DS2000
Fall 2021
Handout: Strings

We've seen the numeric data types *int* and *float* and the arithmetic operations that we can apply to values of those data types.

Now, we're covering the *string* data type. Strings are used in Python to store letters, words, and sentences. Literals of type string can be written using either single or double quotes:

```
name = 'laney strange'
initial = "ls"
```

Some of the arithmetic operators we saw in numeric types also apply to strings, but their behavior is different. (Same operator, multiple behaviors: we call these operators **overloaded.**)

Operator	Operation	Example	Value of s
+	Concatenation	a = 'hello' b = 'world' s = a + b	'helloworld'
*	Multiple concatenation	a = 'laney' s = a * 3	'laneylaneylaney'
[ ]	Index counting from zero forwards; Negative index, counting from -1 backwards	a = 'northeastern' s = a[0] s = a[1] s = a[2] s = a[-1] s = a[-2] s = a[-3]	'n' 'o' 'r' 'n' 'r' 'e'
[start:end]	Slicing	a = 'northeastern' s = a[2:4] s = a[3:] s = a[:4]	'rt' 'theastern' 'nort'

The position of a letter within a string begins at 0. (This count-from-zero happens a lot in Python, and in most programming languages. It's weird at first, but we'll all get used to it).

If your string is 'hello', then *h* is at position 0, *e* is at position 1, and so on. Try it out in interactive mode:

```
>>> a = 'hello'
>>> a[0]          # prints 'h'
>>> a[1]          # prints 'e'
>>> a[2]          # prints 'l'
```

Strings are Like Lists

You can iterate over a string in the same way you can iterate over a list. You have two choices: by *value*, or by *position*.

Iterate by value:

```
s = 'data'
for letter in s:
    print(letter)           # prints d, then a, then t, then a
```

Iterate by position:

```
s = 'data'
for i in range(len(s)):
    print(s[i])             # prints d, then a, then t, then a
```

But... Strings are Immutable

Unlike lists, strings cannot be modified; they can only be replaced. With a list, you can do this:

```
lst = [4, 5, 6]
lst[0] = 18                 # now lst contains [18, 5, 6]
```

Strings don't do that. In the index example above, we use the index operator (`[]`) to access, but not modify, individual characters within a string. Strings actually *can't* be modified once they're created. Look what happens when we try to modify a letter in an existing string:

```
a = 'pump'
a[0] = 'j'  # try to change pump to jump.... TypeError :(
```

As a workaround, if we really do want to change pump to jump, you can create a new string that's a little variation on the original:

```
a = 'pump'
b = 'j' + a[1:]
print(b)          # prints 'jump'
print(a)          # prints 'pump'
```

When you see the assignment operator *seeming* to modify an existing string, it actually doesn't. In a couple of statements like this:

```
a = 'hello'
a = 'goodbye'
```

We're not, in fact, modifying the original string but creating a new one and putting the same label on it.

Here's what memory looks like after the first statement, `a = 'hello'`:

After the second statement, **`a = 'goodbye'`**, the original value 'hello' still exists in the same location in memory, but we've added a whole new string and put the label **`a`** on it:

More String Functions/Methods

`len(mystr)`
Returns the length of the string *mystr*.

`mystr.upper()`
`mystr.lower()`
Returns an all-uppercase or all-lowercase version of the string *mystr*. (Note that strings are immutable, so the original string *mystr* does not change. You can save the result in another string, as in **`new_str = mystr.upper()`** .)

Try it in interactive mode:

```
>>> mystr = 'hello'
>>> mystr.upper()
```

`mystr.find(s)`
Returns the starting position of the first instance of the given string *s* in the object *mystr*.

Try it in interactive mode:

```
>>> mystr = 'hello'
>>> mystr.find('el')
>>> mystr.find('o')
```

`mystr.count(s)`
Returns the number of occurrences of the given string *s* in the object *mystr*.

Try it in interactive mode:

```
>>> mystr = 'northeastern'
>>> mystr.count('e')
>>> mystr.count('n')
```

mystr.isdigit()
Returns a boolean indicating if the string *mystr* comprises one or more digits.

Try it in interactive mode:

```
>>> mystr = '1235'
>>> mystr.isdigit()
>>> mystr = '123a'
>>> mystr.isdigit()
```

mystr.replace(a, b)
Returns a string which is a copy of *mystr* but with all occurrences of *a* replaced with *b*. Because strings are immutable, this functions returns a copy and the original string *mystr* does not change.

Try it in interactive mode:

```
>>> mystr = 'hello'
>>> mystr.replace('h', 'j')
```