String Indexing, Slicing, and Looping

15-110 – Wednesday 02/08

Learning Goals

- **Index** and **slice** into strings to break them up into parts

- Use for loops to loop over strings by **index**

How Can We Process Text Data?

So far, we've mainly written programs that deal with numbers.

We can use strings in programs, but there isn't much we can do with them so far. But real text data is **modular** – you can break it up into sentences, words, letters.

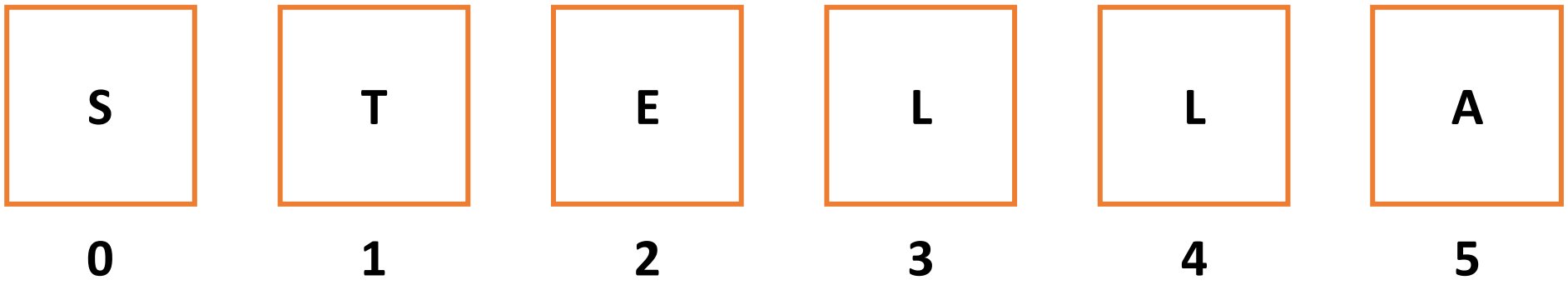
How can we write code that treats text like a **set of data** instead of a single item?

Indexing and Slicing

Strings are Made of Characters

Unlike numbers and Booleans, Python can break strings down into 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. This is called **indexing**.

```
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 prove useful 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...

`"a"`?

`"c"`?

`"3"`?

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(value)` 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 (we can remove the second colon when the step is `1`)

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

`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"`?

Example: Extract Information from Text

Let's assume we have a variable `text` that holds a greeting: `"Hello NAME"`. We want to extract just the name from the text.

We can use string slicing! Start the slice at the location of the **first character** of the name.

```
text = "Hello Jonathan"
name = text[len("Hello "):] # "Jonathan"
```

More String Things

Special Characters

Most characters that appear in text can be typed directly into strings, but some are more difficult to work with. These include the enter character (**newline**) and the tab character (**tab**). To represent these characters in a string, we'll use a shorthand:

```
"ABC\nDEF" # '\n' = newline, or pressing enter/return
"ABC\tDEF" # '\t' = 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.

Haiku by Carol A. Coiffait

Triple Quotes

Early in the semester we showed how you can use triple-quotes to create multi-line comments. You can also use them to create multi-line strings, and you can type special characters into those strings directly, without using escape sequences!

```
s = """This Autumn midnight
Orion's at my window
shouting for his dog."""
```

is equivalent to:

```
s = "This Autumn midnight\nOrion's at my window\nshouting for his dog."
```

Membership Checks

We can now introduce a new operator called `in` (and its opposite `not in`) to see whether an individual character or substring occurs in a string. This returns a Boolean. This is very handy when you want to check whether something occurs in a piece of text!

```
"e" in "Hello" # True
"W" in "CRAZY" # False
"seven" in "Four score and seven years ago" # True
"wow" not in "That's impressive" # True
```

ASCII Functions

Finally, there are two built-in functions that will let us find the ASCII values of characters when we need them. `ord(c)` lets you find the ASCII value of a given one-character string, and `chr(x)` returns the character associated with the given ASCII value.

```
ord("K") # 75
chr(76) # "L"
```

Looping with Strings

Looping Over Strings

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

If the string is `s`, the string's first index is `0` and the last index is `len(s)-1`. Use `range(len(s))` to visit all possible indexes.

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

Algorithmic Thinking with Strings

If you need to solve a problem that involves doing something with every character in a string, use a for loop over that string.

For example – how do we make a version of a string that doesn't include any spaces? Make a new string by checking each character and only add each one if it isn't a space.

```
s = "Wow! This is so exciting!"
result = ""
for i in range(len(s)):
    if s[i] != " ": # note the space between the quotes!
        result = result + s[i]
print(result) # "Wow!Thisissoexciting!"
```

For Loop Indexes are Flexible

For loops may seem straightforward when the loop control variable refers to each index in the string. But we can get more creative with **what** the variable is used for when necessary!

For example – how would you check whether a string is a palindrome (the same front-to-back as it is back-to-front)? Use the variable as the front index **and the back index offset**.

```
def isPalindrome(s):
    for i in range(len(s)):
        front = s[i]
        back = s[len(s) - 1 - i]
        if front != back:
            return False
    return True
```

Activity: Coding with Strings

You might be able to recognize a person by the types of punctuation they use in text messages. Maybe one friend loves exclamation points while another friend never uses them.

You do: write a function `getPunctuationFrequency(text, punc)` that takes a text message (a string) and a punctuation character (another string) and returns the frequency of how often that character appears in the text compared to other characters - the number of times it appears over the total number of characters.

For example, `getPunctuationFrequency("That's so exciting!! Good for you man!", "!")` would return ~0.079, because exclamation marks form 3/38 = ~0.079 as a ratio of the characters in the text.

[if time] Try it with real data!

We can try running our analysis function on real texts!

Websites like Project Gutenberg make the text of books available online for free. You can copy that text into a string, then run that string through the function.

Running the function through some popular classic fiction and trying out a few different types of punctuation already gleans interesting results. For example, the character **.** takes up 1.15% of text in *The Great Gatsby* compared to 0.82% in *Pride and Prejudice*; on the other hand, the character **;** takes up only 0.03% of text in *The Great Gatsby* compared to 0.21% of text in *Pride and Prejudice*.

Combining these frequencies together can give us an interesting map of the writing styles of different authors!

Learning Goals

- **Index** and **slice** into strings to break them up into parts

- Use for loops to loop over strings by **index**

Sidebar: When do we use `len(s)` vs. `len(s)-1`?

It can be hard to tell when to use `len(s)` vs. `len(s)-1`. What do these two expressions really mean?

`len(s)` is the **length** of the string, the number of characters it contains. Because the first index of a string is 0, not 1, `s[len(s)]` returns an error.

> On the other hand, `s[len(word):]` creates a slice that starts exactly `len(word)` characters into the string, which could be useful.

`len(s)-1` is the **last index** of the string. `s[len(s)-1]` returns the last character of a string.