Lesson 6: More About Strings

Introduction to Strings

Words and sentences are fundamental to how we communicate, so it follows that we'd want our computers to be able to work with words and sentences as well.

In Python, the way we store something like a word, a sentence, or even a whole paragraph is as a string. A string is a sequence of characters contained within a pair of 'single quotations' or "double quotations". A string can be any length and can contain any letters, numbers, symbols, and spaces.

> **Note:** In computer science, sequences of characters are referred to as strings. Strings can be any length and can include any character such as letters, numbers, symbols, and whitespace (spaces, tabs, new lines).

In this lesson, we'll learn more about strings and how they are treated in Python. We will learn how to slice strings, select specific characters from strings, search strings for characters, iterate through strings, and use strings in conditional statements.

Let's get started.

For common tasks, there are often already methods written by someone smart, and working with strings is no different. In this lesson, we'll explore some of the string processing tools that come pre-built within Python's standard library.

> **Note:** Strings are immutable in Python. This means that once a string has been defined, it can't be changed. There are no mutating methods for strings. This is different from data types like lists, which can be modified once they are created.

Indexing strings

A string can be thought of as a list of characters. And like any other list, each character in a string has an index.

This means that indexing and slicing of strings works just as it does for lists and tuples. See the silly example below:

In [1]:
```
fav_fruit = 'blueberry'
```

We can select specific letters from this string using the *index* (i.e. the location of a value within a list). So, every character within a string is assigned to a numbered slot within that string. So, just like with lists and tuples, we can extract that information using the ordering and assignment within a string.

In [2]:
```
print(fav_fruit[0])
```

```
b
```

In [3]:
```
print(fav_fruit[1])
```

```
l
```

In [ ]:
```
print(fav_fruit[0])
print(fav_fruit[1])
print(fav_fruit[2])
print(fav_fruit[3])
```

> Note how the output for `fav_fruit[1]` doesn't give you the first letter of "blueberry", but instead "l". That's because list (and string) indexing begins at `0`, not `1`. Keep that in mind.

In [4]:
```
print(fav_fruit[0])
```

```
b
```

It's important to note that indices of strings must be integers. If we were to try to select a non-integer index we would get a `TypeError`. For example:

In [ ]:
```
print(fav_fruit[1.5])
```

In [ ]:
```
fruit = "Berry"
indx = fruit[6]
```

> **`IndexError`:** When indexing into a string in Python, if you try to access an index that doesn't exist, an `IndexError` is generated. For example, the following code would create an `IndexError`:

Slicing strings

More importantly, not only can we select a single character from a string, but we can also select entire chunks of characters from a string via the following syntax:

In [ ]:
```
string[first_index:last_index]
```

This is called slicing a string. When we slice a string we are creating a substring - a brand new string that starts at (and includes) the `first_index` and ends at (but excludes) the `last_index`.

Let's look at some examples of this. Recall our favorite fruit:

In [5]:
```
fav_fruit = "blueberry"
```

The indices of the string correspond to the order of the letters, starting at 0 and counting up until

you reach that last character.

So, if we wanted a new string that contains the letters `be`, we'd need to slice `fav_fruit` as follows:

In [ ]:
```
print(fav_fruit[4:6]) # output: be
```

Notice how the character at the first index, `b`, is included, but the character at the last index, `r`, is excluded. If you count out the indices 4 and 6 in the string, you can see how the `r` is outside that range.

We can also have open-ended selections. If we remove the first index, the slice starts at the beginning of the string and if we remove the second index the slice continues to the end of the string.

In [6]:
```
print(fav_fruit[:4]) # output: blue
print (fav_fruit[4:]) # output: berry
```

```
blue
berry
```

Again, notice how the `b` from `berry` is excluded from the first example and included in the second example.

Another example:

In [7]:
```
my_str = 'The Dude abides.'

print(my_str[5])
print(my_str[:6])
print(my_str[::2])
print(my_str[::-1])
```

```
u
The Du
TeDd bds
.sediba eduD ehT
```

> **Indexing and Slicing Strings:** Python strings can be indexed using the same notation as lists, since strings are lists of characters. A single character can be accessed with bracket notation (`[index]`), or a substring can be accessed using slicing (`[start:end]`).Indexing with negative numbers counts from the end of the string.

In [ ]:
```
str = 'yellow'
str[1]     # => 'e'
str[-1]    # => 'w'
str[4:6]   # => 'ow'
str[:4]    # => 'yell'
str[-3:]   # => 'low'
```

> **Iterate String:** To iterate through a string in Python, use `for...in` notation.

In [ ]:
```
str = "hello"
for c in str:
  print(c)

# h
# e
# l
# l
# o
```

Previously, in Lesson 4, we talked about counting the occurances `'G'` within a specific string using a `for` loop. A more efficient method is to use the string method `count()`.

In [8]:
```
seq = 'GACAGACUCCAUGCACGUGGGUAUCAUGUC' # DNA sequence (a string)

seq.count('G') # count G's
```

Out[8]: 8

In [9]:
```
seq.count('g')
```

Out[9]: 0

In [10]:
```
seq.count('C') # count C's
```

Out[10]: 8

In [11]:
```
# Count G's and C's
seq.count('G') + seq.count('C')
```

Out[11]: 16

> **Note:** This notation might be a bit new. We have a variable, followed by a dot (.), and then a function. These functions are called methods in the language of object-oriented programming (OOP). If you have a string my_str, and you want to execute one of Python's built-in string methods on it, the syntax is:
> `my_str.string_method_of_choice(*args)`

In general, the count method gives the number of times a substring appears in a string. We can learn more about its behavior by playing with it.

In [12]:
```
seq.count('GAC') # substrings of more than one characater
```

Out[12]: 2

In [13]:
```
'AAAAAAA'.count('AA') # substrings cannot overlap
```

Out[13]: 3

In [14]:
```
seq.count('nonsense') # something that's not there
```

Out[14]: 0

Finding the index of a start codon

Another string method, `.find()`, can be used too find the index of the start codon in A sequence.

In [15]:
```
seq.find('AUG')
```

Out[15]: 10

In [16]:
```
seq
```

Out[16]: 'GACAGACUCCAUGCACGUGGGUAUCAUGUC'

Wow, that was easy! The `find()` method gives the index where the substring argument first appears. But, what if a substring is not in the string?

In [17]:
```
seq.find('nonsense')
```

Out[17]: -1

In this case, `find()` returns `-1`. This shouldn't be interpreted as index `-1`. `find()` always returns positive indices if it finds a substring. Note that you should not use `find()` to test if a substring is present using the in operator we already learned about.

In [18]:
```
'AUG' in seq
```

Out[18]: True

> **The `in` Syntax:** The `in` syntax is used to determine if a letter or a substring exists in a string. It returns `True` if a match is found, otherwise `False` is returned..

In [19]:
```
game = "Popular Nintendo Game: Mario Kart"
```

In [ ]:
```
print("t" in game) # prints: True
```

In [ ]:
```
print("x" in game) # prints: False
```

Similar to `find()`, we can use another method `rfind()` to find the last instance of a start codon. Basically, `rfind()` searches a given string from right to left.

In [20]:
```
seq.rfind('AUG')
```

Out[20]: 25

In [21]:
```
seq
```

Out[21]: 'GACAGACUCCAUGCACGUGGGUAUCAUGUC'

In [22]:
```
len(seq)
```

Out[22]: 30

The `join()` method

One of the most useful string methods is the `join()` method. Say we have a list of words that we want to craft into a sentence.

In [23]:
```
word_tuple = ('The', 'Dude', 'abides.')
word_tuple[0] + word_tuple[1] + word_tuple[2]
```

Out[23]: 'TheDudeabides.'

Now, we want to concatenate them into a single string. (This is sort of like the opposite of taking a string and making a list of its characters by doing a `list()` type conversion.) We need to know what we want to put between each word. In this case, we want a space. Here's the nifty syntax to do that.

In [24]:
```
' '.join(word_tuple)
```

Out[24]: 'The Dude abides.'

In [25]:
```
word_tuple[0] + ' ' + word_tuple[1] + ' ' + word_tuple[2]
```

Out[25]: 'The Dude abides.'

We now have a single string with the elements of the tuple, separated by spaces. The string before the dot (`.`) specifies what goes between the strings in the list or tuple (or other iterable). If we wanted " * " between each word, we could do that, too.

In [26]:
```
' * '.join(word_tuple)
```

Out[26]: 'The * Dude * abides.'

> **String Concatenation:** To combine the content of two strings into a single string, Python provides the + operator. This process of joining strings is called concatenation.

In [ ]:
```
x = 'One fish, '
y = 'two fish.'

z = x + y
print(z) # output: One fish, two fish.
```

The format() method

The `format()` method is incredibly powerful. We won't go over all use cases here, but let's see one of the more intuitive and commonly used cases. Again, this is best learned by example.

In [27]:
```
my_str = """
Let's do a Mad Lib!
During this camp, I feel {adjective}.
The instructor give us {plural_noun}.
""".format(adjective='truculent', plural_noun='haircuts')

print(my_str)
```

```
Let's do a Mad Lib!
During this camp, I feel truculent.
The instructor give us haircuts.
```

See the pattern? Given a string, the `format()` method takes args that are themselves strings. Within the string, the name of the kwargs are given in braces. Then, the arguments in the `format()` method inserts the strings at the places delimited by braces.

Now, what if we want to insert a number into a string? We could convert it to a string, but we should instead use **string conversions**. These are short directives that specify how the number should be represented in a string. A complete list is here. The table below shows some that are commonly used.

conversion	description
`d`	integer
`04d`	integer with four digits, possibly with leading zeros
`f`	float, default to six digits after decimal
`.8f`	float with 8 digits after the decimal
`e`	scientific notation, default to six digits after decimal
`.16e`	scientific notation with 16 digits after the decimal
`s`	display as a string

In [29]:
```
print('There are {n:d} states in the US.'.format(n=50))
```

```
There are 50 states in the US.
```

In [30]:
```
print('Your file number is {n:d}.'.format(n=23))
```

```
Your file number is 23.
```

In [31]:
```
print('π is approximately {pi:f}.'.format(pi=3.14))
```

```
π is approximately 3.140000.
```

In [32]:
```
print('e is approximately {e:.8f}.'.format(e=2.7182818284590451))
```

```
e is approximately 2.71828183.
```

In [33]:
```
print("Avogadro's number is approximately {N_A:e}.".format(N_A=6.022e23))
```

```
Avogadro's number is approximately 6.022000e+23.
```

In [34]:
```
print('ε₀ is approximately {eps_0:.16e} F/m.'.format(eps_0=8.854187817e-12))
```

```
ε₀ is approximately 8.8541878170000005e-12 F/m.
```

In [35]:
```
print('That {thing:s} really tied the room together.'.format(thing='rug'))
```

```
That rug really tied the room together.
```

Note the syntax. In the braces, we specify the name of the kwarg, and then we put a colon followed by the string conversion. Note also that I used double quotes on the outside of the string containing Avogadro's number so that I could include an apostrophe in the string. Finally, note that we got a subscript zero using the Unicode character, `₀`.

f-strings

f-strings are strings that are prefixed with an `f` or `F` that allow convenient insertion of entries into strings. Such as:

In [36]:
```
n_states = 50
file_number = 23
pi = 3.14
e = 2.7182818284590451
N_A = 6.022e23
eps_0=8.854187817e-12
thing = 'rug'
```

In [ ]:
```
print(f'There are {n_states} states in the US.')
```

In [ ]:
```
print(f'Your file number is {file_number}.')
```

In [ ]:
```
print(f'π is approximately {pi}.')
```

In [ ]:
```
print(f'e is approximately {e:.8f}.')
```

In [ ]:
```
print(f"Avogadro's number is approximately {N_A}.")
```

In [ ]:
```
print(f'ε₀ is approximately {eps_0} F/m.')
```

In [ ]:
```
print(f'That {thing} really tied the room together.')
```

And so much more!

A few other string methods you may find particularly useful in your programming forays are:

- `.lower()` : returns a string with all uppercase characters converted into lowercase.
- `.upper()` : returns a string with all lowercase characters converted into uppercase.
- `.strip()` : used to remove characters from the beginning and end of a string. A string argument can be passed to the method, specifying the set of characters to be stripped. With no arguments to the method, whitespace is removed.
- `.split()` : splits a string into a list of items. If no argument is passed, the default behavior is to split on whitespace. If an argument is passed to the method, that value is used as the delimiter on which to split the string.
- `.replace()` : returns a copy of the string with all occurrences of given substring replaced by a different substring. If the optional argument `count` is given, only the first `count` occurrences are replaced.

> **Note:** You can find a complete list of string methods from the Python doc pages. Various methods will come in handy when parsing strings going forward.

Proceed to Assignment 6.