43b8ab8f1c

TiVentures added Variable Assignment

2 contributors

1003 lines (1003 sloc) | 20.5 KB

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 lecture 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
```

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 this will create an error'
```

```
  File "", 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 [5]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Out[5]:
```
"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 [6]:
```
# We can simply declare a string
'Hello World'
```

Out[6]:
```
'Hello World'
```

In [7]:
```
# 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]:
```
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]:
```
len('Hello World')
```

Out[9]:
```
11
```

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

spaces and punctuation.

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

In [11]:
```
#Check
s
```

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

In [12]:
```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [13]:
```
# Show first element (in this case a letter)
s[0]
```

Out[13]:
```
'H'
```

In [14]:
```
s[1]
```

Out[14]:
```
'e'
```

In [15]:
```
s[2]
```

Out[15]:
```
'l'
```

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

In [16]:
```
# Grab everything past the first term all the way to the length of s whic
s[1:]
```

Out[16]:
```
'ello World'
```

In [17]:
```
# 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'
```

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)
 in ()
      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'

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
s.split('W')
```