Strings in Python

Let's learn about String data type in Python

WE'LL COVER THE FOLLOWING

How to Create a String

Strings are usually created in one of three ways. You can use single, double or triple quotes. Let's take a look!

```
my_string = "Welcome to Python!"
another_string = 'The bright red fox jumped the fence.'
a_long_string = '''This is a
multi-line string. It covers more than
one line'''

print(my_string)
print(another_string)
print(a_long_string)
```

The triple quoted line can be done with three single quotes or three double quotes. Either way, they allow the programmer to write strings over multiple lines. If you print it out, you will notice that the output retains the line breaks. If you need to use single quotes in your string, then wrap it in double quotes. See the following example.

```
my_string = "I'm a Python programmer!"
otherString = 'The word "python" usually refers to a snake'
tripleString = """Here's another way to embed "quotes" in a string"""

print(my_string)
print(otherString)
print(tripleString)
```

The code above demonstrates how you could put single quotes or double quotes into a string. There's actually one other way to create a string and that is by using the **str** method. Here's how it works:

```
my_number = 123
my_string = str(my_number)

print(my_number)
print(my_string)
```

If you run the code above, you'll notice that you have transformed the integer value into a string and assigned the string to the variable *my_string*. This is known as **casting**. You can cast some data types into other data types, like numbers into strings. But you'll also find that you can't always do the reverse, such as casting a string like 'ABC' into an integer. If you do that, you'll end up with an error like the one in the following example:

```
int('ABC')
# ValueError: invalid literal for int() with base 10: 'ABC'
```

We will look at exception handling in a later chapter, but as you may have guessed from the message, this means that you cannot convert a literal into an integer. However, if you had done

```
x = int("123")
```

then that would have worked fine.

It should be noted that a string is one of Python immutable types. What this means is that you cannot change a string's content after creation. Let's try to change one to see what happens:

```
my_string = "abc"
my_string[0] = "d"
# TypeError: 'str' object does not support item assignment
```

Here we try to change the first character from an “a” to a “d”; however this raises a TypeError that stops us from doing so. Now you may think that by assigning a new string to the same variable that you’ve changed the string. Let’s see if that’s true:

```
my_string = "abc"
print(id(my_string)) # 140029410169552

my_string = "def"
print(id(my_string)) # 140029409549312

my_string = my_string + "ghi"
print(id(my_string)) # 140029409549200
```

By checking the id of the object, we can determine that any time we assign a new value to the variable, its identity changes.

Note that in Python 2.x, strings can only contain **ASCII** characters. If you require **unicode** in Python 2.x, then you will need to precede your string with a **u**. Here’s an example:

```
my_unicode_string = u"This is unicode!"
```

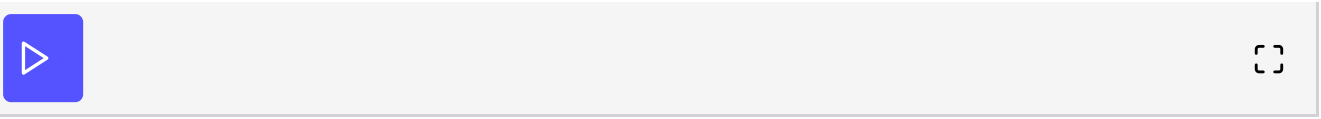
The example above doesn’t actually contain any unicode, but it should give you the general idea. In Python 3.x, all strings are unicode.

Concatenation is a big word that means to combine or add two things together. In this case, we want to know how to add two strings together. As you might suspect, this operation is very easy in Python:

```
string_one = "My dog ate "
string_two = "my homework!"
string_three = string_one + string_two

print(string_one)
```

```
print(string_two)
print(string_three)
```



The ‘+’ operator concatenates the two strings into one.