



Introduction to programming

Data structures and algorithms

More about lists

You learnt last week about how lists can be useful to store multiple values under one variable name. Lists in Python, and most other programming languages, come built in with some useful functions.

Function	Description
<code>list.append(value)</code>	Add a value to the end of a list
<code>list.insert(position, value)</code>	Insert a value at a certain position in the list
<code>list.remove(value)</code>	Remove the first occurrence of the value from the list
<code>list.pop(position)</code>	Remove the value at the position given and return it's value
<code>list.index(value)</code>	Return the position of the first occurrence of the value given.
<code>list.count(value)</code>	Return the number of occurrences of the value
<code>list.sort()</code>	Sort a list alphabetically or numerically

Some of the above functions **return** a value. This simply means that when you use the function, a value is given back. For example when using the **index** function with a list.

```
phones = ["Samsung", "Apple", "HTC", "Sony", "Microsoft Lumia", "LG", "Blackberry"]
position_of_apple = phones.index("Apple")
print position_of_apple
```

When the program runs, the **index** function looks up the position the value **Apple** has in the list and puts this value into the **position_of_apple** variable. In this example that would mean that the program would print out **1** as the value **Apple** appears second in the list (remember lists are zero-indexed!).

There is a really useful function called **len** in Python which calculates the length of a list. In other words, it can tell you how many items are in a list.

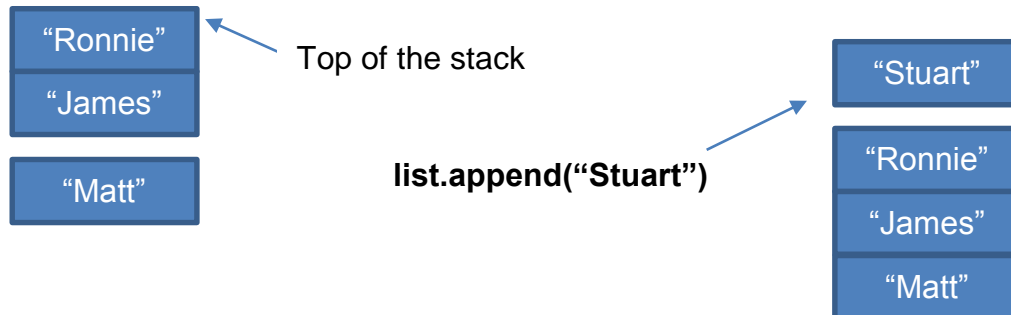
```
phones = ["Samsung", "Apple", "HTC", "Sony", "Microsoft Lumia", "LG", "Blackberry"]
print len(phones)
```



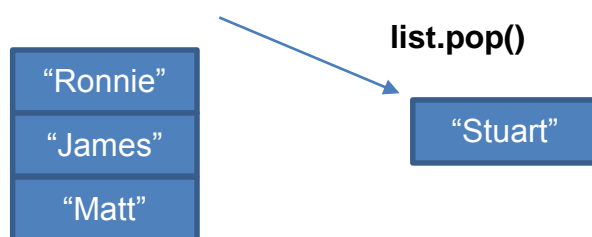
Try using some of the functions in the table above to see what the effect is on the list they are called on.

Stacks

A stack is a data structure that operates much like a real life stack of items, like a stack of books for example. They are described in programming as being **last in, first out** structures as only the top item can be accessed directly. You can implement a stack in Python using the **append** and **pop** functions.



Take for example the above stack of values. You can see that the string **Ronnie** is at the top of the stack and calling the **append** function adds another value to the top.



Using the **pop** function without a value in brackets you can remove the last value that was added to the top of the stack. Here is how the above example could look written in Python:

```
tutors = ["Matt", "James", "Ronnie"]
tutors.pop()
tutors.append("Stuart")
tutors.pop()
print tutors
```

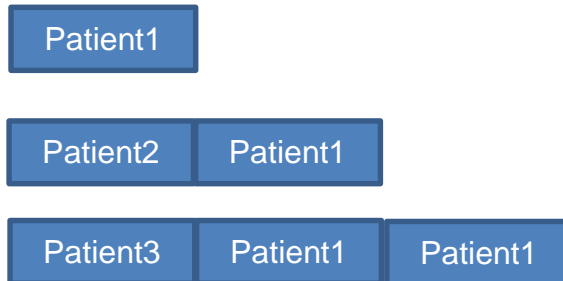
What do you think will be the result printed when the program runs?

An example of when a stack is used is within a word processor when the user uses the undo / redo function or perhaps a [towers of Hanoi game](#).



Queues

A queue can be described as a **first in, first out** data structure as the last value to be added to the queue is the last one to be removed. So the first value that enter the queue are the first to be accessed and removed. Imagine a group of patients entering a waiting room.



In the above diagram, new patients entering the queue are being placed at the beginning and existing patients in the queue are being pushed further towards the top of the queue. Now, when a doctor is ready to see their first patient, the patient at the top of the queue is removed with new patients continuing to be added at the start.



Here is how a queue might look like coded in Python:

```
waiting_room = []

waiting_room.insert(0, "Patient1")
waiting_room.insert(0, "Patient2")
waiting_room.insert(0, "Patient3")

waiting_room.pop()
waiting_room.insert(0, "Patient4")

print waiting_room
```

Uses of queues, include buffers and streams where data is being transferred from one location to another in sequence.

Sets

In Python a set is a list that only contains unique items. There is a **set** function which processes a list and removes any duplicated values.

```
phones = ["Samsung", "Apple", "HTC", "Apple", "Sony", "Microsoft Lumia", "LG", "Sony",
"Apple", "Blackberry"]
print set(phones)
```

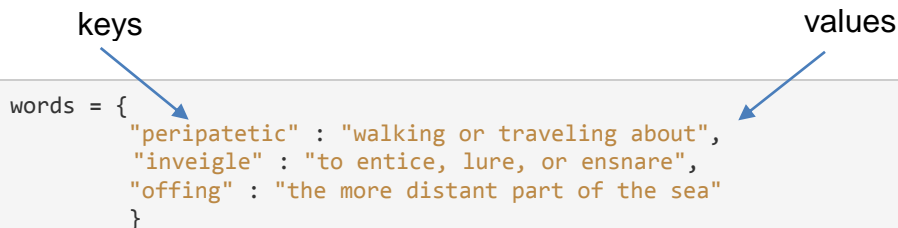


If you run the above program you will see that the duplicate values are removed from the list so **Apple** for example appears only once in the list.

Hashes (dictionaries)

Along with lists (arrays) most programs provide access to another simple data structure called a hash. In Python these are referred to as dictionaries which is a good way to describe how they work. If you think about how you would use an English language dictionary you would find the relevant page that contains a word that you want to look up and on that page a description of the word would be provided.

This is the functionality that a hash data structure provides. There are a list of **values** (the meanings in a dictionary) which are identified by a **key** (the actual word in a dictionary). Here is how a dictionary could look in Python:



```
words = {  
    "peripatetic" : "walking or traveling about",  
    "inveigle" : "to entice, lure, or ensnare",  
    "offing" : "the more distant part of the sea"  
}
```

Note the indentation isn't necessary, it has just been applied for readability. The dictionary looks similar to a list but it uses curly braces `{}` around the key value pairs.

To access a value stored in a dictionary, simply put the key in square brackets after the variable name.

```
print words["offing"]
```

Using the **keys** function on a dictionary will return a list of the keys in the dictionary.

```
print words.keys()
```

Try creating your own dictionary and accessing the values you create in it.

Looping with dictionaries

When looping through a dictionary you will probably want to access both the key and the value. If you use the same bit of code that you used to loop through a list from last week the result might not be what you expected.

```
for w in words:
```



```
print w
# Output is: ['peripatetic', 'offing', 'inveigle']
```

Using the same loop all you can see is a list of the keys in the dictionary. In order to retrieve the key and value each time the loop goes round you need to use the **iteritems** function.

```
for key,value in words.iteritems():
    print key, " => ", value
```

In the above code, using **iteritems** you can get the key and value for each item in the dictionary. For each run of the loop, the key is stored in a variable called **key** and the value in a variable called **value**. These variables could be called anything.

Sorting algorithms

There are many types of sorting algorithms that have benefits and drawbacks in terms of how fast they perform and how complex they are to implement.

Bubble sort

A bubble sort repeatedly goes through each of the values in a list and compares the value next to it to see which one is bigger. If the first value is larger than the second the values swap position in the list (assuming sorting ascending). Once the algorithm has got to the end of the list, it starts again at the beginning making any further swaps. This process continues until no more swaps are needed. Take the list of numbers below as an example.



On the first 'pass' the algorithm will compare **4** and **1**. Four is greater than one so the values are swapped over.



The algorithm then moves on to the next two values. Which after the first swap is **4** and **3**.



Four is still greater than three so the values swap and the algorithm continues to the next two values. The algorithm keeps repeating until there are no more swaps occurring.



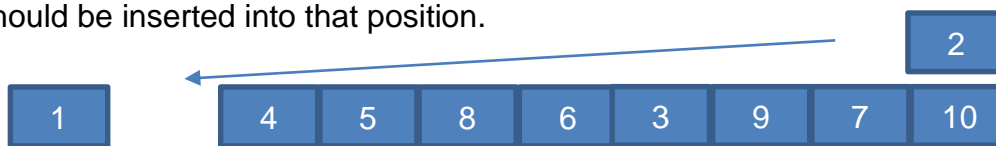
Bubble sorts are very inefficient although simple to implement but they are a good way of demonstrating a basic sorting algorithm.

Insertion sort

An insertion sort works by removing an item from the end of a list and working out where the value should fit into the list.



In the above list of numbers, the number **2** would be removed from the list and then each value, starting with **1** at the start of the list would be checked to see if the value should be inserted into that position.



Insertion sorts are relatively easy to implement and they provide an efficient way of sorting data for small sets of data.

Searching algorithms

Linear search

As the name suggests, a linear search looks through every item in a list in turn until the desired item is found or until the search reaches the end of the list.



Searching for the value **3** in the list, the algorithm will check the first value, 4, which is not equal, then 1, which is not equal before checking the 3rd value which is a 3 so it matches and the value being searched for is found.

The efficiency of a linear search depends on the length of the list being searched.

Binary search

A binary search can only be performed on a sorted list as it works by taking the middle element and then determining whether the value being searched will be in the lower half or top half of the list.





With the sorted list above, if you wanted to search for the value **17** the first step would be for the algorithm to select the middle value which is **45**. The algorithm then checks whether the value you are looking for is less than or greater than 45 and splits the list in half either taking the lower half or upper half. In this example, 17 is less than 45 so the algorithm takes the lower half of the list.



The algorithm repeats taking the middle value of the list again and compares it against the value being searched for. In this case, 17 is less than 28 so take the lower half of the list.



In the final step the value is found.

In theory, binary searches perform much faster than a linear search. However in order for a binary search to take place, the list must first be sorted. The time it takes to sort a list, especially for larger lists, can outweigh the performance benefit of doing a binary search.