#3-1: Strings and Lists I

CS SCHOLARS – PROGRAMMING

Hw2 Recap

When reading code with nested conditionals, note that you should read **every level of conditional**, not just the most recent one! All relevant branches must pass to reach the desired state.

When building test cases for edge cases, it's best to test BOTH sides of the edge, not just one.

When working with graphics, note that our new starter code puts the tkinter setup code **in a function**. You can just call that test function to run your code; you don't need to set it up yourself at the top level!

Don't add additional parameters to function definitions! You should be able to solve the problem with only the provided input.

Learning Goals

Read and write code using **strings** and **lists**

Index and **slice** into strings/lists to break them up into parts

Use **for loops** to loop over strings/lists by **index**

String and List Syntax

String Syntax

We introduced **strings** as a core datatype in Week 1. Strings are defined as text inside of quotes.

```
s = "Hi everyone!"
```

We can concatenate strings together, and we can also repeat strings with multiplication.

```
"ABC" + "DEF" # "ABCDEF"
"HA" * 3 # "HAHAHA"
```

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."
```

Strings are Collections of Characters

Unlike numbers and Booleans, strings can be broken down into individual parts (**characters**). We say that a string is a **sequence** of characters. This is a core part of how they're represented in Python.

We can use a special operator called `in` to see whether an individual part occurs in the string. This returns a Boolean.

```
"e" in "Hello" # True
"W" in "CRAZY" # False
```

What if we want to store a sequence of some other datatype in a single value?

Lists are Containers for Data

A **list** is a new data type that holds a sequence of data values.

Example: a sign-in sheet for a class.

Sign In Here
0. Elena
1. Max
2. Eduardo
3. Iyla
4. Ayaan

Lists make it possible for us to assemble and analyze a collection of data **using only one variable**.

List Syntax

We use **square brackets** to set up a list in Python.

```
a = [ ] # empty list
b = [ "uno", "dos", "tres" ] # list with three strings
c = [ 1, "dance", 4.5 ] # lists can have mixed types
```

Basic List Operations

Lists share most of their basic operations with strings.

```
a = [ 1, 2 ] + [ 3, 4 ] # concatenation – [ 1, 2, 3, 4]
b = [ "a", "b" ] * 2 # repetition – [ "a", "b", "a", "b" ]
d = 4 in [ "a", "b", 1, 2 ] # membership – False
```

Activity: Evaluate the Code

You do: what will each of the following code snippets evaluate to?

```
[ 5 ] * 3
```

```
"A" in "easy"
```

```
[ 1 ] + [ ] + [ "B" ]
```

Indexing and Slicing

Strings are Made of Characters

While problem solving, we'll often want to access the individual parts of strings, lists, and other sequences. For example, how can we access a specific character in a string?

STELLA

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

S	**T**	**E**	**L**	**L**	**A**
0	**1**	**2**	**3**	**4**	**5**

Getting Values 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"
```

The same thing works with lists!

```
lst = [ 15, 110 ]
lst[1] # 110
```

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

Common Indexes

How do we get the first character in a string?

```
s[0]
```

How do we get the last element in a list?

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

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

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

Activity: Guess the Index

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

`"a"`?

`"c"`?

`"3"`?

Slicing Produces a Substring/Subset

We can also get a whole substring from a string or subset from a list 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"
s[2:len(s):1]   # "cde"
s[0:len(s)-1:1] # "abcd"
s[0:len(s):2]   # "ace"
```

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(var)` for end, and `1` for step. But the syntax to use default values looks a little different.

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

`lst[1:]` is the list without the first element (start is `1`)

`lst[:len(lst)-1]` is the list without the last element (end is `len(lst)-1`)

`lst[::3]` is every third element of the list (step is `3`)

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"
```

Activity: Find the Slice

You do: Given the list

```
lst = [ 2, 4, "t", "r", 3.4, 8.1, 23, "okay", 110, "woo" ]
```

what slice would we need to get the sublist `[ "t", 8.1, 110 ]`?

List Index Assignment

Using indexes, we can do something new and cool – we can directly change the values inside a list!

If you assign a list index to a new value, like how you would set a variable to a value, it will change the value in the list permanently.

```
lst = [ "a", "b", "c" ]
lst[1] = "foo"
lst # [ "a", "foo", "c" ]
```

String Index Assignment Doesn't Work

However, index assignment **only works for lists**. You can't use index assignment on strings; you'll get a runtime error. To modify a string, you need to assign the whole variable to a new value instead.

```
s = "abc"
s[1] = "z" # TypeError
s = s[:1] + "z" + s[2:] # s now holds "azc"
```

This is because of how strings and lists are stored in computer memory. We call lists **mutable** and strings **immutable**. Mutable values can be modified directly; immutable values cannot.

Learn more: https://www.cs.cmu.edu/~15110-s23/slides/week5-1-references.pdf

Looping with Sequences

Looping Over Sequence Indexes

Now that we have indexes and slices, we can **loop** over the characters in a string or the elements in a list by visiting each index in the value in order.

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

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

lst = [ "What", "a", "nice", "day!" ]
for i in range(len(lst)):
    print(i, lst[i])
```

Example: Looping over Sequences

We can develop algorithms using loops over strings and lists whenever we need to visit each index in the string/list to solve a problem. For example, the following loop sums all the values in `prices`.

```
prices = [ 5.50, 3, 2.75 ]
total = 0
for i in range(len(prices)):
    total = total + prices[i]
print(total)
```

Activity: Code Reading

What will the following code print?

```
s = "abcdefg"
count = 0
for i in range(len(s)):
    if i % 3 == 0:
        print(s[i])
        count = count + 1
print(count)
```

Example: Building New Sequences

We can also build new strings/lists using concatenation within a loop!

```
lst = []
for i in range(1, 11):
    lst = lst + [i] # concatenate to end
print(lst)
```

Example: Building New Sequences

You can even use an existing string/list to create a new string/list. For example, this code makes a backwards version of the given string.

```
s = "awesome"
result = ""
for i in range(len(s)):
    result = s[i] + result # add s[i] to front!
print(result)
```

Activity: Double Numbers

Write a bit of code that takes a list of numbers and creates a **new** list of numbers where each number has been doubled.

For example, given `[1, 2, 3]`, the code should produce `[2, 4, 6]`.

Algorithmic Thinking with Loops

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

Read and write code using **strings** and **lists**

Index and **slice** into strings/lists to break them up into parts

Use **for loops** to loop over strings/lists by **index**