Loops, Dictionaries, and Lists

Lesson 2

Before we dive into loops, dictionaries, and list, this lesson will first cover how to get help within Python and some other useful tools and functions. We will also edit our script from lesson 1 to be more efficient. Comments in blue will indicate code written in your shell text editor of choice (e.g. `nano` or `vim`) and comments in brown will indicate commands used in the terminal (bold indicates commands you need to type in).

Getting help in Python using `help()` and `dir()`

You can use the `help()` function to obtain more information on a variable. This may not work for all variables, however, in which case you will need to enter the variable type within the parentheses to get a list of the command options that are allowed with that particular variable type. For example, you can enter `help(float)` into Python to further understand how to use `float` or `help(str)` for more information on strings. To escape out of the help screen just press `q`. Although, many people prefer to use other options because the `help()` function does not give clear examples of when and how a specific command is used. A web search generally produces more understandable results, especially for people who are new to Python or programming. Some commonly used resources are:

https://docs.python.org/3/
https://python/about.com/
https://learningnetwork.cisco.com/community/learning_center/python-programming-training-videos
https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial ← *Resource for learning/practicing regular expressions*

There are many more resources but be aware of the version the website or document is referring to. The website https://docs.python.org/3/ allows you to browse various versions of Python if you are interested in information about an older version.

Using the `dir()` function is slightly different than `help()`. This function will list all of the attributes nested within the specific variable you put within the parentheses. Excluding the names that are surrounded by -, you can use the other commands using dot notation.
For example:

```
host:~ name$ python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more
information.
>>> DNA='GTCAATG'
>>> print(DNA)
GTCAATG
>>> dir(DNA)
['__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', 'isascii',
'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']
>>> DNA.isdigit()
False
>>> DNA.isalnum()
True
>>> DNA.swapcase()
gtcaatg
```

Recall that '>>>' indicates the commands are being executed in Python.

Adding to dnaexample.py

Using our script from lesson 1, `dnaexample.py`, we will add new code as well as edit some of the code we already wrote to make it more efficient. First, we are going to determine the melting temperature (Tm) of the sequence. To begin, make the sequence longer and comment out the line of code that allows you to enter a sequence (The line that contains the `input()` function). You can create your own sequence, just be aware that your outputs may be different than those listed here (for now, just make sure your sequence is longer than 14 base pairs).

```
 #! /usr/bin/env python

DNASeq = 'ATGTCTCATTCAAAGCA'
#DNASeq = input("Enter a DNA sequence: ")
```

One approach for calculating the melting temperature is focusing on the strongly bound (G and C) and weakly bound (A and T) nucleotide pairs. Add the following code to the end of `dnaexample.py`:

```
TotalStrong = NumberG + NumberC
TotalWeak = NumberA + NumberT
MeltTemp = (4 * TotalStrong) + (2 * TotalWeak)
print("Melting Temp: %.1f C" % (MeltTemp))
```

Save and run the full script in the terminal and your output should say the melting temperature (`46.0 C`) below the output from the initial script.

`if` statement

This equation, however, is better suited for shorter DNA sequences (<14 base pairs). For more information on calculating the melting temperatures of different length DNA sequences see Panjkovich and Melo, 2005. You can use an `if` statement to give conditional rules, *e.g.,* execute formula X on DNA sequences above a certain length and formula Y on DNA sequences below a certain length. `if` statements in Python use indents to allow Python to understand what you want the conditions to be and when to move on from the statement to other commands. Some other languages use indents for visual purposes and require brackets, or something similar, to encompass the statement. Indents are required in Python and the script won't run properly without them. You can use spaces or tabs for indents. Tabs are recommended because you need to use the same amount of spaces for each line and that can get messy, especially when sharing code with others. A shortcut: Highlight multiple lines you would like indented and press 'command]' or press 'command [' to move the highlighted lines in the opposite direction. Taking all of this into account, add the following code above the line that calculates the melt temp:

```
if SeqLength >= 14:
     MeltTempLong = 64.9 + 41 * (TotalStrong - 16.4) / SeqLength
     print ("Tm Long (>14): %.1f C" % (MeltTempLong))
```

Note the colon after the first line beginning with `if`, this is crucial. Generally, this code is saying, "If the sequence length is greater than 14 nucleotides, do this calculation, and print the result in this statement." In this case, if the DNA sequence is not greater than 14 base pairs, the rest of the lines in the `if` statement are skipped and the next set of commands will be executed.

Adding `else:` to an `if` statement

With the above code, the melting temperature of the DNA sequences that are less than 14 base pairs long will not be calculated. We can include an `else:` statement, which will say, “If the sequence length is greater than 14 nucleotides, do this calculation, but if this is false do a different calculation.” We are including an ‘alternate route’ for the statement to move to if the first condition is false. We can edit the original melt temp calculation to add an `else:` condition. The entire statement should look as follows:

```
if SeqLength >= 14:
     MeltTempLong = 64.9 + 41 * (TotalStrong - 16.4) / SeqLength
     print ("Tm Long (>14): %.1f C" % (MeltTempLong))
else:
     MeltTemp = (4 * TotalStrong) + (2 * TotalWeak)
     print ("Tm Short: %.1f C" % (MeltTemp))
```

Note: A colon is required after the first line beginning with if and the line beginning with else.

Try out different length DNA sequences to make sure your script is running properly.

Using `elif` to shorten script

Sometimes, when you have a long list of conditions for an `if` statement, your script can become very indented and it can be difficult to read. The command `elif` (a combination of <u>el</u>se and <u>if</u>) can be used to create a cleaner script. For example, if you are assigning multiple names with numbers you may use this code:

```
if X == 1:
     Value = "one"
else:
     if X == 2:
          Value = "two"
     else:
          if X == 3:
               Value = "three"
```

You can see how this can get messy quickly with all of the indentation. Also, note the double equals sign (==) after the `if`. A single equals sign would assign ‘`1`’ to ‘`X`’, for example. And make sure you remember the colons! Making this section of code slightly more simplistic, we can use the `elif` command:

```
if X == 1:
     Value = "one"
elif X == 2:
     Value = "two"
```

```
elif X == 3:
     Value = "three"
```

There is an even better way to write this code using lists, which we will get to later in this lesson.

`for` loops

This example will use amino acids and their molecular weights to determine the total weight of a protein. A `for` loop cycles through a list of items executing a set of commands for each item. Remember that an `if` statement is exactly that; a statement. Many people get confused and think of `if` statements as loops. So be sure to keep `if` statements and `for` loops separate in your head, however, they have a similar format. An example of a `for` loop in pseudocode:

```
for Object in MyData
     run this command with Object
     now run this command with Object
     ...continue with lines of commands for Object
Return back to main commands
```

This section of code is saying, “For all of the items in MyData, call each item Object and run it through the lists of commands. Do this with each item in MyData and then exit the loop and continue on with the rest of the script.” Like with `if` statements, Python knows where the `for` loop ends because it is no longer indented. Usually, MyData will be in a `list`. MyData may be a list of integers 1-10. This list would be created with square brackets as follows:

```
>>> MyData = [1,2,3,4,5,6,7,8,9,10]
>>> MyData
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The function `list()` can translate a string into a list. This function will save you time for not having to type all of the commas and quotations needed in a list.

```
>>> MyData2 = list(123456)
>>> MyData2
['1', '2', '3', '4', '5']
```

Begin a new script called `proteinweight.py` and save it to your scripts folder (or wherever you prefer to save it). Remember to use the shebang (`#!`) on the first line. Also, make sure the file is executable using the `chmod` command from lesson 1. Add the following code to your new script:

```
#! /usr/bin/env python
```

```
ProteinSeq = "FDILSATFTYGNR"
for AminoAcid in ProteinSeq:
     print(AminoAcid)
```

If you run this in the terminal, it should print out a stacked list of all the letters in `ProteinSeq`. When learning how to use loops, one thing that can be confusing is that, in this example, the variable `AminoAcid` was never mentioned before. This is because `AminoAcid` is assigned to an item in `ProteinSeq` every time it loops through. If this is still fuzzy, you will understand this idea more with further practice.

Switching back to the python script from lesson 1, `dnaexample.py`, we can use some of these commands to cut down on some lines of code within the script. Instead of calculating the percentage of each nucleotide in the DNA sequence separately, we can use one loop. The previous code that we used looked like this:

```
print ("A occurs in %.1f%% of %d bases." % (100 *
     NumberA/SeqLength, SeqLength))
print ("C occurs in %.1f%% of %d bases." % (100 *
     NumberC/SeqLength, SeqLength))
print ("T occurs in %.1f%% of %d bases." % (100 *
     NumberT/SeqLength, SeqLength))
print ("G occurs in %.1f%% of %d bases." % (100 *
     NumberG/SeqLength, SeqLength))
```

Creating a loop will significantly cut down on the length of code needed.

```
DNASeq = 'ATGTCTCATTCAAAGCA'

BaseList = "AGTC"
for Base in BaseList:
     Percent = 100 * DNASeq.count(Base) / SeqLength
     print ("%s: %4.1f%%" % (Base, Percent))
```

Dictionaries

Dictionaries are similar to lists but they are slightly more complex. Unlike lists, dictionaries are in no particular order and contain keys, each of which contains a specified value. You can also have a dictionary of lists where each key contains a list instead of a single value. To create a dictionary write the name of your dictionary followed by curly brackets `{}`. This creates an empty dictionary. To add components to the dictionary, add the first key and the value you'd like to be associated with that key separated by a colon. This is the basic pattern for each key and value, all separated by commas. Example:

```
>>> MyData = {}        ← Creates empty dictionary
>>> MyData
{}
>>> MyData = {'Parrot':14, 'Turtle':28, 'Sheep':4, 'Snail':17}
```

In this example, the dictionary `MyData` contains keys, which are animal names, and each animal is assigned a number. This data may represent the number of each animal in a wildlife rehabilitation center. You can use this dictionary to obtain the number of an animal using the name of the animal.

```
>>> print (MyData['Turtle'])
28
```

Notice that when calling a key from a dictionary, you need to use the square brackets `[]`. Python then tells you the value associated with that key. You can also make a dictionary on multiple lines instead of one very long line.

```
>>> MyData = {
... 'Parrot':14,
... 'Turtle':28,
... 'Sheep':4,
... 'Snail':17}
>>> MyData
{'Parrot': 14, 'Turtle': 28, 'Sheep': 4, 'Snail': 17}
```

Note: This can only be done in Python when a statement is contained between brackets or parentheses (`{ }`, `()`, or `[]`).

Now we will create a dictionary with each amino acid as a key and its molecular weight as the value. We will obtain the data from the web using regular expressions. (In case you need to practice regular expressions: https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial). Open https://practicalcomputing.org/aminoacid.html in your browser. It should be a table of amino acids, the abbreviated name, the letter name, and molecular weight. We won’t want all of this information so we will remove some parts later. First, we want to find the source code. To do this, select View on the browser page and find ‘Page Source’ of ‘View Source’, which may be in the tab ‘Developer’. Right clicking and selecting ‘View Page Source’ may also work. The exact location of the source code may be slightly different on some computers. This will open a new page of code. We do not want all of this code. Find the first line that begins with `<tr><td>Alanine` and copy through the line that begins with

`<tr><td>Stop`. A line that begins with `<tr><td>` denotes the beginning of a table row boundary. Paste this code into a blank text editing file (*e.g.,* BBEdit, TextWrangler). Since we only want the amino acid letter abbreviation and the molecular weight, we can use regular expressions to remove the parts of the code that we do not want.

First, create a search term that will grab both of the items wanted from the code (letter abbreviation and molecular weight). In the 'Find' tool of the text editor you are using, type the following into the search term:

```
.+(.)</td><td>([\d\.]+).+
```

Remember that the sections within parentheses are the parts of the code that we want to capture. In this search term we used the period instead of `\w` because there are symbols other than letters, numbers, and an underscore within the last line of code. For the replacement string, we will put the captured code within the format for a dictionary (the key within single quotes and the value after a colon):

```
'\1':\2,
```

Select replace all and your text editor page should look as follows:

```
'A':89.09,
'R':174.20,
'N':132.12,
'D':133.10,
'C':121.15,
'Q':146.15,
'E':147.13,
'G':75.07,
'H':155.16,
'I':131.17,
'L':131.17,
'K':146.19,
'M':149.21,
'F':165.19,
'P':115.13,
'S':105.09,
'T':119.12,
'W':204.23,
'Y':181.19,
'V':117.15,
'X':0.0,
'-':0.0,
'*':0.0,
```

Now you can make this section of code a proper dictionary by adding a dictionary name on top followed by a curly bracket, removing the last comma, and adding the closing curly bracket.

```
AminoDict = {
'A':89.09,
'R':174.20,
'N':132.12,
'D':133.10,
... lines omitted
'V':117.15,
'X':0.0,
'-':0.0,
'*':0.0
}
```

Add this dictionary code to the `proteinweight.py` script directly above the line that defines `ProteinSeq` and under the shebang line. Now we have the basics for a script that will calculate the total weight of a protein. We need to modify the original `for` loop so it contains the molecular weight.

```
MolWeight = 0
for AminoAcid in ProteinSeq:
     MolWeight = MolWeight + AminoDict[AminoAcid]
```

This loop now accumulates the molecular weight of the protein as it adds each amino acid weight throughout each loop. The last section of code for this script is to add a print statement so we can see the protein sequence and the total weight.

```
print ("Protein: ", ProteinSeq)
print ("Molecular weight: %.1f" % (MolWeight))
```

Like the script `dnaexample.py`, you could change the code so any protein sequence could be added using the `input()` function. If you decide to use this, it is safe to add the `.upper()` method in case of mistakes. The full script should look similar to this:

```
#! /usr/bin/env python

AminoDict = {
'A':89.09,
'R':174.20,
'N':132.12,
'D':133.10,
'C':121.15,
'Q':146.15,
'E':147.13,
'G':75.07,
'H':155.16,
'I':131.17,
'L':131.17,
'K':146.19,
'M':149.21,
'F':165.19,
'P':115.13,
'S':105.09,
'T':119.12,
'W':204.23,
'Y':181.19,
'V':117.15,
'X':0.0,
'-':0.0,
'*':0.0
}

ProteinSeq = "FDILSATFTYGNR"
MolWeight = 0
for AminoAcid in ProteinSeq:
     MolWeight = MolWeight + AminoDict[AminoAcid]

print ("Protein: ", ProteinSeq)
print ("Molecular weight: %.1f" % (MolWeight))
```

More useful functions for dictionaries

The `get()` function can extract values from dictionaries by placing the key within the parentheses. Similar to using the command `AminoDict['T']` to extract the value associated with the `'T'` key, you could instead use `AminoDict.get('T')` for the same results. Watch the difference in square brackets and parentheses.

The `keys()` function will list all of the keys inside a dictionary and is used similar to `get()`. Since keys are not listed in any particular order, you can print out a list of sorted keys used the `sorted()` function. You can also save your dictionary in a sorted manner if you wish to use them in a loop and want the objects to be in order.

```
>>> sorted(AminoDict.keys())
```

```
['*', '-', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L',
'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'X', 'Y']
>>>
>>> SortedKeys = sorted(AminoDict.keys())    ← Saving sorted list in 'SortedKeys'
>>> SortedKeys
['*', '-', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L',
'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'X', 'Y']
>>> for MyKeys in SortedKeys:
...       print (MyKeys)
...
*
-
A
C
D
E
F
G
H
I
K
L
M
N
P
Q
R
S
T
V
W
X
Y
```

You can use the `.values()` function to view a list of the values within the dictionary. When using they `.keys()` and `.values()` functions, the order will not necessarily be alphabetical or numerical, however, the lists will be in the same order relative to each other.

```
>>> AminoDict.keys()
dict_keys(['A', 'R', 'N', 'D', 'C', 'Q', 'E', 'G', 'H', 'I',
'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W', 'Y', 'V', 'X', '-',
'*'])
>>> AminoDict.values()
dict_values([89.09, 174.2, 132.12, 133.1, 121.15, 146.15,
147.13, 75.07, 155.16, 131.17, 131.17, 146.19, 149.21, 165.19,
115.13, 105.09, 119.12, 204.23, 181.19, 117.15, 0.0, 0.0, 0.0])
```

Lists

Lists are commonly used in Python and can make creating scripts much easier. Unlike dictionaries, lists have a specific order. Lists can be a mixture of data types. You might use a list if you have a dataset containing the mass of 20 ducks. Instead of creating a new variable for each duck's weight, you can create a list which contains all of the weights. This format allows you to easily run this data through a loop later on. Lists are created using square brackets, `[]`. Remember that dictionaries are created using curly brackets, `{}`. Earlier, we used the `list()` command to easily create a list out of a string.

```
>>> MyList = list('abcdefg')
>>> MyList
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

You can also use the `range()` function for a list of integers. The range of numbers wanted are separated by a comma, but the number to the right of the comma (the end of the range) will not be included.

```
>>> NumList = list(range(0,9))
>>> NumList
[0, 1, 2, 3, 4, 5, 6, 7, 8]      ← Note that 9 is not included
```

Retrieving values from a list can be slightly confusing, one reason being the first value in a list is item '0', not '1'. You can retrieve a single value or a range of values using a colon. This example shows how to select a single value from a list and a range of values:

```
>>> MyList[0]
'a'
>>> MyList[2:5]
'c', 'd', 'e'
```

Notice the first line that has a `0` between the brackets retrieves the first value in the list, `'a'`. Replacing the `0` with a `1` would retrieve `'b'`. As for the second command, note that the 5th value in the list, `'f'`, was not printed in the output. The easiest way to remember how to retrieve the values you want is to think of the first number in the command (2) as inclusive and the second number (5) as exclusive. So the value associated with the number to the right of the colon will not be included in your search. You can also include only one number before or after the colon to get the range of values up to or after the specified starting/ending point.

```
>>> MyList[:4]
```

```
['a', 'b', 'c', 'd']
>>> MyList[5:]
['f', 'g']
```

You can also use negative numbers when retrieving values from lists. In this way, the numbers begin at the end of the list instead of the beginning. For example, `'a'` is in position 0 when using positive numbers but switches to position -7 when using negative numbers. Similarly, `'g'` is in position -1 instead of 6.

```
>>> MyList[0]
'a'
>>> MyList[-7]
'a'
>>> MyList[6]
'g'
>>> MyList[-1]
'g'
>>> MyList[-5:-2]
['c', 'd', 'e']
>>> MyList[-3:0]     ← Returns empty brackets
[]
>>> MyList[-3:]
['e', 'f', 'g']
```

You can see that adding a `0` to the right of the colon returns empty brackets. If you are retrieving values using negative numbers leave the right side of the colon blank to return all of the values from the list starting from the specified value. Adding a third value to a range will return the values in specified steps. If you want every other value, add `:2` to the range. You can reverse the list by making the step number negative.

```
>>> MyList[0:6:2]
['a', 'c', 'e']
>>> MyList[::3]       ← Returns the entire list by steps of 3
['a', 'd', 'g']
>>> MyList[::-1]      ← Reverses direction of list
['g', 'f', 'e', 'd', 'c', 'b', 'a']
>>> MyList[-2:-5:-2]
['f', 'd']
```

These same steps can be used with the `range()` function that was mentioned earlier, with slight differences. You can create a list of negative and positive integers and specify the steps.

```
>>> list(range(0,20,2))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> list(range(-10,11,2))
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]
>>> list(range(0,8,-1))       ← This will not work to reverse the direction of the integers
[]
>>> list(range(8,0,-1))       ← Note that now the list will not include 0 but will include 8
[8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(-11,-5))
[-11, -10, -9, -8, -7, -6]
>>> list(range(-11,-5,-1))    ← The same goes for negative numbers
[]
>>> list(range(-5,-11,-1))
[-5, -6, -7, -8, -9, -10]
```

If you need to create a variable from a value in a list you can simply call out the value of interest and assign it a name. You can do this for multiple variables within the same command as well.

```
>>> Letter1 = MyList[0]
>>> Letter1
'A'
>>> Letter2, Letter3 = MyList[1:3]
>>> Letter2
'b'
>>> Letter3
'c'
```

Note: Remember that the value associated with the number to the right of the colon will be excluded.

Imagine you just ran as assay using a 96 well plate and now you want a vertical list of all the well names (*i.e.,* A1, A2, A3 … H11, H12) instead of a table format with each letter as a row and each number as a column. We can use nested loops that assign the correct amount of letter/integer pairs and place them in order. We will need to use the `chr()` function, which returns the corresponding character based on the ASCII number. (Here is an ASCII character table if you would like to know more: https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html). This loop will go through characters A through H assigning the integers 1 through 12 along the way.

```
#! /usr/bin/env python

for Let in range(65, 73):
    for Num in range(1,13):
        print (chr(Let) + str(Num))
```

```
A1
A2
A3
A4
A5
… lines omitted
H7
H8
H9
H10
H11
H12
```

Note: Don't forget the colon at the end of each line that begins with 'for'.

Strings vs lists

Strings and lists are similar in many ways, however, individual items in a string cannot be changed like they can in a list.

```
>>> String = 'ABCDEFG'
>>> List = list('ABCDEFG')
>>> String[4]
'E'
>>> List[4]
'E'
>>> String[4] = 'P'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> List[4] = 'P'
>>> List
['A', 'B', 'C', 'D', 'P', 'F', 'G']
```

In this example, you can see that Python returned an error when attempting to change a character in the string but allowed the same character to be changed in the list. Earlier in the lesson you learned how to create a list and in this same fashion you can convert a string into a list.

```
>>> String = 'ABCDEFG'
>>> StringNoMore = list(String)
>>> StringNoMore
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

The `.join()` method can be used to combine lists of strings or individual characters. You will need to specify how you want the lists or characters joined. This method can be confusing because it is slightly different than most, so here are some examples using a list of colors:

```
>>> ColorList = ['Red', 'Orange', 'Yellow', 'Green', 'Blue',
'Violet']
>>> ColorList
['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Violet']
>>> ''.join(ColorList)        ← Joining items of ColorList with no space
'RedOrangeYellowGreenBlueViolet'
>>> '\t'.join(ColorList)
'Red\tOrange\tYellow\tGreen\tBlue\tViolet'
>>> '+'.join(ColorList)
'Red+Orange+Yellow+Green+Blue+Violet'
>>> ' '.join(ColorList)
'Red Orange Yellow Green Blue Violet'
```

As you can see, whatever you place between the single quotes before the `.join()` method will be placed between each of the items within the list. This can be useful for creating a string from a list of characters and many other tasks that would otherwise be very time consuming.

There may be times when you want to add an item to a list. You can do this by using the `.append()` function. You cannot use this function on a new list. The list must already exist.

```
>>> List = list('123456')
>>> List
['1', '2', '3', '4', '5', '6']
>>> List.append('7')
>>> List
['1', '2', '3', '4', '5', '6', '7']
```

The `7` added to this list was placed at the end of the list. You can add elements to specified locations as well using matching indicies.

```
>>> List[0:0] = ['hello']      ← Place 'hello' in position 0
>>> List
['hello', '1', '2', '3', '4', '5', '6','7']
>>> List[8:8] = ['goodbye']
>>> List
['hello', '1', '2', '3', '4', '5', '6','7','goodbye']
```

You can replace list values by selecting the range you want removed. By placing the same number on both sides of the colon (the first two examples), nothing will be removed and your item will be inserted into that position. Similarly, you can remove sections of lists by specifying the elements you wish to remove and replacing them with an empty list.

```
>>> List = list(range(1,11))
>>> List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> List[2:6] = []
>>> List
[1, 2, 7, 8, 9, 10]
```

You can check if a value is in a list and Python will return true or false.

```
>>> List = list(range(0,5340))
>>> 386 in List
True
>>> 6329 in List
False
```

The method `.sort()` is a simple way to sort lists and will directly change the variable. If you want to keep the original list, assign a new name to the sorted list. This works for integers and characters.

```
>>> NumList = [6, 19, 34, 5, 12, 17, 8, 26, 29, 3]
>>> NumList.sort()
>>> NumList
[3, 5, 6, 8, 12, 17, 19, 26, 29, 34]
>>> CharList = ['g', 'r', 'p', 'c', 'd', 'e', 'k', 'y']
>>> CharList.sort()
>>> CharList
['c', 'd', 'e', 'g', 'k', 'p', 'r', 'y']
>>> NumList2 = sorted(NumList)
>>> NumList = [6, 19, 34, 5, 12, 17, 8, 26, 29, 3]
>>> NumList
[6, 19, 34, 5, 12, 17, 8, 26, 29, 3]
>>> NumList2
[3, 5, 6, 8, 12, 17, 19, 26, 29, 34]
```

The `set()` function lists the unique elements within a list. For example, if you have a dataset with certain field sites grouped together by color within a list, you can use this function to see which colors are used. You might be organizing field notebooks and want to color code each group so you need to find out what color pens you need. To do this you could do the following:

```
>>> FieldColors = ['green', 'yellow', 'red', 'yellow', 'blue',
'green', 'green','yellow', 'pink', 'green', 'orange', 'blue',
'orange', 'red', 'white']
>>> list(set(FieldColors))
['green', 'red', 'orange', 'yellow', 'pink', 'blue', 'white']
```

This is a very simplistic example but this function can be particularly helpful for long lists that would be excruciating to count by hand. This function also can be used to make sure there aren't any typos or elements that you want removed from the list.

Other ways to modify lists

You can manipulate lists in many ways, often with simple commands. The star symbol, `*`, can be used to copy a list and add that copy to the original list.

```
>>> List = list('1234')
>>> DoubleList = List * 2
>>> DoubleList
['1', '2', '3', '4', '1', '2', '3', '4']       ← Original list repeated
```

`for` loops are often used to apply a desired change or calculation to each list value. Two stars, `**`, represent an exponent.

```
>>> Values = list(range(1,11))
>>> Values
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Squares = []       ← Create empty list to save modified elements into
>>> for Value in Values:      ← For each element in the Values list…
...     Squares.append(Value**3)     ← Cube each element and add it to the Squares list
...
>>> Squares
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Note: Remember to press 'return' twice to exit a code block in Python.

A more simple way to do this is with list comprehension (more info here: https://www.pythonforbeginners.com/basics/list-comprehensions-in-python). In general, you can modify your list with a set of square brackets using an expression and a `for` loop.

```
>>> Values = list(range(1,11))
>>> Values
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Squares = [Element**3 for Element in Values]
>>> Squares
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Instead of looping through each element in the list, there is a single loop with all the calculations done one after the other. List comprehension can be used, for example, to extract a subset of nucleotides from a list of genes. Remember from earlier, when retrieving items from a list, the first item starts with 0 and the element in the place of the number to the right of the colon will not be included.

```
>>> GeneList = ['AGTTCAGAAT', 'TGCTAAGTAC', 'ACGAAGTCCC']
>>> FirstCodons = [Seq[0:3] for Seq in GeneList]
>>> FirstCodons
['AGT', 'TGC', 'ACG']
```

So, in this example, the variable named `Seq` looped through each of the elements in the list `GeneList` and extracted the first 3 elements within the string and added that to the new list `FirstCodons`. We could add other commands within a list comprehension. For example, you may want to add each of the elements in the `FirstCodon` list to another DNA sequence by simply adding a plus symbol, `+`.

```
>>> GeneList = ['AGTTCAGAAT', 'TGCTAAGTAC', 'ACGAAGTCCC']
>>> AddOn = 'TGGACT'
>>> NewGeneList = [(AddOn + Seq[0:3]) for Seq in GeneList]
>>> NewGeneList
['TGGACTAGT', 'TGGACTTGC', 'TGGACTACG']
```

In the same manner, you could count the number of a certain nucleotide within each gene using a list comprehension.

```
>>> [Seq.count('T') for Seq in NewGeneList]
[3, 3, 2]
```

Note: Since this was not stored within a variable name Python prints out the return directly. If it were saved in a variable you would need to call out the variable name to print out the output.

List comprehension can also be used to change a list of integers to strings, which can be tricky otherwise.

```
>>> Pink = list(range(0,10))
>>> Pink
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> Orange = [str(N) for N in Pink]
>>> Orange
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

References

Haddock S, Dunn C (2010) Practical computing for biologists. Sunderland (Massachusetts): Sinauer Associates.

Alejandro Panjkovich, Francisco Melo, Comparison of different melting temperature calculation methods for short DNA sequences, *Bioinformatics*, Volume 21, Issue 6, 15 March 2005, Pages 711–722, https://doi.org/10.1093/bioinformatics/bti066