Lists

The simplest data structure in Python. A list is simply what it sounds like: a set of values in a particular order. Each value in a list is identified by an index. The values that make up a list are called its *elements*. Lists are similar to strings, which are ordered lists of characters, except that the elements of a list can be any type.

To create a list, you can use the bracket syntax:

```
list_name = [item1, item2, item3, itemN]
```

Let's make a list of numbers and a list of sequences:

In [ ]:
```
numbers = [1,2,3]
sequences = ['ATG', 'TGA', 'CCC']
empty_list = []
```

The last list we created is an empty list and is analogous to an empty string (`''`).

The `len()` function can be used on lists and returns the number of items in the list:

In [1]:
```
empty_list = []
len(empty_list)
```

Out[1]: 0

In [2]:
```
sequences = ['ATG', 'TGA', 'CCC']
len(sequences)
```

Out[2]: 3

The elements of a list don't have to be the same type:

In [ ]:
```
mixed_list = [1, 'ATG', 1.3, [1,2]]
```

The last element of this list is another list!

List indexing

Elements of a list are accessed using the bracket operator, the same way as the characters of a string. Just as with strings, the indixing of elements starts at 0, and the largest index is the length of the list - 1.

In [3]:
```
numbers = [1, 2, 3]
numbers[1]
```

Out[3]: 2

Any integer expression can be used as an index:

In [4]:
```
numbers = [1, 2, 3]
numbers[9-8]
```

Out[4]: 2

If an index has a **negative** value, it counts backward from the end of the list:

In [5]:
```
numbers = [1, 2, 3]
numbers[-1]
```

Out[5]: 3

Lists are mutable

Unlike strings, lists are mutable, meaning they can be changed. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be changed:

In [6]:
```
numbers = [1, 2, 3]
numbers[0] = 4
numbers
```

Out[6]: [4, 2, 3]

Traversing a list

We can iterate through the elements of a list using a `for` loop, exactly the same way we iterated through the characters of a string:

In [8]:
```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
print(numbers)
```

```
1
2
3
[1, 2, 3]
```

However, if you want to change an element as you go along, you also need access to its index. Therefore we often use the following form of iteration through a list:

In [10]:
```
numbers = [1, 2, 3]
for i in range(len(numbers)):
    numbers[i] = numbers[i] - 1
numbers
```

Out[10]:
```
[0, 1, 2]
```

The idiom `range(len(words))` produces all the possible indices that can be used to access elements of the list.

Recall that `range(n)` generates a sequence of integers starting from 0 until n-1, where n is the number given as its argument:

In [12]:
```
i = list(range(10))
i
```

Out[12]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also specify a different start and step:

In [13]:
```
i = list(range(1,10,2))
i
```

Out[13]:
```
[1, 3, 5, 7, 9]
```

To determine if a value is in a list, we can use the `in` operator:

In [14]:
```
sequences = ['ATG', 'TGA', 'CCC']
'ATG' in sequences
```

Out[14]:
```
True
```

The `max()` function returns the maximum item in a list:

In [15]:
```
numbers = [1, 2, 4, 3]
max(numbers)
```

Out[15]:
```
4
```

`max()` works with `str` values in lists as well. Can you guess which item in this list `max()` will return?

In [16]:
```
sequences = ['ATG', 'CCC', 'TGA']
max(sequences)
```

Out[16]:
```
'TGA'
```

Of course, if there's a `max()` function, there's a `min()` function:

In [17]:
```
sequences = ['ATG', 'CCC', 'TGA']
min(sequences)
```

Out[17]:
```
'ATG'
```

Exercise 8a

Write a function, `list_square(my_list)`, that prints the square of each value in `my_list`.

In [19]:
```
def list_square(my_list):
    for n in my_list:
        print(n**2)

t = [1, 2, 3]
list_square(t)
```

```
1
4
9
```

Debugging

When you apply your code to large datasets it can become unwieldy to debug by printing and checking the results by hand (that's what we are writing the program for in the first place!) Here are some suggestions for debugging programs applied to large datasets:

Create small datasets that you can easily verify

If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest meaningful example you can find. This is best done by editing the files or extracting parts of the file (the Linux `head` function is useful for that).

Print diagnostic information

Consider printing summaries of the data, e.g. the number of items in a list (is it the size that you expect?)

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value using the `type` function.

Sanity checks

Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "completely illogical".

Slices

List slices work exactly as they do for strings:

In [20]:
```
numbers = [1, 2, 3, 4]
numbers[1:3]
```

Out[20]:
```
[2, 3]
```

Lists as references to objects in memory

In Python, variables are references to objects in memory. This is exemplified well with lists:

In [24]:
```
#a = [1, 2, 3]
#b = a
#a == b
#a is b
b = a[:]
a == b
a is b
```

Out[24]:
```
False
```

In [29]:
```
a = [1,2,3]
b = [1,3,2]
a == b
#a is b
```

Out[29]:
```
False
```

The equality operator for lists determines if **all** the elements of the two lists are the same, and in the same order. The `is` operator determines if two varaiables refer to the same object in memory. Since variables are just references to objects, if we assign one variable to another, both variables refer to the same object, and if we change one, that also changes the other:

In [31]:
```
a = [1, 2, 3]
b = a
b[0] = 5
a
```

Out[31]:
```
[5, 2, 3]
```

If you want to modify a list and also keep a copy of the original, make a copy of the list, rather than a new reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy. The easiest way to clone a list is to use the slice operator:

In [ ]:
```
a = [1, 2, 3]
b = a[:]
b[0] = 5
a
```

Now you can make changes to `b` without worrying about affecting `a`.

The `enumerate()` function adds a counter to an iterable, such as a value in a list:

In [34]:
```
my_list = ['ATG', 'TAG', 'AAA']
for i, v in enumerate(my_list):
    print(i, v)
```

```
0 ATG
1 TAG
2 AAA
```

This is useful if we want to iterate through a list keeping track of the position of each value:

Let's write a function, `double_values()`, that doubles the values of items in a list, without worrying about changing the list:

In [39]:
```
def double_values(some_list):
    for index, value in enumerate(some_list):
        #print(index, value)
        some_list[index] = 2 * value
    return some_list

mixed_list = [1, 2, 'ATG', 2.3]
double_values(mixed_list)
```

Out[39]:
```
[2, 4, 'ATGATG', 4.6]
```

We could do the same thing manually by manually creating a counter variable:

In [44]:
```
def double_values(some_list):
    index = 0
    for value in some_list:
        some_list[index] = 2 * value
        index += 1
    return some_list

mixed_list = [1, 2, 'ATG', 2.3]
double_values(mixed_list)
print(mixed_list)
```

```
[2, 4, 'ATGATG', 4.6]
```

Notice that if a function modifies a list sent to it as a parameter, the list is permenantly changed.

Let's write the same function without the modifying its parameter (i.e. the list passed to it):

In [49]:
```
def double_values(some_list):
    index = 0
    new_list = some_list[:]
    for value in some_list:
        new_list[index] = 2 * value
        index += 1
    return new_list

mixed_list = [1, 2, 'ATG', 2.3]
print(double_values(mixed_list))
#print(mixed_list)
```

```
1
2
ATG
2.3
[2, 4, 'ATGATG', 4.6]
```

Which approach is better? The second is less error prone, but ultimately, the choice is context dependent.

List methods

Like strings, lists have methods that allow you to query and change a list.

To append an item to a list, we can use the `list.append(element)` method:

In [52]:
```
n = [1, 2]
b = [3, 4]
n.append(b)
n
```

Out[52]:
```
[1, 2, [3, 4]]
```

Let's use append to write a function, `powers_of_2(n)`, that creates a list of powers of 2 for all integer numbers less than `n`:

In [53]:
```
def powers_of_2(n):
    powers = []
    for i in range(n):
        powers.append(2**i)
    return powers

powers_of_2(5)
```

Out[53]:
```
[1, 2, 4, 8, 16]
```

To concatenate two lists, use the `extend(list_of_elements)` method:

In [59]:
```
list1 = ['a', 'b']
list2 = ['c', 'd']
list2.extend(list1)
```

You can also concatenate lists using the `+` operator:

In [63]:
```
list1 = ['a', 'b']
list2 = ['c', 'd']
list3 = list1 + list2
list3
```

Out[63]:
```
['a', 'b', 'c', 'd']
```

Lists have other useful methods:

- `list.count(value)` : count the number of times a value occurs in a list
- `list.index(value)` : the index of the first occurrence of a value in a list
- `list.insert(index, object)` : insert an object at a specific position
- `list.pop(obj = list[-1])` : remove a value at a given position, by default the last
- `list.remove(object)` : for removing a given object, regardless of its position
- `list.sort()` : sort a list numerically/alphabetically

Exercise 8b

Using the list below, briefly experiment with each of the above methods:

In [81]:
```
my_list = [1, 3, 7, 4, 5, 1]
```

In [82]:
```
my_new = my_list[:]
my_new.sort()
my_list
```

Out[82]:
```
[1, 3, 7, 4, 5, 1]
```

Lists and strings

Strings can be thought of as a list of characters, and you can easily convert a string into a list:

In [ ]:
```

```

If you want to convert the string into a list of words, use a string's `split` method (`str.split(str=' ', num=string.count(str))`):

In [ ]:

Comma delimited files (.csv) are a common tabular data format. A line in a comma delimited file may look something like:

In [ ]:

To convert this into a list, we can use the `split()` method again:

In [ ]:

To strip off the new line we can do this:

In [ ]:

Notice that the values are strings. To convert to numbers we could iterate through the list one value at a time:

In [ ]:

Exercise 8c

Write a function, `string_to_float(my_list)` , that converts a list of strings, like the one below, into a list of numbers.

In [ ]:

We can also go in the opposite direction - from a list to a string using the `join` method:

In [ ]:
```
my_list = ['ATG', 'CCC', 'ATG']
''.join(my_list) # join by empty string
','.join(my_list) # join by comma
my_list # the actual list is unchanged
```

Nested lists and matrices

As we have seen above, a nested list is a list that appears as an element in another list. To extract an element from the nested list, we can proceed in two steps:

In [ ]:

Or in one step:

In [ ]:

Nested lists are a way of representing a matrix. For example:

In [ ]:
```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

The variable `matrix` is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix or extract a single element from the matrix using the double-index form where the first index selects the row, and the second index selects the column:

In [ ]:
```
'''
matrix
1, 2, 3
4, 5, 6
7, 8, 9
'''
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Exercie 8d

Excract the element in row 3, column 2 of the matrix below.

In [ ]:
```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [ ]:

Here's code for creating a matrix of zeros with user-defined numbers of rows and columns:

In [ ]:
```
def zeros_matrix(rows, columns):
    """create a matrix with a given number of rows and columns
    """
    matrix = []
    for row in range(rows):
        matrix.append([0] * columns)
    return matrix

zeros_matrix(3,2)
```

Additional Exercises

Answer key (http://rna.colostate.edu/dokuwiki/doku.php?id=script:ex8)

8e) Make a list, my_list, containing three values. Check that the object you created is a list using the `type()` function.

In [ ]:

8f) Make the following three lists into a single matrix:

In [ ]:
```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]
```

8g) Using list indices, extract the value in row 2, column 2 from the matrix in 8f.

In [ ]:

8h) Using slicing, extract the first two rows from the matrix in 8f.

In [ ]:

8i) Using an index, extract the last row of the matrix in 8f, assuming you don't know how many rows are in the matrix:

In [ ]:

8j) Write a function, `element_replace(matrix, n, i, c)`, that replaces the element in the nth row and ith column with value c and returns a new matrix with the revision.

In [ ]:

8k) Write a function called `sum_rows(matrix)` that sums the elements in each row in a matrix and returns the values as a list. Hint: use a nested for loop. The outer for loop will iterate through each row and the nested for loop will iterate through each element in each row.

In [ ]:

8l) Write a function called `sum_columns(matrix)` that sums the elements in each column in a matrix and returns the values as a list.

In [ ]: