Lecture 3: Strings and lists

Other built-in types

As we saw in the previous lecture, Python includes some basic data types to store different types of numbers: bool, int, float, and complex class. These classes are defined in the source interpreter. Python also includes a number of other useful built-in types for storing data. We will focus on two more: strings and lists.

Containers and sequences

The strings and list classes are a little bit different than the classes you've seen before in that they are designed to hold multiple values. For this reason, they are called containers. There are other container classes, such as tuples, dictionaries, and sets that are also important. Strings and lists are a subclass of containers known as sequences. Sequences not only are containers for multiple elements, but these elements also have an order. Strings typically contain words and phrases. You can't mix up the letters and expect the word to remain the same. The same is true with lists.

Strings

Strings contain 0 or more text characters. Like almost everything in Python, strings are also objects, encoded by the 'str' class. This data type is very useful for encoding many things biologists care about – DNA/protein sequence, species names, or tissue types to name a few. Some examples of strings:

'ATAACCGTAGACCGT', 'Caenorhabditis elegans', 'Homo sapiens', 'Heart'

Just like with numerics, there is an implicit way to create a string object. The Python interpreter will take anything between two single quotes, two double quotes, or two triple quotes and create a string instance. For example:

```
>>> empty = ''
>>> type(empty)
<class 'str'>
>>> print(empty)

>>> name = 'Patrick McGrath'
>>> print(name)
Patrick McGrath
>>> name = "Patrick McGrath"
>>> print(name)
Patrick McGrath
>>> """Patrick McGrath"""
>>> print(name)
Patrick McGrath
```

Note that you can't mix and match the quotes to create a string.

```
>>> name = "Patrick McGrath'
  File "<stdin>", line 1
    name = "Patrick McGrath'
                           ^
SyntaxError: EOL while scanning string literal
```

Sometimes you might want your string to contain an apostrophe or quotes within it. But this gets into an issue as python uses the single quote to define the string.

```
>>> phrase = 'She'll be coming around the mountain when she comes'
  File "<stdin>", line 1
    phrase = 'She'll be coming around the mountain when she comes'
                   ^
SyntaxError: invalid syntax
```

There are a couple ways around it: If you want to use the apostrophe in your string, use the double quotes to define string (or vice-versa)

```
>>> phrase = "She'll be coming around the mountain when she comes"
>>> phrase = 'If you want to read "Watership Down", I would recommend it'
```

If you want to use both, you can always use the triple quotes to define the string:

```
>>> phrase = """It's fun to read "Watership Down"."""
>>> print(phrase)
It's fun to read "Watership Down".
```

However, you can still have issues if you want the string to end with a quote:

```
>>> phrase = """It's fun to read "Watership Down""""
  File "<stdin>", line 1
    phrase = """It's fun to read "Watership Down""""
                                                    ^
SyntaxError: EOL while scanning string literal
```

One final way around this is to use the '\' as an escape character. The escape character tells the Python interpreter to treat the next character as text, as opposed to a special character (such as how Python uses the quotes to define the start and stop of the string).

```
>>> phrase = """It's fun to read "Watership Down\""""
>>> print(phrase)
It's fun to read "Watership Down"
>>> phrase = "If you want to read \"Watership Down\", I would recommend
it"
>>> print(phrase)
If you want to read "Watership Down", I would recommend it
```

Input

A useful built-in function is called input(), which takes in a input from the user and stores it as a string:

```
>>> x = input("Enter a number ")
Enter a number 16
>>> x
'16'
>>> type(x)
<class 'str'>
```

Note that this is a great way to get info from a user. However, no matter what gets entered, the data is stored as a string (as you can see, even though the user stored a number, x is of the class 'str').

This can create issues if you want to do any sort of math on x as you can't combine a string object with a number object:

```
>>> x + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The type error results because you can't add an int to a string. The solution is to convert x into an integer by explicitly creating an int:

```
>>> int_x = int(x)
>>> type(int_x)
<class 'int'>
>>> int_x
16
>>> int_x + 3
19
```

You can also create a string from an integer using the same explicit route:

```
>>> str_x = str(int_x)
>>> type(str_x)
<class 'str'>
>>> str_x
'16'
```

As always, when encountering a new class or object, try using the dir() built-in function to understand what it can do.

```
>>> dir(phrase)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Looks like there are lots of useful methods/attributes. Let's try a few out:

```
>>> phrase
'If you want to read "Watership Down", I would recommend it'
>>> phrase.title()
'If You Want To Read "Watership Down", I Would Recommend It'
>>> phrase.upper()
'IF YOU WANT TO READ "WATERSHIP DOWN", I WOULD RECOMMEND IT'
>>> phrase.lower()
'if you want to read "watership down", i would recommend it'
>>> phrase
'If you want to read "Watership Down", I would recommend it'
```

Pretty straightforward. These three functions manipulate the capitalization of the letters in a string and return it as a new string. Note that these functions DO NOT change the value of the underlying string, rather they return a brand new string that you can store within a variable.

If we look over the magic methods, there are a few that look familiar. The string class has an __add__() method, so apparently we can add strings together:

```
>>> "Patrick" + "McGrath"
'PatrickMcGrath'
```

```
>>> "Patrick".__add__("McGrath")
'PatrickMcGrath'
```

The plus sign acts to concatenate two strings together using the add magic method. String also has a __mul__ method. This is a bit unintuitive:

```
>>> "Patrick" * 6
'PatrickPatrickPatrickPatrickPatrickPatrick'
>>> "Patrick".__mul__(6)
'PatrickPatrickPatrickPatrickPatrickPatrick'
>>> "Patrick" * "McGrath"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

The string multiply method only works with integers and acts to repeat the sta

Accessing individual characters of a string

As mentioned above, string is a sequence of letters in a defined order. You can access the individual characters of a string using the [] operator. The [] takes a parameter that details the letters that you want. This is called indexing a string. For example:

```
>>> phrase[0]
'I'
>>> phrase[1]
'f'
>>> phrase[2]
' '
```

Note that Python is 0-indexed so that 0 returns the first letter of the string.

Python also allows convenient ways to return the last letter of the string using a negative index:

```
>>> phrase[57]
't'
>>> phrase[-1]
't'
```

The len built-in function is useful to find the length of a string using the __len__ magic method:

```
>>> len(phrase)
58
```

To return the last element of the string, we have to subtract one since the first index is 0:

```
>>> phrase[len(phrase)-1]
't'
>>> phrase[len(phrase)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Strings are immutable.

One peculiarity regarding strings is that you cannot modify them once they are created. For example:

```
>>> phrase[6]='a'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> phrase[1]='b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

You can replace a string completely, however, it creates an entire new string someplace else in memory:

```
>>> phrase += '. Its a fun read'
>>> phrase
'If you want to read "Watership Down", I would recommend it. Its a fun read'
>>> phrase = 'Brand new string'
>>> phrase
'Brand new string'
>>> phrase = 'If you want to read "Watership Down", I would recommend it'
>>> phrase
'If you want to read "Watership Down", I would recommend it'
```

You might wonder why this is important. For example, if you were storing a genome (for example, the ~4 million bp of *E. coli*), it might seem natural to store it as a string (genome). However, you couldn't change one of the bases (genome[5453] = 'A'), and if you wanted to append an extra base at the end (i.e. genome += 'C'), you would have to recopy the entire 4 million bp string. This would take a lot of time.

Slicing parts of a string

Besides indexing a string, you can also slice a string to return multiple characters. Slicing requires the use of a colon (:) like so:

```
>>> phrase[0:6]
'If you'
>>> phrase[23:42]
'tership Down", I wo'
>>> phrase[18:-2]
'd "Watership Down", I would recommend '
```

As should be obvious, this returns multiple characters of the string. Note that phrase[0:6] returns the first 5 characters.

Slicing also can use default values, so omitting the first number will default to zero and omitting the last will default to the end of the string. As so:

```
>>> phrase[:6]
'If you'
>>> phrase[18:]
'd "Watership Down", I would recommend it'
>>> phrase[:]
'If you want to read "Watership Down", I would recommend it'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
```

```
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Finally, besides passing the start and stop indexes to the [] operator, you can also add a third parameter to return a subset of elements (i.e. 2 returns every other element), or switch the order of the elements (-1, returns them backwards):

```
>>> phrase[::2]
'I o att ed"aesi on,Iwudrcmedi'
>>> phrase[::-1]
'ti dnemmocer dluow I ,"nwoD pihsretaW" daer ot tnaw uoy fI'
```

Lists

Lists are similar to strings, however, they are much more versatile. Lists are mutable and can store any sort of data. The implicit method of creating a list is using the square brackets with commas in between each like so:

```
>>> numbers = []
>>> type(numbers)
<type 'list'>
>>> dir(numbers)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> numbers = [1,4,9,16,25,36,49]
>>> numbers[0]
1
>>> numbers[6]
49
```

You can also change the value of an individual element like so:

```
>>> numbers[3] = 15
>>> numbers
[1, 4, 9, 16, 25, 36, 49]
>>> numbers[9] = 15
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Note what happens when you try to assign an element that doesn't exist. If you want to extend a list, you either need to use the + operator:

```
>>> numbers += [64,81,100]
>>> numbers
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

There are also two built in methods, append and extend, that can be used to add to the list. Append takes in a single element to add to the list. Extend takes in a list and adds it to the list:

```
>>> numbers.append(121)
>>> numbers
[1, 4, 16, 25, 36, 49, 64, 81, 100, 121]
>>> numbers.extend([144,169,196])
>>> numbers
[1, 4, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

reverse and sort are two list methods that modify the list as you probably would expect:

```
>>> numbers
[1, 4, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
>>> numbers.reverse()
>>> numbers
[196, 169, 144, 121, 100, 81, 64, 49, 36, 25, 16, 4, 1]
>>> numbers.sort()
>>> numbers
[1, 4, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

You can also remove elements of a list using a variety of methods. For example, you can utilize the pop method to remove a specific element (pop also returns that element in case you want to do something with it:

```
>>> numbers.pop(3)
25
>>> numbers
[1, 4, 16, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

Note the 25 is now gone from the numbers list. The remove method takes in a single argument and removes any elements that match the value of the argument.

```
>>> numbers.remove(64)
>>> numbers
[1, 4, 16, 36, 49, 81, 100, 121, 144, 169, 196]
```

You can also clear the entire list using the clear method.

```
>>> numbers.clear()
>>> numbers
[]
```

You can also use the del statement to remove elements:

```
>>> del numbers[8]
>>> numbers
[1, 4, 16, 25, 36, 49, 64, 81, 121, 144, 169, 196]
```

del is nice to delete multiple elements at the same time. You can use the slice notation, just like with a string:

```
>>> del numbers[1:3]
>>> numbers
[1, 25, 36, 49, 64, 81, 121, 144, 169, 196]
```

Shallow copies

One funny thing about lists occurs when you try to make a copy of a list:

```
>>> numbers
```

```
[1, 25, 36, 49, 64, 81, 121, 144, 169, 196]
>>> numbers_copy = numbers
>>> numbers_copy
[1, 25, 36, 49, 64, 81, 121, 144, 169, 196]
```

Ok, so far, nothing strange seems to have happened. But now let's try to modify the new list:

```
>>> numbers_copy[0] = 252
>>> numbers_copy
[252, 25, 36, 49, 64, 81, 121, 144, 169, 196]
>>> numbers
[252, 25, 36, 49, 64, 81, 121, 144, 169, 196]
```

Apparently, modifying the new list also modifies the original list. This is what is referred to as a shallow copy. If you want to create a brand new list (called a deep copy), you need to use the copy method:

```
>>> numbers_newest = numbers.copy()
>>> numbers_newest
[252, 25, 36, 49, 64, 81, 121, 144, 169, 196]
>>> numbers_newest[0] = 1
>>> numbers_newest
[1, 25, 36, 49, 64, 81, 121, 144, 169, 196]
>>> numbers
[252, 25, 36, 49, 64, 81, 121, 144, 169, 196]
```

Functions that work on lists

There are also a number of built-in functions (i.e. not methods of the list class) that take a list as an input.

sort: This function takes in a list as an argument and returns a sorted list, leaving the original list unchanged.

```
>>> numbers = [44,33,11,535,2,4324,2342]
>>> sorted(numbers)
[2, 11, 33, 44, 535, 2342, 4324]
>>> letters = ['a','g','s','c','z','f','x']
>>> sorted(letters)
['a', 'c', 'f', 'g', 's', 'x', 'z']
>>> numbers
[44, 33, 11, 535, 2, 4324, 2342]
>>> letters
['a', 'g', 's', 'c', 'z', 'f', 'x']
```

sum: This function adds up all of the elements of a list, as long as they are numbers

```
>>> sum(numbers)
7291
>>> sum(letters)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

List comprehensions

One of the most useful features of lists is the ability to modify each element of the list in any way you want. The basic syntax is as follows:

[<Modify element code> for <give a name to each element> in <specify the list that will be modified>]

As an example, here are two ways you can take the square of each element or convert an element to a different datatype.

```
>>> numbers = [1,2,3,4,5]
>>> numbers
[1, 2, 3, 4, 5]
>>> [x*x for x in numbers]
[1, 4, 9, 16, 25]
>>> [float(x) for x in numbers]
[1.0, 2.0, 3.0, 4.0, 5.0]
```

Useful string methods

There are a number of useful string methods you will need to use.

We've already seen what upper() does. It comes in useful as DNA sequence is alternatively written as both uppercase and lowercase. If we are comparing two sequences using Python's comparison operator (==):

```
>>> 'accct' == 'ACCCT'
False
```

In most cases we would consider those two sequences as the same. This can be fixed by using the upper method:

```
>>> 'accct'.upper() == 'ACCCT'
True
```

The find() method searches the string for a smaller string and returns the first index where it is found. If the string isn't found, it returns a -1

```
>>> phrase = 'Watership Down'
>>> phrase.find('Watership')
0
>>> phrase.find('Down')
10
>>> phrase.find(' ')
9
>>> phrase.find('Up')
-1
>>> phrase.find('WATERSHIP')
-1
```

The count() method takes in a string and counts how many times it is found in the parent string:

```
>>> phrase = 'Georgia Institute of Technology'
>>> phrase.count('t')
3
>>> phrase.count('T')
1
```

The join() method takes in a list of strings and joins them together with the original string as a separator:

```
>>> letters
['v', 'w', 'x', 'y', 'z']
>>> ' '.join(letters)
'v w x y z'
>>> ','.join(letters)
'v,w,x,y,z'
```

The split() method takes in a string and splits the parent string into a list

```
>>> phrase
'Watership Down'
>>> phrase.split()
['Watership', 'Down']
>>> phrase.split(' ')
['Watership', 'Down']
>>> phrase.split('Down')
['Watership ', '']
>>> phrase.split('Watership')
['', ' Down']
>>> phrase.split('ship')
['Water', ' Down']
```

The startswith()/endwith() methods takes in a string determines if the object starts or ends with that phrase.

```
>>> filename = 'firstscript.py'
>>> filename
'firstscript.py'
>>> filename.endswith('.py')
True
>>> filename.endswith('.docx')
False
>>> filename.endswith('firstscript')
False
```

The strip() method allows you to remove characters from the end of a string. If the characters are not present, nothing happens. If no argument is specified, whitespace is removed (whitespace is any spaces, tabs, or new line characters)

```
>>> filename.strip('.py')
'firstscript'
>>> filename.strip('.doc')
'firstscript.py'
>>> temp = '23423 '
>>> temp
'23423 '
>>> temp.strip()
'23423'
>>> temp = '23423   '
>>> temp
'23423   '
>>> temp.strip()
'23423'
```

The replace() method allows you to replace a search string from the given string with a replacement string. If more than one occurrence of the search string is found, all are replaced:

```
>>> filename.replace('.py','.doc')
'firstscript.doc'
>>> filename.replace('firstscript', 'secondscript')
'secondscript.py'
>>> filename.replace('p','...')
'firstscri...t....y'
```

Chaining methods together

One of the more interesting things that you can do in python is chain methods together (there is no practical limit to how many times you can do this)

```
>>> phrase
'Georgia Institute of Technology'
>>> phrase.count('T')
1
>>> phrase.upper()
'GEORGIA INSTITUTE OF TECHNOLOGY'
>>> phrase
'Georgia Institute of Technology'
>>> phrase.count('T')
1
>>> phrase.count('t')
3
>>> phrase.upper()
'GEORGIA INSTITUTE OF TECHNOLOGY'
>>> phrase.upper().count('T')
4
```