Python 3

Sequences

In the next section, we will learn about strings, tuples and lists. These are all examples of python sequences. A string is a sequence of characters `'ACGTGA'`, a tuple `(0.23, 9.74, -8.17, 3.24, 0.16)` and a list `['dog', 'cat', 'bird']` are sequences of any kind of data. We'll see much more detail in a bit.

In python a type of object gets operations that belong to that type. Sequences have sequence operations so strings can also use sequence operations. Strings also have their own specific operations.

You can ask what the length of any sequence is

```
1  >>>len('ACGTGA') # length of a string
2  6
3  >>>len( (0.23, 9.74, -8.17, 3.24, 0.16) )   # length of a tuple, needs two
   parentheses (( ))
4  5
5  >>>len(['dog', 'cat', 'bird'])  # length of a list
6  3
```

You can also use string-specific functions on strings, but not on lists and vice versa. We'll learn a lot more about this later on. `rstrip()` is a string method or function. You get an error if you try to use it on a list.

```
1  >>> 'ACGTGA'.rstrip('A')
2  'ACGTG'
3  >>> ['dog', 'cat', 'bird'].rstrip()
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  AttributeError: 'list' object has no attribute 'rstrip'
```

What functions go with my object?

How do you find out what functions work with an object? There's a handy function `dir()`. As an example what functions can you call on our string `'ACGTGA'`?

```
1  >>> dir('ACGTGA')
2  ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
   '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
   '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
   '__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']
```

`dir()` will return all atributes of an object, among them its functions. Technically, functions belonging to a specific object are called methods. You can call `dir()` on any object, most often, you'll use it in the interactive python shell.

Strings

- A string is a series of characters starting and ending with single or double quotation marks.
- Strings are an example of a python sequence. A sequence is defined as a positionally ordered set. This means each element in the set has a position, starting with zero, i.e. 0,1,2,3 and so on until you get to the end of the string.

Quotation Marks

- Single (')
- Double (")
- Triple (''' or """)

Notes about quotes:

- Single and double quotes are equivalent.
- A variable name inside quotes is just the string identifier, not the value stored inside the variable.
- Triple quotes are used before and after a string that spans multiple lines.

Use of quotation examples:

```
1  word = 'word'
2  sentence = "This is a sentence."
3  paragraph = """This is a paragraph. It is
4  made up of multiple lines and sentences. And goes
5  on and on.
6  """
```

Strings and the `print()` function

We saw examples of `print()` earlier. Lets talk about it a bit more. `print()` is a function that takes one or more comma-separated arguments.

Let's use the `print()` function to print a string.

```
1  >>>print("ATG")
2  ATG
```

Let's assign a string to a variable and print the variable.

```
1  >>>dna = 'ATG'
2  ATG
3  >>> print(dna)
4  ATG
```

What happens if we put the variable in quotes?

```
1  >>>dna = 'ATG'
2  ATG
3  >>> print("dna")
4  dna
```

> The literal string 'dna' is printed to the screen, not the contents 'ATG'

Let's see what happens when we give `print()` two literal strings as arguments.

```
1  >>> print("ATG","GGTCTAC")
2  ATG GGTCTAC
```

> We get the two literal strings printed to the screen separated by a space

What if you do not want your strings separated by a space? Use the concatenation operator to concatenate the two strings before or within the `print()` function.

```
1  >>> print("ATG"+"GGTCTAC")
2  ATGGGTCTAC
3  >>> combined_string = "ATG"+"GGTCTAC"
4  ATGGGTCTAC
5  >>> print(combined_string)
6  ATGGGTCTAC
```

> We get the two strings printed to the screen without being separated by a space.
> You can also use this

```
1  >>> print('ATG','GGTCTAC',sep='')
2  ATGGGTCTAC
```

Now, lets print a variable and a literal string.

```
1  >>>dna = 'ATG'
2  ATG
3  >>> print(dna,'GGTCTAC')
4  ATG GGTCTAC
```

> We get the value of the variable and the literal string printed to the screen separated by a
> space

How would we print the two without a space?

```
1  >>>dna = 'ATG'
2  ATG
3  >>> print(dna + 'GGTCTAC')
4  ATGGGTCTAC
```

Something to think about: Values of variables are variable. Or in other words, they are mutable,
changeable.

```
1  >>>dna = 'ATG'
2  ATG
3  >>> print(dna)
4  ATG
5  >>>dna = 'TTT'
6  TTT
7  >>> print(dna)
8  TTT
```

Errors and Printing

Let's look at the typical errors you will encouter when you use the `print()` function.

What will happen if you forget to close your quotes?

```
1  >>> print("GGTCTAC)
2    File "<stdin>", line 1
3      print("GGTCTAC)
4                    ^
5  SyntaxError: EOL while scanning string literal
```

What will happen if you forget to enclose a string you want to print in quotes?

```
1  >>> print(GGTCTAC)
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  NameError: name 'GGTCTAC' is not defined
5  >>> GGTCTAC = 5  # define a variable
6  >>> print(GGTCTAC)
7  5
```

```
1  >>> print "boo"
2    File "<stdin>", line 1
3      print "boo"
4              ^
5  SyntaxError: Missing parentheses in call to 'print'
```

In python2, the command was `print`, but this changed to `print()` in python3, so don't forget the parentheses!

Special/Escape Characters

How would you include a new line, carriage return, or tab in your string?

Escape Character	Description
\n	New line
\r	Carrage Return
\t	Tab

Let's include some escape characters in our strings and `print()` functions.

```
1  >>> string_with_newline = 'this sting has a new line\nthis is the second line'
2  >>> print(string_with_newline)
3  this sting has a new line
4  this is the second line
```

> We printed a new line to the screen

`print()` adds spaces between arguments and a new line at the end for you. You can change these with `sep=` and `end=`. Here's an example: `print('one line', 'second line' , 'third line', sep='\n', end = '')`

A neater way to do this is to express a multi-line string enclosed in triple quotes (""").

```
1  >>> print("""this string has a new line
2  ... this is the second line""")
3  this string has a new line
4  this is the second line
```

Let's print a tab character (\t).

```
1  >>> line = "value1\tvalue2\tvalue3"
2  >>> print(line)
3  value1  value2  value3
```

> We get the three words separated by tab characters. A common format for data is to separate columns with tabs like this.

You can add a backslash before any chacter to force it to be printed as a literal. This is called 'escaping'. This is only really useful for printing literal quotes ' and "

```
1  >>> print('this is a \'word\'')  # if you want to print a ' inside '...'
2  this is a 'word'
3  >>> print("this is a 'word'") # maybe clearer to print a ' inside "..."
4  this is a 'word'
```

> In both cases actual single quote character are printed to the screen

If you want every character in your string to remain exactly as it is, declare your string a raw string literal with 'r' before the first quote. This looks ugly, but it works.

```
1  >>> line = r"value1\tvalue2\tvalue3"
2  >>> print(line)
3  value1\tvalue2\tvalue3
```

> Our escape characters '\t' remain as we typed them, they are not converted to actual tab characters.

Concatenation

To concatenate strings use the concatenation operator '+'

```
1  >>> promoter= 'TATAAA'
2  >>> upstream = 'TAGCTA'
3  >>> downstream = 'ATCATAAT'
4  >>> dna = upstream + promoter + downstream
5  >>> print(dna)
6  TAGCTATATAAAATCATAAT
```

> The concatenation operator can be used to combine strings. The newly combined string can be stored in a variable.

The difference between string + and integer +

What happens if you use `+` with numbers (these are integers or ints)?

```
1  >>> 4+3
2  7
```

For strings, `+` concatenates; for integers, `+` adds.

You need to convert the numbers to strings before you can concatenate them

```
1  >>> str(4) + str(3)
2  '43'
```

Determine the length of a string

Use the `len()` function to calculate the length of a string. This function takes a sequence as an argument and returns an int

```
1  >>> print(dna)
2  TAGCTATATAAAATCATAAT
3  >>> len(dna)
4  20
```

> The length of the string, including spaces, is calculated and returned.

The value that `len()` returns can be stored in a variable.

```
1  >>> dna_length = len(dna)
2  >>> print(dna_length)
3  20
```

You can mix strings and ints in `print()`, but not in concatenation.

```
1  >>> print("The lenth of the DNA sequence:" , dna , "is" , dna_length)
2  The lenth of the DNA sequence: TAGCTATATAAAATCATAAT is 20
```

Changing String Case

Changing the case of a string is a bit different than you might first expect. For example, to lowercase a string we need to use a method. A method is a function that is specific to an object. When we assign a string to a variable we are creating an instance of a string object. This object has a series of methods that will work on the data that is stored in the object. Recall that `dir()` will tell you all the methods that are available for an object. The `lower()` function is a string method.

Let's create a new string object.

```
1  dna = "ATGCTTG"
```

> Look familiar? It should!!! Creating a string object is what we have been doing all along!!! Jeez!!!

Now that we have a string object we can use string methods. The way you use a method is to put a '.' between the object and the method name.

```
1  >>> dna = "ATGCTTG"
2  >>> dna.lower()
3  'atgcttg'
```

> the lower() method returns the contents stored in the 'dna' variable in lowercase.

The contents of the 'dna' variable have not been changed. Strings are immutable. If you want to keep the lowercased version of the string, store it in a new variable.

```
1  >>> print(dna)
2  ATGCTTG
3  >>> dna_lowercase = dna.lower()
4  >>> print(dna)
5  ATGCTTG
6  >>> print(dna_lowercase)
7  atgcttg
```

The string method can be nested inside of other functions.

```
1  >>> dna = "ATGCTTG"
2  >>> print(dna.lower())
3  atgcttg
```

> The contents of 'dna' are lowercased and passed to the `print()` function.

If you try to use a string method on a object that is not a string you will get an error.

```
1  >>> nt_count = 6
2  >>> dna_lc = nt_count.lower()
3  Traceback (most recent call last):
4    File "<stdin>", line 1, in <module>
5  AttributeError: 'int' object has no attribute 'lower'
```

> You get an AttributeError when you use a method on the an incorrect object type. We are told that the int object (an int is returned by `len()`) does not have a function called lower.

Now let's uppercase a string.

```
1  >>> dna = 'attgct'
2  >>> dna.upper()
3  'ATTGCT'
4  >>> print(dna)
5  attgct
```

> The contents of the variable 'dna' were returned in upper case. The contents of 'dna' were not altered.

Find and Count

`count(str)` returns the number of exact matches of `str` it found (as an int)

```
1    >>> dna = 'ATGCTGCATT'
2    >>> dna.count('T')
3    4
```

> The number of times 'T' is found is returned. The string stored in 'dna' is not altered.

Find and Replace

`replace(str1,str2)` returns a new string with all matches of `str1` in a string replaced with `str2`.

```
1    >>> dna = 'ATGCTGCATT'
2    >>> dna.replace('T','U')
3    'AUGCUGCAUU'
4    >>> print(dna)
5    ATGCTGCATT
6    >>> rna = dna.replace('T','U')
7    >>> print(rna)
8    AUGCUGCAUU
```

> All occurrences of T are replaced by U. The new string is returned. The orginal string has not actually been altered. If you want to reuse the new string, store it in a variable.

Extracting a Substring, or Slicing

Parts of a string can be located based on position and returned. This is because a string is a sequence. Coordinates start at 0. You add the coordinate in square brackets after the string's name.

This string 'ATTAAAGGGCCC' is made up of the following sequence of characters, and positions (starting at zero).

Position/Index	Character
0	A
1	T
2	T
3	A
4	A
5	A
6	G
7	G
8	G
9	C
10	C
11	C

Let's return the 4th, 5th, and 6th nucleotides. To do this, we need to count like a computer and start our string at 0 and return the 3rd, 4th, and 5th characters. This will be everything from 3 to 6. Python counts the gaps before each character in the string, starting at 0.

```
1  >>> dna = 'ATTAAAGGGCCC'
2  >>> sub_dna = dna[3:6]
3  >>> print(sub_dna)
4  AAA
```

> The chacters with indices 3, 4, 5 are returned. Or in other words, every charater starting at index 3 and up to but not including, the index of 6 are returned.

Let's return the first 6 characters.

```
1  >>> dna = 'ATTAAAGGGCCC'
2  >>> sub_dna = dna[0:6]
3  >>> print(sub_dna)
4  ATTAAA
```

> Every character starting at index 0 and up to but not including index 6 are returned. This is the same as dna[:6]

Let's return every character from index 6 to the end of the string.

```
1  >>> dna = 'ATTAAAGGGCCC'
2  >>> sub_dna = dna[6:]
3  >>> print(sub_dna)
4  GGGCCC
```

> When the second argument is left blank, every character from index 6 and greater is returned.

Let's return the last 3 characters.

```
1  >>> sub_dna = dna[-3:]
2  >>> print(sub_dna)
3  CCC
```

> When the second argument is left blank and the first argument is negative (-X), X characters from the end of the string are returned.

Locate and Report

The positional index of an exact string in a larger string can be found and returned with the string method `find()`. An exact string is given as an argument and the index of its first occurrence is returned. -1 is returned if it is not found.

```
1  >>> dna = 'ATTAAAGGGCCC'
2  >>> dna.find('T')
3  1
4  >>> dna.find('N')
5  -1
```

> The substring 'T' is found for the first time at index 1 in the string 'dna' so 1 is returned. The substring 'N' is not found, so -1 is returned.

Other String Methods

Since these are methods, be sure to use in this format string.method().

function	Description
`s.strip()`	returns a string with the whitespace removed from the start and end
`s.isalpha()`	tests if all the characters of the string are alphabetic characters. Returns True or False.
`s.isdigit()`	tests if all the characters of the string are numeric characters. Returns True or False.
`s.startswith('other_string')`	tests if the string starts with the string provided as an argument. Returns True or False.
`s.endswith('other_string')`	tests if the string ends with the string provided as an argument. Returns True or False.
`s.split('delim')`	splits the string on the given exact delimiter. Returns a list of substrings. If no argument is supplied, the string will be split on whitespace.
`s.join(list)`	opposite of `split()`. The elements of a list will be concatenated together using the string stored in 's' as a delimiter.

String Formatting

Strings can be formated using the `format()` function. Pretty intuitive, but wait til you see the details! For example, if you want to include literal stings and variables in your print statement and do not want to concatenate or use multiple arguments in the `print()` function you can use string formatting.

```
1  >>> string = "This sequence: {} is {} nucleotides long and is found in {}."
2  >>> string.format(dna,dna_len,gene_name)
3  'This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in
   Brca1.'
4  >>> print(string) # string.format() does not alter string
5  This sequence: {} is {} nucleotides long and is found in {}.
6  >>> new_string = string.format(dna,dna_len,gene_name)
7  >>> print(new_string)
8  This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in
   Brca1.
```

We put together the three variables and literal strings into a single string using the function `format()`. The original string is not altered, a new string is returned that incorporates the arguments. You can save the returned value in a new variable. Each `{}` is a placeholder for the strings that need to be inserted.

Something nice about `format()` is that you can print int and string variable types without converting first.

You can also directly call the format function on a string inside a print function. Here are two examples

```
1  >>> string = "This sequence: {} is {} nucleotides long and is found in {}."
2  >>> print(string.format(dna,dna_len,gene_name))
3  This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in
   Brca1.
```

Or you use the `format()` function on a literal string:

```
1  >>> print( "This sequence: {} is {} nucleotides long and is found in
   {}.".format(dna,dna_len,gene_name))
2  This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in
   Brca1.
```

> There is no need to store the string in a variable.

The `format()` mini-language

So far, we have just used `{}` to show where to insert the value of a variable in a string. You can add special characters inside the `{}` to change the way the variable is formatted when it's inserted into the string.

> You can number these, not necessarily in order.

```
1  >>> '{0}, {1}, {2}'.format('a', 'b', 'c')
2  'a, b, c'
3  >>> '{2}, {1}, {0}'.format('a', 'b', 'c')
4  'c, b, a'
```

To change the spacing of strings and the way numbers are formatted, you add `:` and other special characters like this `{:>5}` to right-justify a string in a five-character field.

Lets right justify some numbers.

```
1   >>> print( "{:>5}".format(2) )
2       2
3   >>> print( "{:>5}".format(20) )
4      20
5   >>> print( "{:>5}".format(200) )
6     200
```

How about padding with zeroes? This means the five-character field will be filled as needed with zeroes to the left of any numbers you want to display

```
1   >>> print( "{:>05}".format(2) )
2   00002
3   >>> print( "{:>05}".format(20) )
4   00020
```

Use a `<` to indicate left-justification.

```
1   >>> print( "{:<5} genes".format(2) )
2   2     genes
3   >>> print( "{:<5} genes".format(20) )
4   20    genes
5   >>> print( "{:<5} genes".format(200) )
6   200   genes
```

Center aligning is done with `^` instead of `>` or `<`. You can also pad with characters other than 0. Here let's try `_` or underscore as in `:_^`.

```
1   >>> print( "{:_^10}".format(2) )
2   ____2_____
3   >>> print( "{:_^10}".format(20) )
4   ____20____
5   >>> print( "{:_^10}".format(200) )
6   ___200____
```

> Text can be centered by using ':^10'. 10 of course is your column width. The '^' indicates center justification. In our example an underscore is used to replace the empty spaces to make things easier to see.

Summary of special formatting symbols so far

Here are some of the ALIGNMENT options:

Option	Meaning	
'<'	Forces the field to be left-aligned within the available space (this is the default for most objects).	
'>'	Forces the field to be right-aligned within the available space (this is the default for numbers).	
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types.	
'^'	Forces the field to be centered within the available space.	

> Here's an example
>
> `{ : x < 10 s}`
>
> fill with `x`
> left justify `<`
> `10` a field of ten characters `s` a string

Common Types

type	description
b	convert to binary
d	decimal integer
e	exponent, default precision is 6, uses `e`
E	exponent, uses `E`
f	floating point, default precision 6 (also F)
g	general number, float for values close to 0, exponent for others; also G
s	string, default type (see example above)
x	convert to hexadecimal, also X
%	converts to % by multiplying by 100

What's the point?

So much can be done with the `format()` function. Here is one last example, but not the last functionality of this function. Let truncate a long floating point number. The default is 6 decimal places. Note that the function rounds to the nearest decimal place, but not always exactly the way you expect because of the way computers represent decimals with 1s and 0s.

```
1  '{:f}'.format(3.141592653589793)
2  '3.141593'
3  >>> '{:.4f}'.format(3.141592653589793)
4  '3.1416'
```

Lists and Tuples

Lists

Lists are data types that store a collection of data.

- Lists are used to store an ordered, *indexed* collection of data.
- Values are separated by commas
- Values are enclosed in square brackets '[]'
- Lists can grown and shrink
- Values are mutatable

Accessing Values in Lists

To retrieve a single value in a list use the value's index in this format list[index]. This will return the value at the specified index, starting with 0.

Here is a list:

```
1  >>> codons = [ 'atg' , 'aaa' , 'agg' ]
```

There are 3 values with the indices of 0, 1, 2

Index	Value
0	atg
1	aaa
2	agg

Let's access the 0th value.

```
1  >>> codons = [ 'atg' , 'aaa' , 'agg' ]
2  >>> codons[0]
3  'atg'
```

The value can be saved for later use by storing in a variable.

```
1  >>> codons = [ 'atg' , 'aaa' , 'agg' ]
2  >>> first_codon = codons[0]
3  >>> print(first_codon)
4  atg
```

> Each value can be saved in a new variable to use later.

The values can be retrieved and used directly.

```
1  >>> codons = [ 'atg' , 'aaa' , 'agg' ]
2  >>> print(codons[0])
3  atg
4  >>> print(codons[1])
5  aaa
6  >>> print(codons[2])
7  agg
```

> The 3 values are independently accesses and immediately printed. They are not stored in a variable.

If you want to access the values starting at the end of the list, use negative indices.

```
1  >>> codons = [ 'atg' , 'aaa' , 'agg' ]
2  >>> print(codons[-1])
3  agg
4  >>> print(codons[-2])
5  aaa
```

> Using a negative index will return the values from the end of the list. For example, -1 is the index of the last value 'agg'. This value also has an index of 2.

Changing Values in a List

Individual values can be changed using the value's index and the assignment operator.

```
1  >>> print(codons)
2  ['atg', 'aaa', 'agg']
3  >>> codons[2] = 'cgc'
4  >>> print(codons)
5  ['atg', 'aaa', 'cgc']
```

What about trying to assign a value to an index that does not exist?

```
1  >>> codons[5] = 'aac'
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in <module>
4  IndexError: list assignment index out of range
```

codon[5] does not exist, and when we try to assign a value to this index we get an IndexError. If you want to add new elements to the end of a list use `codons.append('taa')` or `codons.extend(list)`. See below for more details.

Extracting a Subset of a List, or Slicing

This works in exactly the same way with lists as it does with strings. This is because both are sequences, or ordered collections of data with positional information. Remember python counts the divisions between the elements, starting with 0.

Index	Value
0	atg
1	aaa
2	agg
3	aac
4	cgc
5	acg

```
1  >>> codons = [ 'atg' , 'aaa' , 'agg' , 'aac' , 'cgc' , 'acg']
2  >>> print (codons[1:3])
3  ['aaa', 'agg']
4  >>> print (codons[3:])
5  ['aac', 'cgc', 'acg']
6  >>> print (codons[:3])
7  ['atg', 'aaa', 'agg']
8  >>> print (codons[0:3])
9  ['atg', 'aaa', 'agg']
```

codons[1:3] returns every value starting with the value of codons[1] up to but not including codons[3] codons[3:] returns every value starting with the value of codons[3] and every value after. codons[:3] returns every value up to but not including codons[3] codons[0:3] is the same as codons[:3]

List Operators

Operator	Description	Example
`+`	Concatenation	[10, 20, 30] + [40, 50, 60] returns [10, 20, 30, 40, 50, 60]
`*`	Repetition	['atg'] * 4 returns ['atg','atg','atg','atg']
`in`	Membership	20 in [10, 20, 30] returns True

List Functions

Functions	Description	Example
`len(list)`	returns the length or the number of values in list	len([1,2,3]) returns 3
`max(list)`	returns the value with the largest ascii value	max(['a','A','z']) returns 'z'
`min(list)`	returns the value with the smallest ascii value	min(['a','A','z']) returns 'A'
`list(seq)`	converts a tuple into a list	list(('a','A','z')) returns ['a', 'A', 'z']
`sorted(list, key=None, reverse=False)`	returns a sorted list based on the key provided	sorted(['a','A','z']) returns ['A', 'a', 'z']
		sorted(['a','A','z'],key=str.lower) returns ['a', 'A', 'z']

List Methods

Remember methods are used in the following format list.method().

For these examples use: `list = [1,2,3]` and `codons = [ 'atg' , 'aaa' , 'agg' ]`

Method	Description	Example
`list.append(obj)`	appends an object to the end of a list	list.append(9) ; print(list) ; returns [1,2,3,9]
`list.count(obj)`	counts the occurence of an object in a list	list.count(2) returns 1
`list.index(obj)`	returns the lowest index where the given object is found	list.index(2) returns 1
`list.pop()`	removes and returns the last value in the list. The list is now 1 value shorter	list.pop() returns 3
`list.insert(index, obj)`	inserts a value at the given index	list.insert(0,100) ; print(list) returns [100, 1, 2, 3]
`list.extend(new_list)`	adds the provided list to the end of list	list.extend(['a', 'z']) ; print(list) returns [1, 2, 3, 'a', 'z']
`list.pop(index)`	removes and returns the value of the index argument. The list is now 1 value shorter	list.pop(0) returns 1
`list.remove(obj)`	finds the lowest index of the given object and removes the value. The list is now 1 value shorter	codons.remove('aaa') ; print(codons) returns ['atg' , 'agg']
`list.reverse()`	reverses the order of the list	list.reverse() ; print(list) returns [3,2,1]
`list.copy()`	Returns a shallow copy of list. Shallow vs Deep only matters in multidementional datastructures.	
`list.sort([func])`	sorts a list using the provided function. Does not return a list. The list has been changed. Advanced list sort will be covered once writting your own functions has been disccussed.	codons.sort() ; print(codons) returns ['aaa', 'agg', 'atg']

Be careful how you make a copy of your list

```
1  >>> list=['a', 'one', 'two']
2  >>> l2=list
3  >>> l2.append('1')
4  >>> print(list)
5  ['a', 'one', 'two', '1']
6  >>> print(l2)
7  ['a', 'one', 'two', '1']
```

Not what you expected?! Both lists have changed because we only copied a pointer to the original list when we wrote `l2=list`.

Let's copy the list using the `copy()` method.

```
1  >>> list=['a', 'one', 'two']
2  >>> l2=list.copy()
3  >>> l2.append('1')
4  >>> print(list)
5  ['a', 'one', 'two']
```

There we go, we get what we expect this time!

Building a List one Value at a Time

Now that you have seen the `append()` function we can go over how to build a list one value at a time.

```
1  >>> words = []
2  >>> print(words)
3  []
4  >>> words.append('one')
5  >>> words.append('two')
6  >>> print(words)
7  ['one', 'two']
```

We start with a an empty list called 'words'. We use `append()` to add the value 'one' then to add the value 'two'. We end up with a list with two values. You can add a whole list to another list with `words.extend(['three','four','five'])`

Link to Python 3 Problem Set