_list
```

In the above example we are using the **in** keyword to check if an item is in a list and then give a true or false value; this works for lists, sets and dictionaries. Searching using the in keyword works on unsorted lists but searching on sorted lists is much quicker.

Exercise

For this exercise you must write a loop that searches through all elements in a list to find the matching item, make sure to print out if the item is found. This will be an example of a very basic search algorithm. Save your code as Exercise_8.py.