Strings

Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

In [8]:
```
# Single word
'hello'
```

Out[8]:
```
'hello'
```

In [9]:
```
# Entire phrase
'I am Abhishek'
```

Out[9]:
```
'I am Abhishek'
```

In [10]:
```
# We can also use double quote
"I am Abhishek"
```

Out[10]:
```
'I am Abhishek'
```

In [11]:
```
# Be careful with quotes!
' I'm Abhishek'
```

```
  File "<ipython-input-11-45f7c6095d6a>", line 2
    ' I'm Abhishek'
       ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in `I'm` stopped the string. You can use combinations of double and single quotes to get the complete statement.

In [12]:
```
" I'm Abhishek "
```

Out[12]:
```
" I'm Abhishek "
```

Now let's learn about printing strings!

Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

In [13]:
```
# We can simply declare a string
'Hello World'
```

Out[13]:
```
'Hello World'
```

In [14]:
```
# Note that we can't output multiple strings this way
'Hello World 1'
'Hello World 2'
```

Out[14]:
```
'Hello World 2'
```

We can use a print statement to print a string.

In [15]:
```
print('Hello World 1')
print('Hello World 2')
print('Use \n to print a new line')
print('\n')
print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

String Basics

We can also use a function called len() to check the length of a string!

In [16]:
```
len('Hello World')
```

Out[16]:
```
11
```

** Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

String Indexing

In Python, we use brackets `[]` after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called `s` and then walk through a few examples of indexing.

In [17]:
```
# Assign s as a string
s = 'Hello World'
```

In [18]:
```
#Check
s
```

Out[18]: 'Hello World'

In [19]:
```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [20]:
```
# Show first element (in this case a letter)
s[0]
```

Out[20]: 'H'

In [21]:
```
s[1]
```

Out[21]: 'e'

In [22]:
```
s[2]
```

Out[22]: 'l'

We can use a `:` to perform *slicing* which grabs everything up to a designated point. For example:

In [23]:
```
# Grab everything past the first term all the way to the length of s which
 is len(s)
s[1:]
```

Out[23]: 'ello World'

In [24]:
```
# Note that there is no change to the original s
s
```

Out[24]: 'Hello World'

In [25]:
```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[25]: 'Hel'

In [26]:
```
#Everything
s[:]
```

Out[26]: 'Hello World'

We can also use negative indexing to go backwards.

In [27]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[27]: 'd'

In [28]:
```
# Grab everything but the last letter
s[:-1]
```

Out[28]: 'Hello Worl'

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

In [29]:
```
# Grab everything, but go in steps size of 1
s[::1]
```

Out[29]: 'Hello World'

In [30]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[30]: 'HloWrd'

In [31]:
```
# We can use this to print a string backwards
s[::-1]
```

Out[31]: 'dlroW olleH'

String Properties

It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed or replaced. For example:

In [32]:
```
s
```

Out[32]:
```
'Hello World'
```

In [33]:
```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
<ipython-input-33-3a9c668aa5ab> in <module>()
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment
```

Notice how the error tells us directly what we can't do, change the item assignment!

Something we *can* do is concatenate strings!

In [34]:
```
s
```

Out[34]:
```
'Hello World'
```

In [35]:
```
# Concatenate strings!
s + ' by Abhishek'
```

Out[35]:
```
'Hello World by Abhishek'
```

In [36]:
```
# We can reassign s completely though!
s = s + ' by Abhishek!'
```

In [37]:
```
print(s)
```

```
Hello World by Abhishek!
```

In [38]:
```
s
```

Out[38]:
```
'Hello World by Abhishek!'
```

We can use the multiplication symbol to create repetition!

In [39]:
```
letter = 'a'
```

In [40]:
```
letter*10
```

Out[40]:
```
'aaaaaaaaaa'
```

Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

In [41]:
```
s
```

Out[41]: 'Hello World by Abhishek!'

In [42]:
```
# Upper Case a string
s.upper()
```

Out[42]: 'HELLO WORLD BY ABHISHEK!'

In [43]:
```
# Lower case
s.lower()
```

Out[43]: 'hello world by abhishek!'

In [44]:
```
# Split a string by blank space (this is the default)
s.split()
```

Out[44]: ['Hello', 'World', 'by', 'Abhishek!']

In [45]:
```
# Split by a specific element (doesn't include the element that was split
 on)
s.split('W')
```

Out[45]: ['Hello ', 'orld by Abhishek!']

Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

In [46]:
```
'Insert another string with curly brackets: {}'.format('The inserted strin
g')
```

Out[46]: 'Insert another string with curly brackets: The inserted string'