Python 3

Sequences

In the next section, we will learn about strings, tuples, and lists. These are all examples of python sequences. 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 different types of data. We'll see 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

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

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

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

What object-specific functions or "methods" go with my object?

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

```
>>> dir('ACGTGA')
['__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 methods. Methods are functions belonging to a specific class (object type).
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 quotation marks:

- Single and double quotes are equivalent.
- A variable name inside quotes is just the string identifier, not the value stored inside the variable. `f''` or f-strings are useful for variable interpolation in python
- Triple quotes (single or double) are used before and after a string that spans multiple lines.

Use of quotation examples:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences. And goes
on and on.
"""
```

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.

```
>>>print("ATG")
ATG
```

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

```
>>>dna = 'ATG'
ATG
>>> print(dna)
ATG
```

What happens if we put the variable in quotes?

```
>>>dna = 'ATG'
ATG
>>> print("dna")
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.

```
>>> print("ATG","GGTCTAC")
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.

```
>>> print("ATG"+"GGTCTAC")
ATGGGTCTAC
>>> combined_string = "ATG"+"GGTCTAC"
ATGGGTCTAC
>>> print(combined_string)
ATGGGTCTAC
```

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

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

Now, lets print a variable and a literal string.

```
>>>dna = 'ATG'
ATG
>>> print(dna,'GGTCTAC')
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?

```
>>>dna = 'ATG'
ATG
>>> print(dna + 'GGTCTAC')
ATGGGTCTAC
```

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

```
>>>dna = 'ATG'
ATG
>>> print(dna)
ATG
>>>dna = 'TTT'
TTT
>>> print(dna)
TTT
```

> The new value of the variable 'dna' is printed to the screen when `dna` is an argument for the `print()` function.

`print()` and Common Errors

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

What will happen if you forget to close your quotes?

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

> We get a 'SyntaxError' if the closing quote is not used

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

```
>>> print(GGTCTAC)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'GGTCTAC' is not defined
```

> We get a 'NameError' when the literal string is not enclosed in quotes because Python is looking for a variable with the name GGTCTAC

```
>>> print "boo"
  File "<stdin>", line 1
    print "boo"
              ^
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	Carriage Return
\t	Tab

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

```
>>> string_with_newline = 'this sting has a new line\nthis is the second line'
>>> print(string_with_newline)
this sting has a new line
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 (""").

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

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

```
>>> line = "value1\tvalue2\tvalue3"
>>> print(line)
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 character to force it to be printed as a literal. This is called 'escaping'. This is only really useful for printing literal quotes ' and "

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

```
>>> line = r"value1\tvalue2\tvalue3"
>>> print(line)
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 '+'

```
>>> promoter= 'TATAAA'
>>> upstream = 'TAGCTA'
>>> downstream = 'ATCATAAT'
>>> dna = upstream + promoter + downstream
>>> print(dna)
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)?

```
>>> 4+3
7
```

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

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

```
>>> str(4) + str(3)
'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

```
>>> print(dna)
TAGCTATATAAAATCATAAT
>>> len(dna)
20
```

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

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

```
>>> dna_length = len(dna)
>>> print(dna_length)
20
```

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

```
>>> print("The lenth of the DNA sequence:" , dna , "is" , dna_length)
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.

```
dna = "ATGCTTG"
```

> Look familiar?

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.

```
>>> dna = "ATGCTTG"
>>> dna.lower()
'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.

```
>>> print(dna)
ATGCTTG
>>> dna_lowercase = dna.lower()
>>> print(dna)
ATGCTTG
>>> print(dna_lowercase)
atgcttg
```

The string method can be nested inside of other functions.

```
>>> dna = "ATGCTTG"
>>> print(dna.lower())
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.

```
>>> nt_count = 6
>>> dna_lc = nt_count.lower()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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.

```
>>> dna = 'attgct'
>>> dna.upper()
'ATTGCT'
>>> print(dna)
attgct
```

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

Find and Count

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.

```
>>> dna = 'ATTAAAGGGCCC'
>>> dna.find('T')
1
>>> dna.find('N')
-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. `count(str)` returns the number (as an int) of exact matches of str it found

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

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

Replace one string with another

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

```
>>> dna = 'ATGCTGCATT'
>>> dna.replace('T','U')
'AUGCUGCAUU'
>>> print(dna)
ATGCTGCATT
>>> rna = dna.replace('T','U')
>>> print(rna)
AUGCUGCAUU
```

> All occurrences of T are replaced by U. The new string is returned. The original 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.

You can get to any part of a string with the following syntax [start : end : step].

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 start counting at 0 and remember that python counts the gaps between each character, starting with zero.

```
index      0   1   2   3   4   5   6   7   8 ...
string       A   T   T   A   A   A   G   G  ...
```

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

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

Let's return the first 6 characters.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[0:6]
>>> print(sub_dna)
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.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[6:]
>>> print(sub_dna)
GGGCCC
```

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

Let's return the last 3 characters.

```
>>> sub_dna = dna[-3:]
>>> print(sub_dna)
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.

Reverse a string or a list

There is no reverse function, you need to use a slice with step -1 and empty start and end.

For a string, it looks like this

```
>>> dna='GATGAA'
>>> dna[::-1]
'AAGTAG'
```

Other String Methods

Since these are methods, be sure to use in this syntax `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.

split

`split` is a method or a way to break up a string on a set of characters. What is returned is a list of elements with the characters that were used for breaking are removed. We will be going over lists in more detail in the next session. Don't get too worried about this.

Lets look at this string:

```
00000xx000xx000000000000xx0xx00
```

Let's split on 'xx' and get a list of the 0's

What is the 's' in `s.split(delim)` ?

What is the 'delim' in `s.split(delim)` ?

Let's try it:

```
>>> string_to_split='00000xx000xx000000000000xx0xx00'
>>> string_to_split.split('xx')
['00000', '000', '000000000000', '0', '00']
>>> zero_parts = string_to_split.split('xx')
>>> print(zero_parts)
['00000', '000', '000000000000', '0', '00']
```

> We started with a string and now have a list with all the delimeters removed

Here is another example. Let's split on tabs to get a list of numbers in tab separated columns.

```
>>> input_expr = '4.73\t7.91\t3.65'
>>> expression_values = input_expr.split('\t')
>>> expression_values
['4.73', '7.91', '3.65']
```

join

`join` is a method or a way to take a list of elements, of things, and turn them into a string with something put in between each element. List will be covered in the next session in more detail.

Let's join a list of Ns `list_of_Ns = ['NNNNN', 'NNN', 'N', 'NNNNNNNNNNNNNNN', 'NN']` on 'xx' to get this string:

```
NNNNNxxNNNxxNxxNNNNNNNNNNNNNNNxxNN
```

What is the 's' in `s.join(list)` ?

What is the 'list' in `s.join(list)` ?

```
>>> list_of_Ns = ['NNNNN', 'NNN', 'N', 'NNNNNNNNNNNNNNN', 'NN']
>>> list_of_Ns
['NNNNN', 'NNN', 'N', 'NNNNNNNNNNNNNNN', 'NN']
>>>
>>> string_of_elements_with_xx = 'xx'.join(list_of_Ns)
>>> string_of_elements_with_xx
'NNNNNxxNNNxxNxxNNNNNNNNNNNNNNNxxNN'
```

> We started with a list now have all the elements now in one string with the delimiter added in between each element.

Let's take a list of expression values and create a tab delimited string that will open nicely in a spreadsheet with each value in its own column:

```
>>> expression_values = ['4.73', '7.91', '3.65']
>>>expression_values
['4.73', '7.91', '3.65']
>>> expression_value_string = '\t'.join(expression_values)
>>> expression_value_string
'4.73\t7.91\t3.65'
```

> print this to a file and open it in Excel! It is beautiful!!

String Formatting

Strings can be formated using new f-strings `f''`, `f""` and `f'''`

`'''`. That last one is the triple quote multiline string. 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.

```
>>> f'This sequence: {dna} is {dna_len} nucleotides long and is found in {gene_name}.'
'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 f-strings. 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 variable that needs to be inserted.

Something very nice about f-strings is that you can print int and string variable types without converting first.

You will often put f-strings inside print functions.

```
>>> print(f'This sequence: {dna} is {dna_len} nucleotides long and is found in
{gene_name}.')
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is found in Brca1.
```

There is an older function `format()` that is similar, but not as concise. Here's an example in case you see one in older code:

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

The f-string 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.

Lets right justify some numbers.

```
>>> print( f"{2:>5}" )   # 2 is the number we want to print, the characters after the
colon define
                        # the formatting e.g. > for right justify 5 for min field width
    2
>>> print( f"{20:>5}" )
   20
>>> print( f"{200:>5}" )
  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

```
>>> print( f"{2:05}" )
00002
>>> print( f"{20:05}" )
00020
```

Use a `<` to indicate left-justification.

```
>>> print( f"{2:<5} genes" )
2     genes
>>> print( f"{20:<5} genes" )
20    genes
>>> print( f"{200:<5} genes" )
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 `:_^`. The fill symbol goes before the alignment symbol.

```
>>> print( f"{2:_^10}" )
____2_____
>>> print( f"{20:_^10}" )
____20____
>>> print( f"{200:_^10}" )
___200____
```

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 round a floating point number to fewer decimal places, starting with a lot. (The default is 6.) 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.

```
>>> f'{3.141592653589793:f}'
'3.141593'   # note this is converted to a string, useful for printing
             # they are f-strings, after all, so this makes sense
>>> f'{3.141592653589793:.4f}'
'3.1416'
```

F-strings allow you to embed expressions inside string literals, so you can do things like this. Neat.

```
>>> f'sum is {3+4}'
'sum is 7'
>>> f'sum is {3.1234+4.4324:.2f}'
'sum is 7.56'
```