#3-2: Strings and Lists

CS SCHOLARS – PROGRAMMING

Hw2 Recap

Hw2 feedback is released!

Recap on: Variable Scope, Function Tracing, and Conditional Tracing

Note on Advanced content - Concurrency

Hw2 Recap on Variable Scope

The following code prints all the numbers from 1 to 50 and whether or not they're prime. What are all the global and local variables in the code?

```
def isPrime(num):
    if num < 2:
        return False
    factor = 2
    while factor < num
        if num % factor == 0:
            return False
        factor = factor + 1
    return True

for x in range(50):
    print(x, isPrime(x))
```

Hw2 Recap on Function Tracing

Let's go back to our original example. What are all the function call names, argument values, and returned values that happen in this code?

```
def outer(x):
    y = x / 2
    print("outer y:", y)
    return inner(y) + 3

def inner(x):
    y = x + 1
    print("inner y:", y)
    return y

print(outer(4))
```

Hw2 Recap on Conditionals

Say we want to print the secret message in this code. Which conditions must be met?

```
if x % 2 == 0:
    print("nope")
elif x > 20:
    if y < 99:
        print("nope")
    else:
        if x < 40:
            print("SECRET MESSAGE!")
```

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** or **component**

Use string/list **methods** to call functions directly on values

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"
```

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.

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
```

Example: Using Lists

We could write a bit of code to construct a list of the numbers from 1 to 10.

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

Activity: Evaluate the Code

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

```
[ 5 ] * 3
```

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

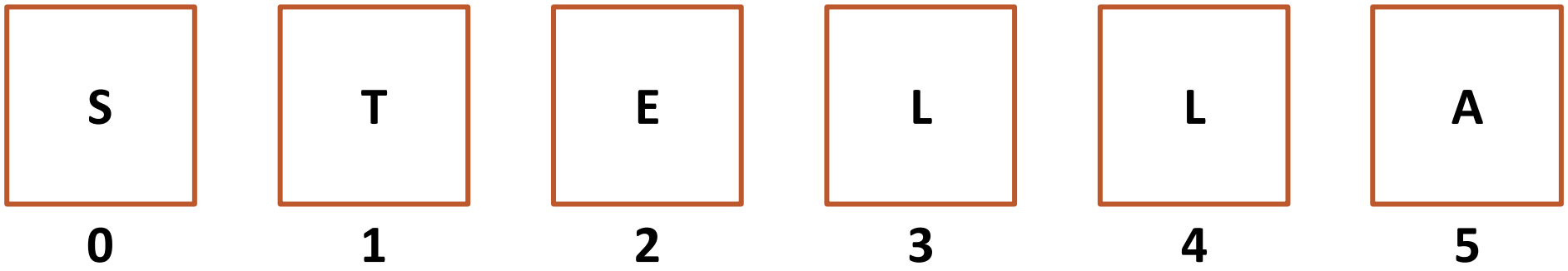
```
[ 1 ] + [ ] + [ "B" ]
```

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?

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



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 character (end is `len(lst)-1`)

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

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:]
```

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-s22/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 Lists

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`.

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

Looping Over Sequences Directly

If we don't care about where values are located in a sequence, we don't need to use a `range` in the `for` loop. We can loop over the parts of a sequence directly by providing the value instead of a `range`.

```
for <itemVariable> in <sequenceValue>:
    <itemActionBody>
```

For example, if we run the following code, it will print out each string in the list with an exclamation point after it.

```
wordlist = [ "Hello", "World" ]
for word in wordList:
    print(word + "!")
```

Example: Looping over Strings

Another example – how do we count the number of exclamation points in a string? We don't need the indexes, so we can loop over the string directly.

```
s = "Wow!! This is so! exciting!!!"
count = 0
for c in s:
    if c == "!":
        count = count + 1
print(count) # 6
```

Choosing Loops

How do you decide whether to loop over a range or loop over the value directly? Think about whether you need to know **where** the parts are located in the sequence.

For example – how would you check whether a string is a palindrome (the same front-to-back as it is back-to-front)? Use a range so that you can use the index variable as both 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
```

Activity: `findMax(lst)`

Write a function `findMax(lst)` which takes a list of numbers and returns the largest number in the list.

Hint: consider what **variables** you'll need to keep track of, and what type of **loop** you should use.

Methods

Methods Are Called Differently

Most string and list built-in functions (and data structure functions in general) work differently from other 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 and list methods that have already been written; you can find them at

docs.python.org/3/library/stdtypes.html#string-methods

and

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

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 Methods Return Information

Some methods return information about the value.

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

`s.count(x)` and `lst.count(x)` return the number of times the subpart `x` occurs in `s` or `lst`.

`s.index(x)` and `lst.index(x)` return the index of the subpart `x` in `s` or `lst`, or raise an error if it doesn't occur in the value.

```
s = "hello"
lst = [10, 20, 30, 40, 50]

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

s.count("l") # 2
lst.count(20) # 1

s.index("o") # 4
lst.index(5) # ValueError!
```

Example: Checking a String

As an example of how to use 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()
```

Activity: Evaluate the Code

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

```
"Yay".islower()
```

```
lst = [4, 8, 10, 8, 6, 4]
lst.count(4)
lst.index(4)
```

Some Methods Create New Values

Other methods return a new value 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.split(c)` returns a list that has split up the string based on the separator character, `c`.

```
s = "Hello"

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

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

d = "one-two-three".split("-")
# d = [ "one", "two", "three" ]
```

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 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/or an **operation** in order to isolate the first name from the rest of the string.

Some Methods Change the Value

Finally, there are some methods that let us **change the value itself**, without reassigning the variable. We call these **destructive** methods.

`lst.append(item)` destructively adds a new item to the end of a list.

`lst.remove(item)` destructively removes the given item from a list once.

`lst.sort()` destructively sorts the list by comparing all the elements.

Note that the function calls do not return a new list; the original list is changed instead. That means we typically call the method **by itself** and do not assign the result to a variable.

```
lst = [ 1, 2, "a" ]


lst.append("b")

# lst = [ 1, 2, "a", "b" ]


lst.remove("a")

# lst = [ 1, 2, "b" ]


lst = [ 40, 2, 13, 7 ]

lst.sort()

# lst = [ 2, 7, 13, 40]
```

Example: `getFactors(n)`

Let's write a function that takes an integer and returns a list of all the factors of that integer.

```
def getFactors(n):
    factors = [ ]
    for num in range(1, n+1): # n is a possible factor
        if n % num == 0:
            factors.append(num)
    return factors
```

Activity: `longWordsOnly(words)`

Write a function `longWordsOnly(words)` that takes a list of words (strings) and returns a new list that only contains words that were longer than 4 characters.

For example, `longWordsOnly(["What", "a", "fabulous", "day", "it", "is", "today"])` would return `["fabulous", "today"]`.

Try using the **append** method to set up the new list instead of using list concatenation!

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** or **component**

Use string/list **methods** to call functions directly on values