Strings

15-110 – Wednesday 09/23

Learning Goals

- **Index** and **slice** into strings to break them up into parts
- Use **for** loops to loop over strings by **index**
- Use **built-in string operations and methods** to solve problems

String Operations

Basic String Operations

There are useful string operations that are similar to operations we've seen before. We've already seen concatenation:

```
"Hello " + "World" # "Hello World"
```

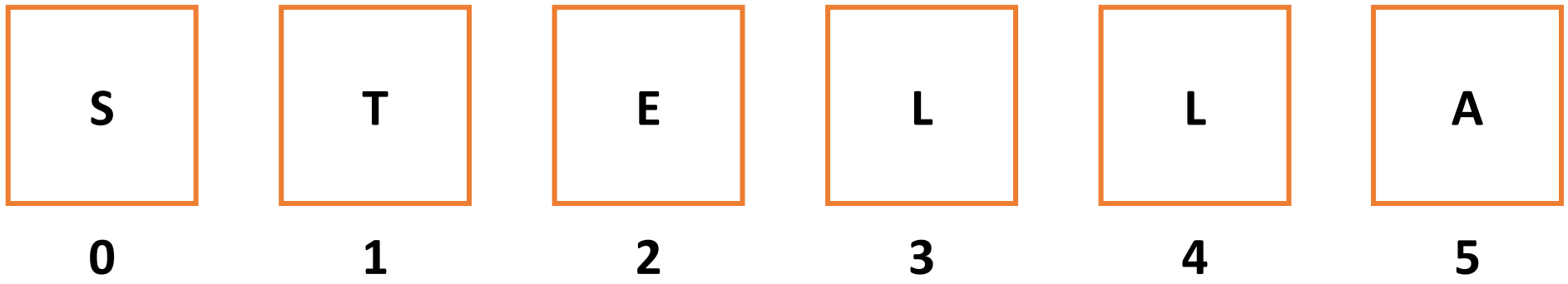
We can also use multiplication to repeat a string a number of times.

```
"Hello" * 3 # "HelloHelloHello"
```

Strings are Made of Characters

Unlike numbers and Booleans, strings can be composed of individual parts (**characters**). How can we access a specific character in a string?

First, we need to determine what each character's position is. Python assigns integer positions in order, starting with 0.



Getting Characters By Location

If we know a character's position, Python will let us access that character directly from the string. Use **square brackets** with the integer position in between to get the character.

```
s = "STELLA"
c = s[2] # "E"
```

We can get the number of characters in a string with the built-in function `len(s)`. This function will come in handy soon.

Common String Indexes

How do we get the first character in a string?

```
s[0]
```

How do we get the last character in a string?

```
s[len(s) - 1]
```

What happens if we try an index outside of the string?

```
s[len(s)] # runtime error
```

Activity: Guess the Index

Given the string `"abc123"`, what is the index of...

`"b"`?

`"1"`?

`"3"`?

Looping Over Strings

Now that we have string indexes, we can **loop** over a string by visiting each index in the string in order.

The string's first index is `0`, and the last index is `len(s) - 1`. Use `range(len(s))`.

```
s = "Hello World"
for i in range(len(s)):
    print(i, s[i])
```

String Slicing Produces a Substring

We can also get a whole substring from a string by specifying a **slice**.

Slices are exactly like ranges – they can have a **start**, an **end**, and a **step**. But slices are represented as numbers inside of **square brackets**, separated by **colons**.

```
s = "abcde"
print(s[2 : len(s) : 1])   # prints "cde"
print(s[0 : len(s)-1 : 1]) # prints "abcd"
print(s[0 : len(s) : 2])   # prints "ace"
```

String Slicing Shorthand

Like with `range()`, we don't always need to specify values for the start, end, and step. These three parts have default values: `0` for start, `len(s)` for end, and `1` for step. But the syntax to use default values looks a little different.

`s[:]` and `s[::]` are both the string itself, unchanged

`s[1:]` is the string without the first character (start is `1` instead of `0`)

`s[:len(s)-1]` is the string without the last character (end is `len(s)-1`)

`s[::3]` is every third character of the string (step is `3`)

Activity: Find the Slice

Given the string `"abcdefghij"`, what slice would we need to get the string `"cfi"`?

The `in` Operator Searches a String

When we have a type that can be broken into parts (like a string), we can use the `in` operator to check if a value occurs in the string.

```
"a" in "apple" # True
"4" in "12345" # True
"z" in "potato" # False
```

Compare Strings With ASCII Values

Python can also compare strings, like how it compares numbers. When it compares two strings, it compares the **ASCII values** of each character in order.

Because the lowercase letters and uppercase letters are listed in order in the ASCII table, we can compare lower-to-lower and upper-to-upper **lexicographically**, in the order they'd appear in the dictionary. But that won't work if we compare lowercase to uppercase letters.

```
"hello" > "goodbye" # True, 'h' larger than 'g'
"book" < "boot" # True, 'boo' equal and 'k' smaller than 't'
"APPLE" < "BANANA" # True, 'A' smaller than 'B'
"ZEBRA" > "aardvark" # False - lowercase letters are larger
```

Other String Operators

We can directly translate characters to ASCII in Python. The function `ord(c)` returns the ASCII number of a character, and `chr(x)` turns an integer into its ASCII character.

```
ord("k") # 107
chr(106) # "j"
```

Finally, there are a few characters we need to treat specially in Python: the enter character (**newline**) and the tab character (**tab**). We can't type these directly into a string, so we'll use a shorthand instead:

```
print("ABC\nDEF") # newline, or pressing enter/return
print("ABC\tDEF") # tab
```

The `\` character is a special character that indicates an **escape sequence**. It is modified by the letter that follows it. These two symbols are treated as a single character by the interpreter.

String Methods

String Methods Are Called Differently

String built-in methods work differently from built-in functions. Instead of writing:

```
isdigit(s)
```

we have to write:

```
s.isdigit()
```

This tells Python to call the built-in string method `isdigit()` **on the string** `s`. It will then return a result normally.

This is how our Tkinter methods work too! `create_rectangle` is called **on** the `canvas`.

Don't Memorize- Use the API!

There is a whole library of built-in string methods that have already been written; you can find them at

docs.python.org/3.8/library/stdtypes.html#string-methods

We're about to go over a whole lot of methods, and it will be hard to memorize all of them. Instead, **use the Python documentation** to look for the name of a function that you know probably exists.

If you can remember which basic actions have already been written, you can always look up the name and parameters when you need them.

Some String Methods Return Information

Some string methods return information about the string.

`s.isdigit()`, `s.islower()`, and `s.isupper()` return `True` if the string is all-digits, all-lowercase, or all-uppercase, respectively.

`s.count(c)` returns the number of times the character `c` occurs in `s`.

`s.find(c)` returns the index of the character `c` in `s`, or `-1` if it doesn't occur in `s`.

Example: Checking a String

As an example of how to use string methods, let's write a function that returns whether a string is composed entirely of unique alphabetic characters. It should return False if there are any numeric characters or any repeated characters.

```
def isUniqueWord(s):
    for i in range(len(s)):
        if s.count(s[i]) > 1:
            return False
        elif s[i].isdigit():
            return False
    return True
```

Some String Methods Create New Strings

Other string methods return a new string based on the original.

`s.lower()` and `s.upper()` return a new string that is like the original, but all-lowercase or all-uppercase, respectively.

`s.replace(a, b)` returns a new string where all instances of the string `a` have been replaced with the string `b`.

`s.strip()` returns a new string with any whitespace (spaces, tabs, newlines) at the beginning and the end of `s` removed.

Example: Making New Strings

We can use these new methods to make a silly password-generating function.

```
def makePassword(phrase):
    phrase = phrase.strip()
    phrase = phrase.lower()
    phrase = phrase.replace("a", "@")
    phrase = phrase.replace("o", "0")
    return phrase
```

Activity: `getFirstName(fullName)`

You do: write the function `getFirstName(fullName)`, which takes a string holding someone's full name (in the format `"Farnam Jahanian"`) and returns just their first name. You can assume the first name will either be one word or will be hyphenated (like `"Soo-Hyun Kim"`).

You'll want to use a **method** and an **operation** in order to isolate the first name from the rest of the string.

Next Lecture: Unit 1 Review

On Friday, we'll talk about how all the things we've learned in Weeks 1-4 connect together, and review any concepts you're still unsure about.

Fill out this poll to tell us which concepts you'd most like to review: http://bit.ly/110-unit1

Learning Goals

- **Index** and **slice** into strings to break them up into parts
- Use **for** loops to loop over strings by **index**
- Use **built-in string operations and methods** to solve problems
- **Feedback: https://bit.ly/110-feedback**