#5: Data Representation

SAMS PROGRAMMING C

Review from Last Week

Combine blocks of code to create complex programs

Use the problem-solving process to develop solutions to new problems

Use testing and debugging to verify that our programs work

Today's Learning Goals

Use **indexing and slicing** on strings while writing functions

Utilize **lists** as data structures when writing programs

Understand the difference between **mutable** and **immutable** datatypes

Storing Multiple Values

Let's say we want to keep track of the first 10 prime numbers. Right now, to do that, we need to make ten different variables.

```
prime1 = 2

prime2 = 3

prime3 = 5

prime4 = 7

prime5 = 11

prime6 = 13

prime7 = 17

prime8 = 19

prime9 = 23

prime10 = 29
```

This feels similar to our argument about loops...

Storing multiple values in a list

Instead, let's condense those values down into a single object which we can **iterate over** to get each individual prime

```
prime1to10 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

We call this data structure a **list**!

List syntax

Lists are denoted by square brackets, where values go inside the brackets (separated by commas). Lists can hold as many values as they need to, or no values at all. Here's an example of an empty list:

```
lst = [ ]
```

And a list of strings:

```
strLst = ["Hello", "World!"]
```

List iteration

We can loop over lists the same way we loop over strings, by accessing each element in order (like how we access each character in order):

```
lst = [2, 4, 6, 8]
for item in lst:
    print(item)
```

But we can also do a lot more...

List Indexing

We can also **index** into a list to get a specific value. Each element of the list is assigned an index based on its position, **starting with 0**. As computer scientists, we always start counting at 0!

Example: in the list `["a", "b", "c"]`...

Index 0: "a"

Index 1: "b"

Index 2: "c"

List/String Indexing

To index into a list, we put square brackets after the list with the index number inside.

```
lst = ["a", "b", "c"]
print(lst[1]) # "b"
```

And guess what – **we can do this with strings, too!** After all, a string is just a 'list' of characters.

```
s = "abc"
print(s[1]) # "b"
```

Indexing examples

Question: how do we get the first character of a string `s`?

Answer: `s[0]`

Question: how do we get the last item in a list `lst`?

Answer: `lst[len(lst)-1]` OR `lst[-1]`

That's right- we can index with negative numbers! This just wraps around the back of the list/string.

Indexing Quiz

Given the list `["a", "b", "c", 1, 2, 3]`, what is the index of...

"b"?

1?

3?

List/String Slicing

We can also get a whole subset of a list or string by specifying a **slice**.

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

```
lst = [0, 2, 4, 6, 8]
print(lst[2:len(lst):1]) # print [4, 6, 8]
print(lst[0:len(lst)-1:1]) # prints [0, 2, 4, 6]
print(lst[0:len(lst):2]) # prints [0, 4, 8]
```

List/String Slicing Shorthand

We can leave the start, end, and step entries blank if we're using the default values.

`lst[:]` and `lst[::]` are both the list itself, unchanged

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

`lst[:-1]` is the list without the last character

`s[::3]` is the string with every third character

Slicing Example

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

Lists vs. Strings: types

So far, lists and strings seem fairly similar. However, they do have two major differences.

First: strings can only hold characters. Lists, on the other hand, can hold **any** type of data. They can even mix those types up!

```
lst = [0, "a", True, None]
print(lst)
```

Lists vs. Strings: mutability

Second, lists can be changed directly (they are **mutable**), while strings are static and cannot be changed (they are **immutable**). For lists, this means we can use indexing and/or slicing to update the lists as needed:

```
lst = ["Carnegie", "Mellon", "University"]
lst[2] = "Rocks!"
print(lst)
lst[-1:] = ["My!", "Socks!"]
print(lst)
```

String are Immutable

If we try to change the characters inside a string directly, we'll have a bad time

```
s = "Carnegie Mellon"
s[1] = "@" # uh oh!
print(s)
```

However, we can still overwrite the string as a whole by changing the variable's value. This makes a brand new string.

```
s = "Carnegie Mellon"
s = s[0] + "@" + s[2:] # this is okay!
print(s)
```

New operation: in

When we have an iterable type (like a list or a string), we can use the **in** operator to check if a value occurs in the list/string.

```
"a" in "apple" # True!
4 in [1,2,3,4,5] # True!
"z" in "potato" # False!
```

For strings only, we can actually check if several characters in a row appear in the string...

```
"erdu" in "superduper" # True!
```

String Example

Example: write the function `longestCommonSubstring(s, t)` that takes two strings, `s` and `t`, and returns the longest substring which both strings have in common.

Useful String Functions

There is a whole library of string functions that have already been written: you can find them at

https://docs.python.org/3.6/library/stdtypes.html#string-methods

We'll go over a few of these in a minute..

There are also some useful built-in string constants, found at:

https://docs.python.org/3.6/library/string.html#string-constants

To use these, you need to import string

```
import string
s = "A"
print(s in string.ascii_letters)
```

Using string methods

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

```
isdigit(s)
```

we have to write:

```
s.isdigit()
```

Also: **because strings are immutable, these methods don't change the string!** They return a new string, so you need to capture the result and use it.

String functions: find

If we want to determine where a character occurs in a string, we use the built-in method **find**. This returns the index of the first appearance of the character, or -1 if it isn't in the string.

```
s = "abcde"

print(s.find("c")) # 2

print(s.find("z")) # -1
```

String functions: replace

replace works the same way it does in the text editor; it replaces all occurrences of the first parameter with the second one.

```
s = "football gooooooal!"

print(s.replace("oo", "8")) # f8tball g888al!"

print(s) # remember, s itself doesn't change!
```

String functions: split

If you need to break a string into multiple parts, use **split**. The function takes a delimiter (the separating character(s)) and returns a list of the separate string parts. The delimiter can be anything you want...

```
s = "Monday,Tuesday,Thursday"

print(s.split(",")) # ["Monday", "Tuesday", "Thursday"]

s = s.replace(",", "!")

print(s) # "Monday!Tuesday!Thursday"

print(s.split("day!")) # ["Mon", "Tues", "Thursday"]
```

String functions: strip

If you need to remove extra whitespace from the beginning or end of a string, you can use **strip**. Whitespace includes spaces, newlines, and tabs.

```
s = "  Testing: 1, 2, 3    "
print("(", s.strip(), ")", sep="") # "(Testing: 1, 2, 3)"
```

List Methods

There is a whole library of list methods that have already been written: you can find them at

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

We'll go over a few of these now!

Note that list functions are also called directly on the list:

```
lst = [1, 2, 3]

lst.append(4)
```

List functions: append, extend, and insert

When we want to add more elements into a list, we can use several functions. First, **append** lets us add a single element to the end of the list. Second, **extend** lets us add a list of elements to the end of the list. Finally, **insert** lets us place a specific element into a specific location, moving the rest of the elements back.

```
lst = [1, 2, 3]

lst.append(4)

print(lst) # [1, 2, 3, 4]

lst.extend([5, 6])

print(lst) # [1, 2, 3, 4, 5, 6]

lst.insert(0, -5)

print(lst) # [-5, 1, 2, 3, 4, 5, 6]
```

List functions: pop and remove

When you want to remove something from a list, you can use one of two different functions. **pop** lets you remove the element at the provided index (or the element at the end if no index is provided); **remove** gets rid of the first occurrence of the provided element.

```
lst = ["a", "b", "c", "d", "e"]

lst.pop(1)

print(lst) # ["a", "c", "d", "e"]

lst.remove("d")

print(lst) # ["a", "c", "e"]
```

List functions: index

To find the location of an element in a list, use the function **index**. NOTE: index will crash if the element is not in the list! Check with 'in' first to make sure the element is in there.

```
lst = [ "a", "b", "c" ]
print(lst.index("b")) # 1
```

List functions: count

To determine how many times an element occurs in a list, use the **count** function!

```
lst = [ 1, 2, 3, 1, 2, 1]
print(lst.count(1)) # 3
```

Example

Example: write the function `solvesCryptarithm(puzzle, solution)`, which returns `True` if the string solution represents a valid solution to the puzzle string, and `False` otherwise.

Read more about cryptarithms and the problem here:

https://www.cs.cmu.edu/~112/notes/colab4.html

Today's Learning Goals

Use **indexing and slicing** on strings while writing functions

Utilize **lists** as data structures when writing programs

Understand the difference between **mutable** and **immutable** datatypes