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).

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

In this Session we'll learn about the following:

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

1. Creating a String

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

In [2]:

```python
# Single word
'hello'
```

Out[2]:

```
'hello'
```

In [3]:

```python
# Entire phrase
'This is also a string'
```

Out[3]:

```
'This is also a string'
```

In [4]:

```python
# We can also use double quote
"String built with double quotes"
```

Out[4]:

```
'String built with double quotes'
```

In [5]:

```python
# Be careful with quotes!
' I'm using single quotes, but this will create an error'
```

```
  File "<ipython-input-5-da9a34b3dc31>", line 2
    ' I'm using single quotes, but this 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 [ ]:

```python
"Now I'm ready to use the single quotes inside a string!"
```

Now let's learn about printing strings!

2. 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 [6]:

```python
# We can simply declare a string
'Hello World'
```

Out[6]:

```
'Hello World'
```

In [7]:

```python
# Note that we can't output multiple strings this way
'Hello World 1'
'Hello World 2'
```

Out[7]:

```
'Hello World 2'
```

We can use a print statement to print a string.

In [8]:

```python
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 [9]:

```python
len('Hello World')
```

Out[9]:

```
11
```

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

3.1 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 [10]:

```python
# Assign s as a string
s = 'Hello World'
```

In [11]:

```python
#Check
s
```

Out[11]:

```
'Hello World'
```

In [12]:

```python
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [13]:

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

Out[13]:

```
'H'
```

In [14]:

```python
s[1]
```

Out[14]:

```
'e'
```

In [15]:

```python
s[2]
```

Out[15]:

```
'l'
```

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

In [16]:

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

Out[16]:

```
'ello World'
```

In [17]:

```python
# Note that there is no change to the original s
s
```

Out[17]:

```
'Hello World'
```

In [18]:

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

Out[18]:

```
'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 [19]:

```
#Everything
s[:]
```

Out[19]:

```
'Hello World'
```

We can also use negative indexing to go backwards.

In [20]:

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

Out[20]:

```
'd'
```

In [21]:

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

Out[21]:

```
'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 [22]:

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

Out[22]:

```
'Hello World'
```

In [23]:

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

Out[23]:

```
'HloWrd'
```

In [24]:

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

Out[24]:

```
'dlroW olleH'
```

4 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 [25]:

```
s
```

Out[25]:

```
'Hello World'
```

In [26]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-976942677f11> 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 [27]:

```
s
```

Out[27]:

```
'Hello World'
```

In [28]:

```
# Concatenate strings!
s + ' concatenate me!'
```

Out[28]:

```
'Hello World concatenate me!'
```

In [29]:

```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

In [30]:

```
print(s)
```

```
Hello World concatenate me!
```

In [31]:

```
s
```

Out[31]:

```
'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

In [32]:

```
letter = 'z'
```

In [33]:

```
letter*10
```

Out[33]:

```
'zzzzzzzzzz'
```

5 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 [34]:

```
s
```

Out[34]:

```
'Hello World concatenate me!'
```

In [35]:

```
# Upper Case a string
s.upper()
```

Out[35]:

```
'HELLO WORLD CONCATENATE ME!'
```

In [36]:

```
# Lower case
s.lower()
```

Out[36]:

```
'hello world concatenate me!'
```

In [37]:

```
# Split a string by blank space (this is the default)
s.split()
```

Out[37]:

```
['Hello', 'World', 'concatenate', 'me!']
```

In [38]:

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

Out[38]:

```
['Hello ', 'orld concatenate me!']
```

There are many more methods than the ones covered here. Visit the Advanced String section to find out more!

6. 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 [39]:

```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

Out[39]:

```
'Insert another string with curly brackets: The inserted string'
```

7. String Formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas'
points = 33

'Last night, '+player+' scored '+str(points)+' points.'  # concatenation

f'Last night, {player} scored {points} points.'          # string formatting
```

There are three ways to perform string formatting.

- The oldest method involves placeholders using the modulo `%` character.
- An improved technique uses the `.format()` string method.
- The newest method, introduced with Python 3.6, uses formatted string literals, called *f-strings*.

Since you will likely encounter all three versions in someone else's code, we describe each of them here.

Formatting with placeholders

You can use `%s` to inject strings into your print statements. The modulo `%` is referred to as a "string formatting operator".

In [40]:

```
print("I'm going to inject %s here." %'something')
```

```
I'm going to inject something here.
```

You can pass multiple items by placing them inside a tuple after the `%` operator.

In [41]:

```python
print("I'm going to inject %s text here, and %s text here." %('some','more'))
```

```
I'm going to inject some text here, and more text here.
```

You can also pass variable names:

In [42]:

```python
x, y = 'some', 'more'
print("I'm going to inject %s text here, and %s text here."%(x,y))
```

```
I'm going to inject some text here, and more text here.
```

Format conversion methods.

It should be noted that two methods `%s` and `%r` convert any python object to a string using two separate methods: `str()` and `repr()` . We will learn more about these functions later on in the course, but you should note that `%r` and `repr()` deliver the *string representation* of the object, including quotation marks and any escape characters.

In [43]:

```python
print('He said his name was %s.' %'Fred')
print('He said his name was %r.' %'Fred')
```

```
He said his name was Fred.
He said his name was 'Fred'.
```

As another example, `\t` inserts a tab into a string.

In [44]:

```python
print('I once caught a fish %s.' %'this \tbig')
print('I once caught a fish %r.' %'this \tbig')
```

```
I once caught a fish this 	big.
I once caught a fish 'this \tbig'.
```

The `%s` operator converts whatever it sees into a string, including integers and floats. The `%d` operator converts numbers to integers first, without rounding. Note the difference below:

In [45]:

```python
print('I wrote %s programs today.' %3.75)
print('I wrote %d programs today.' %3.75)
```

```
I wrote 3.75 programs today.
I wrote 3 programs today.
```

Padding and Precision of Floating Point Numbers

Floating point numbers use the format `%5.2f` . Here, `5` would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, `.2f` stands for how many numbers to show past the decimal point. Let's see some examples:

In [46]:

```python
print('Floating point numbers: %5.2f' %(13.144))
```

```
Floating point numbers: 13.14
```

In [47]:

```python
print('Floating point numbers: %1.0f' %(13.144))
```

```
Floating point numbers: 13
```

In [48]:

```python
print('Floating point numbers: %5.5f' %(13.144))
```

```
Floating point numbers: 13.14400
```

In [49]:

```python
print('Floating point numbers: %10.2f' %(13.144))
```

```
Floating point numbers:      13.14
```

In [50]:

```python
print('Floating point numbers: %25.2f' %(13.144))
```

```
Floating point numbers:                     13.14
```

For more information on string formatting with placeholders visit https://docs.python.org/3/library/stdtypes.html#old-string-formatting (https://docs.python.org/3/library/stdtypes.html#old-string-formatting)

Multiple Formatting

Nothing prohibits using more than one conversion tool in the same print statement:

In [51]:

```python
print('First: %s, Second: %5.2f, Third: %r' %('hi!',3.1415,'bye!'))
```

```
First: hi!, Second:  3.14, Third: 'bye!'
```

Formatting with the `.format()` method

A better way to format objects into your strings for print statements is with the string `.format()` method. The syntax is:

```
'String here {} then also {}'.format('something1','something2')
```

For example:

In [52]:

```
print('This is a string with an {}'.format('insert'))
```

```
This is a string with an insert
```

The .format() method has several advantages over the %s placeholder method:

1. Inserted objects can be called by index position:

In [53]:

```
print('The {2} {1} {0}'.format('fox','brown','quick'))
```

```
The quick brown fox
```

2. Inserted objects can be assigned keywords:

In [54]:

```
print('First Object: {a}, Second Object: {b}, Third Object: {c}'.format(a=1,b='Two',c=
```

```
First Object: 1, Second Object: Two, Third Object: 12.3
```

3. Inserted objects can be reused, avoiding duplication:

In [55]:

```
print('A %s saved is a %s earned.' %('penny','penny'))
# vs.
print('A {p} saved is a {p} earned.'.format(p='penny'))
```

```
A penny saved is a penny earned.
A penny saved is a penny earned.
```

Alignment, padding and precision with `.format()`

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

In [56]:

```python
print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:9}'.format('Apples', 3.))
print('{0:8} | {1:9}'.format('Oranges', 10))
```

```
Fruit    | Quantity 
Apples   |       3.0
Oranges  |        10
```

By default, `.format()` aligns text to the left, numbers to the right. You can pass an optional `<` , `^` , or `>` to set a left, center or right alignment:

In [57]:

```python
print('{0:<8} | {1:^8} | {2:>8}'.format('Left','Center','Right'))
print('{0:<8} | {1:^8} | {2:>8}'.format(11,22,33))
```

```
Left     |  Center  |    Right
11       |    22    |       33
```

You can precede the aligment operator with a padding character

In [58]:

```python
print('{0:=<8} | {1:-^8} | {2:.>8}'.format('Left','Center','Right'))
print('{0:=<8} | {1:-^8} | {2:.>8}'.format(11,22,33))
```

```
Left==== | -Center- | ...Right
11====== | ---22--- | ......33
```

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent:

In [59]:

```python
print('This is my ten-character, two-decimal number:%10.2f' %13.579)
print('This is my ten-character, two-decimal number:{0:10.2f}'.format(13.579))
```

```
This is my ten-character, two-decimal number:     13.58
This is my ten-character, two-decimal number:     13.58
```

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of ten characters.

For more information on the string `.format()` method visit https://docs.python.org/3/library/string.html#formatstrings (https://docs.python.org/3/library/string.html#formatstrings)

Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings offer several benefits over the older `.format()` string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through `.format(var)`.

In [60]:

```
name = 'Fred'

print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

Pass `!r` to get the string representation:

In [61]:

```
print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```

Float formatting follows `"result: {value:{width}.{precision}}"`

Where with the `.format()` method you might see `{value:10.4f}`, with f-strings this can become `{value:{10}.{6}}`

In [62]:

```
num = 23.45678
print("My 10 character, four decimal number is:{0:10.4f}".format(num))
print(f"My 10 character, four decimal number is:{num:{10}.{6}}")
```

```
My 10 character, four decimal number is:   23.4568
My 10 character, four decimal number is:   23.4568
```

Note that with f-strings, *precision* refers to the total number of digits, not just those following the decimal. This fits more closely with scientific notation and statistical analysis. Unfortunately, f-strings do not pad to the right of the decimal, even if precision allows it:

In [63]:

```
num = 23.45
print("My 10 character, four decimal number is:{0:10.4f}".format(num))
print(f"My 10 character, four decimal number is:{num:{10}.{6}}")
```

```
My 10 character, four decimal number is:   23.4500
My 10 character, four decimal number is:     23.45
```

If this becomes important, you can always use `.format()` method syntax inside an f-string:

In [64]:

```python
num = 23.45
print("My 10 character, four decimal number is:{0:10.4f}".format(num))
print(f"My 10 character, four decimal number is:{num:10.4f}")
```

```
My 10 character, four decimal number is:   23.4500
My 10 character, four decimal number is:   23.4500
```

For more info on formatted string literals visit https://docs.python.org/3/reference/lexical_analysis.html#f-strings (https://docs.python.org/3/reference/lexical_analysis.html#f-strings)