Strings

Strings are used in Python to record text information, such as name. 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).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
3.) Differences in Printing in Python 2 vs 3
4.) String Indexing and Slicing
5.) String Properties
6.) String Methods
7.) Print Formatting
```

Creating a String

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

In [1]:
```
# Single word
'hello'
```

Out[1]:
```
'hello'
```

In [2]:
```
# Entire phrase
'This is also a string'
```

Out[2]:
```
'This is also a string'
```

In [3]:
```
# We can also use double quote
"String built with double quotes"
```

Out[3]:
```
'String built with double quotes'
```

```
In [4]:  # Be careful with quotes!
         ' I'm using single quotes, but will create an error'
```

```
  File "<ipython-input-4-6565b0b7b5e3>", line 2
    ' I'm using single quotes, but will create an error'
        ^
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 [10]:  "Now I'm ready to use the single quotes inside a string!"
```

```
Out[10]:  "Now I'm ready to use the single quotes inside a string!"
```

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

```
Out[11]:  'Hello World'
```

```
In [12]:  # note that we can't output multiple strings this way
          'Hello World 1'
          'Hello World 2'
```

```
Out[12]:  'Hello World 2'
```

We can use a print statement to print a string.

```
In [13]:  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?
```

Python 3 Alert!

Something to note. In Python 3, print is a function, not a statement. So you would print statements like this: print('Hello World')

If you want to use this functionality in Python2, you can import form the **future** module.

A word of caution, after importing this you won't be able to choose the print statement method anymore. So pick whichever one you prefer depending on your Python installation and continue on with it.

In [32]:
```
# To use print function from Python 3 in Python 2
from __future__ import print_function

print('Hello World')
```

```
Hello World
```

String Basics

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

In [33]:
```
len('Hello World')
```

Out[33]:
```
11
```

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

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 [2]:
```
# Assign s as a string
s = 'Hello World'
```

In [35]:
```
#Check
s
```

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

In [36]:
```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [21]:
```
# Show first element (in this case a letter)
s[0]
```
Out[21]:
```
'H'
```

In [22]:
```
s[1]
```
Out[22]:
```
'e'
```

In [23]:
```
s[2]
```
Out[23]:
```
'l'
```

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

In [3]:
```
# Grab everything past the first element (including the first element) all the wa
s[1:]
```
Out[3]:
```
'ello World'
```

In [25]:
```
# Note that there is no change to the original s
s
```
Out[25]:
```
'Hello World'
```

In [4]:
```
# Grab everything UP TO the 3rd element (but not including the 3rd element)
s[:3]
```
Out[4]:
```
'Hel'
```

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

In [27]:
```
#Everything
s[:]
```
Out[27]:
```
'Hello World'
```

We can also use negative indexing to go backwards.

In [28]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```
Out[28]:
```
'd'
```

```
In [29]: # Grab everything but the last letter
         s[:-1]

Out[29]: '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 [42]: # Grab everything, but go in steps size of 1
         s[::1]

Out[42]: 'Hello World'
```

```
In [46]: # Grab everything, but go in step sizes of 2
         s[::2]

Out[46]: 'HloWrd'
```

```
 In [5]: # We can use this to print a string backwards = reverse the string
         s[::-1]

 Out[5]: 'dlroW olleH'
```

String Properties

Its 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 [48]: s

Out[48]: 'Hello World'
```

```
In [49]: # Let's try to change the first letter to 'x'
         s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-3a9c668aa5ab> 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 [50]: s
Out[50]: 'Hello World'
```

```
In [52]: # Concatenate strings!
         s + ' concatenate me!'
Out[52]: 'Hello World concatenate me!'
```

```
In [53]: # We can reassign s completely though!
         s = s + ' concatenate me!'
```

```
In [54]: print(s)

         Hello World concatenate me!
```

```
In [58]: s
Out[58]: 'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

```
In [59]: letter = 'z'
```

```
In [60]: letter*10
Out[60]: 'zzzzzzzzzz'
```

Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object (we will learn about these in much more depth later) 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 [61]: s
Out[61]: 'Hello World concatenate me!'
```

In [62]:
```
# Upper Case a string
s.upper()
```
Out[62]:
```
'HELLO WORLD CONCATENATE ME!'
```
In [65]:
```
# Lower case
s.lower()
```
Out[65]:
```
'hello world concatenate me!'
```
In [67]:
```
# Split a string by blank space (this is the default)
s.split()
```
Out[67]:
```
['Hello', 'World', 'concatenate', 'me!']
```
In [68]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```
Out[68]:
```
['Hello ', 'orld concatenate me!']
```
There are many more methods than the ones covered here. Visit the advanced String section to find out more!

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 [79]:
```
'Insert another string with curly brackets: {}'.format('The inserted string')
```
Out[79]:
```
'Insert another string with curly brackets: The inserted string'
```
We will revisit this string formatting topic in later sections when we are building our projects!

Next up: Lists!

In [ ]:
```
```