# Strings

15-110 – Wednesday 02/24

# Announcements

- **Quiz1 today!**
  - Make sure to take the quiz by midnight EST
  - No office hours on quiz days

# Learning Goals

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

# Unit 2 Overview

# Unit 2: Data Structures and Efficiency

**Data Structures:** things we use while programming to organize multiple pieces of data in different ways.

**Efficiency:** the study of how to design algorithms that run quickly, by minimizing the number of actions taken.

These concepts are **connected**, as we often design data structures so that specific tasks have efficient algorithms.

# Unit 2 Topic Breakdown

**Data Structures:** strings, lists, dictionaries, trees, graphs

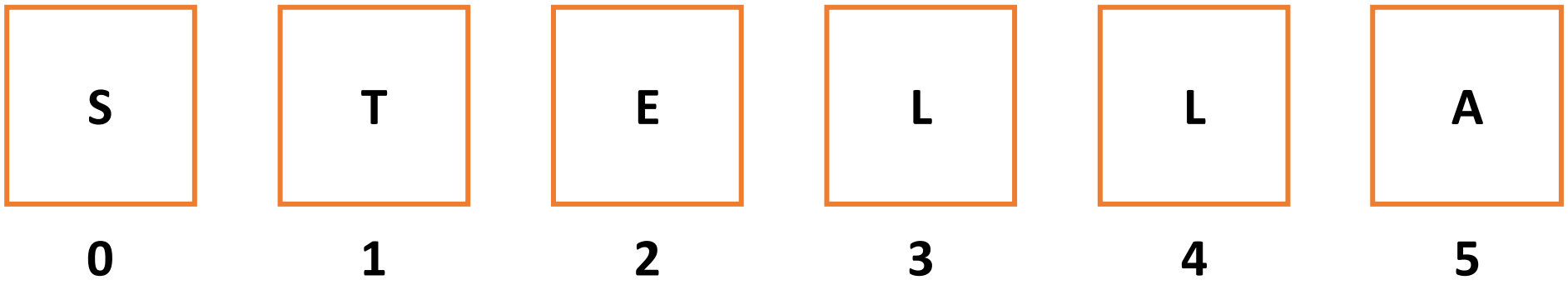
**Efficiency:** search algorithms, Big-O, tractability

# Indexing and Slicing

# Strings are Made of Characters

Unlike numbers and Booleans, strings can be broken 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 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...

`"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(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`)

`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"`?

# 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.

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])
```

# 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 count the number of exclamation points in a string?

```
s = "Wow!! This is so! exciting!!!"
count = 0
for i in range(len(s)):
    if s[i] == "!":
        count = count + 1
print(count)
```

# 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)):
        if s[i] != s[len(s) - 1 - i]:
            return False
    return True
```

# Other String Operators

# 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"
```

# 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 inside the whole.

```
"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
```

# ASCII Conversion Functions

We can directly translate characters to ASCII in Python. The built-in 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"
```

**You do:** logically, what should `chr(ord("a") + 1)` produce?

# String Methods

# String Methods Are Called Differently

String built-in functions (and data structure functions in general) work differently from built-in functions. Instead of writing:

```
isdigit(s)
```

write:

```
s.isdigit()
```

This tells Python to call the built-in string function `isdigit()` **on the string** `s`. It will then return a result normally. We call this kind of function a **method**, because it belongs to a **data structure**.

This is how our Tkinter methods work too! `create_rectangle` is called **on** `canvas`, which is a data structure.

# 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 potentially useful 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`.

```
s = "hello"

s.isdigit() # False
s.islower() # True
"OK".isupper() # True

s.count("l") # 2

s.find("o") # 4
```

# Example: Checking a String

As an example of how to use string methods, let's write a function that returns whether or not a string holds a capitalized name. The first letter of the name must be uppercase and the rest must be lowercase.

```
def formalName(s):
    return s[0].isupper() and s[1:].islower()
```

# Some String Methods Create New Strings

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

```
s = "Hello"
```

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

```
a = s.lower() # a = "hello"
b = s.upper() # b = "HELLO"
```

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

```
c = s.replace("l", "y")
# c = "Heyyo"
```

# Example: Making New Strings

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

```
def makePassword(phrase):
    phrase2 = phrase.lower()
    phrase3 = phrase2.replace("a", "@").replace("o", "0")
    return phrase3
```

# Activity: `getFirstName(fullName)`

**You do:** write the function `getFirstName(fullName)`, which takes a string holding a two-word full name (in the format `"Farnam Jahanian"`) and returns just the 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.

# Learning Goals

- **Index** and **slice** into strings to break them up into parts
- Use for loops to loop over strings by **index**
- Use **string operations and methods** to solve problems
- Feedback: http://bit.ly/110-s21-feedback