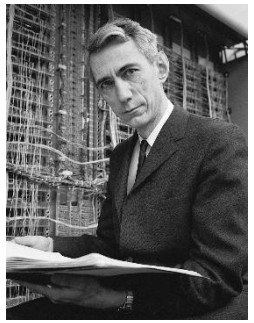
Bonus Slide

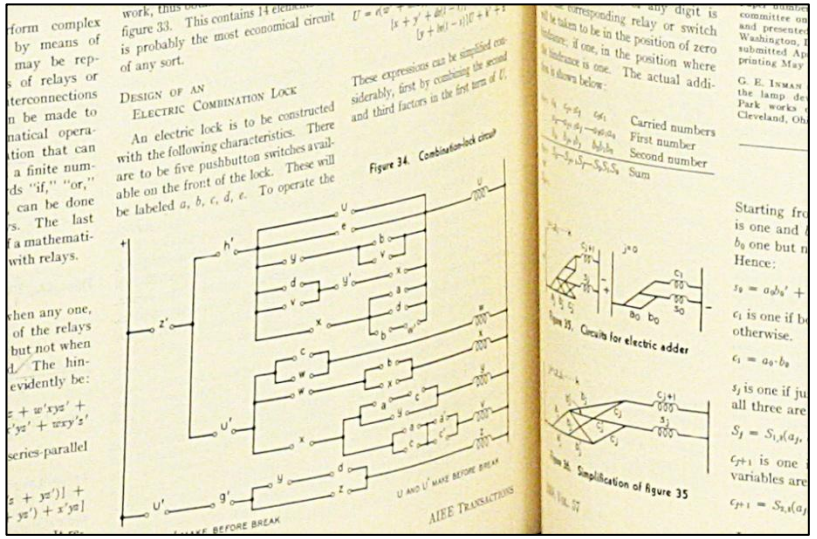
Circuits from a Master's Thesis



In 1937, Claude Shannon published his Master's Thesis, "A Symbolic Analysis of Relay and Switching Circuits". This was the first time Boolean logic was translated into a physical format with electronics.

This work became the foundation of circuit design and made it possible to design the computers we know today.

Shannon also invented the full adder as an example in his paper!



Using the method of symmetric functions, and shifting down for s_j gives the circuits of Figure 35. Eliminating superfluous elements we arrive at Figure 36.

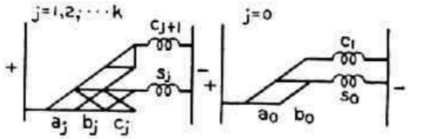


Figure 35. Circuits for electric adder

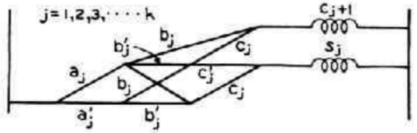


Figure 36. Simplification of figure 35

Unit 1: Fundamentals of Programming

String and Lists

15-110 – Wednesday 09/16

Quizlet2

Learning Goals

- Identify common properties of **strings** and **lists**
- **Index** and **slice** into strings and lists to break them up into parts
- Use for loops to loop over strings and lists by **index**
- Trace code using **strings**, **1D lists,** and **2D lists**

Sequenced Structures Hold Data

Data often comes in sequences. A sentence is a sequence of characters. Your shopping list is a sequence of items.

To represent many things in a given order, we use **ordered collections**.

Strings, which we studied in the first week of the semester, and **lists**, which we're introducing for the first time today, are both ordered collections of data.

That means each element has a position and we can iterate through the structure one element at a time. We'll see how to do that today.

Lists and Operations

Lists are Containers for Data

A **list** is a data structure that holds an ordered collection 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
```

Concatenation and Repetition

We previously saw how to use concatenation and repetition on strings.

```
"ABC" + "DEF" # concatenation - "ABCDEF"
"HA" * 3 # repetition - "HAHAHA"
```

Lists share most of their core operations with strings. You can use concatenation and repetition on these as well.

```
[ 1, 2 ] + [ 3, 4 ] # concatenation – [ 1, 2, 3, 4]
[ "a", "b" ] * 2 # repetition– [ "a", "b", "a", "b" ]
```

New Operator: `in`

The operator `in` (and its opposite `not in`) checks whether a character or substring occurs in a string. This evaluates to 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
"wow" not in "That's impressive" # True
```

And you guessed it, this works for lists too!

```
"c" in ["a", "b", "d"] # membership – False
"c" not in [1, 2, 3] # membership - True
```

Activity: Evaluate the Code

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

```
[ 5 ] * 3
```

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

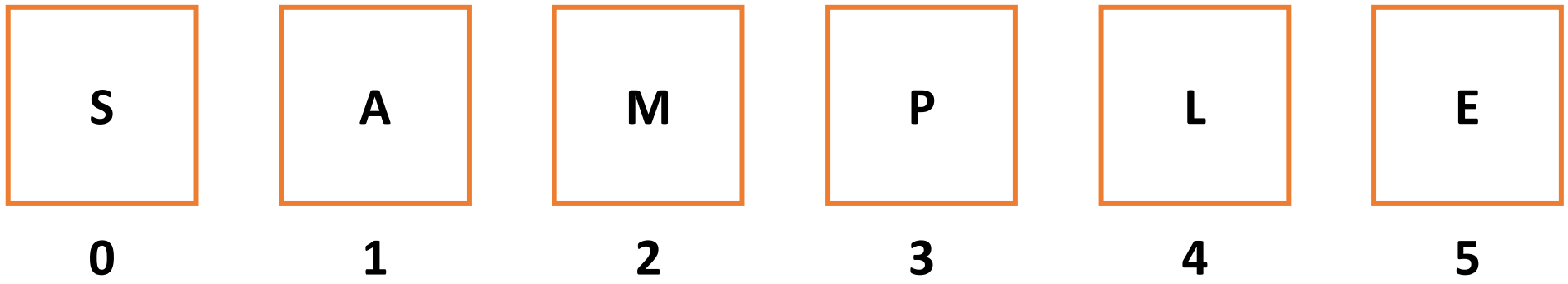
```
"wow" in [ "that's", "amazing", "wow!" ]
```

Indexing and Slicing

Strings and Lists Are Made of Parts

Both strings and lists can be broken down into parts (characters in a string, items in a list). How can we access a specific part?

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 = "SAMPLE"
c = s[2] # "M"
```

We can get the number of characters in a string with the built-in function `len(s)`. This function will prove useful soon.

We can use indexing on lists too!

Like strings, lists are ordered collections, so we can index into them the same way.

```
lst = [ "a", "b", "c", "d" ]
lst[1] # indexing – "b"
```

Just like strings, we can get the number of elements in a list with the built-in function `len(lst)`.

Activity: Guess the Index

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

`"a"`?

`"c"`?

`"3"`?

What will `["a", "b", "c"][1]` evaluate to?

Common Indexes in Strings and Lists

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
```

How do we get the first item in a list?

```
lst[0]
```

How do we get the last item in a list?

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

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

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

String Slicing Produces a Substring

We can also get a whole substring from a string or a sublist 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"
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.

`lst[::]` and `lst[:]` are both the list itself, unchanged (we can remove the second colon when the step is 1)

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

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

`lst[::3]` is every third item of the string (step is `3`)

Example: Extract Information With Slicing

Let's assume we have a variable `recipe` that holds a list with the first element being the name of the recipe and the rest of the elements being the ingredients.

```
recipe = ["Cake", "flour", "sugar", "eggs", "oil"]
```

We want to extract just the ingredients from the list. We can use list slicing to remove the first element of the list but keep the rest. How do we know where to start and end the slice?

Start the slice at the index of the first ingredient, and leave the end open to include everything to the end of the list:

```
ingredients = recipe[1:] # ["flour", "sugar", "eggs", "oil"]
```

String and List Methods

Built-in List Functions

There are a few built-in functions we'll want to use with lists.

```
len(lst) # length of a list (or string)
min(lst) # smallest element of the list
max(lst) # biggest element of the list
sum(lst) # total sum of elements in the list

random.choice(lst) # picks a random element from the
list
```

Methods Are Called Differently

Most string and list built-in functions (and data structure functions in general) work differently from the functions we're used to. 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 an **object**. We will learn more about objects next week!

Don't Memorize, Use Documentation!

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

When you want to know what methods do and what's available, **use the Python documentation** to look for the name of a function that you know probably exists.

If you can remember what has already been done, you can always look up the name and parameters when you need them.

Some Method Examples

Some Methods Return Information

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

```
s = "Hello"
lst = [5, 10, 10, 20]

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

Some Methods Create New Values

`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"
```

Some Methods Change Data Types

`s.split(c)` splits up a string into a list of strings based on the separator character, `c`.

`c.join(lst)` joins a list of strings together into a single string, with the string `c` between each pair.

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

f = "-".join(["ab", "cd", "ef"])
# f = "ab-cd-ef"
```

Looping with Strings and Lists

Looping Over Strings With Range

We previously learned about looping over numbers. Now that we have 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])
```

character position

use indexing to get the value

Looping Over Lists With Range

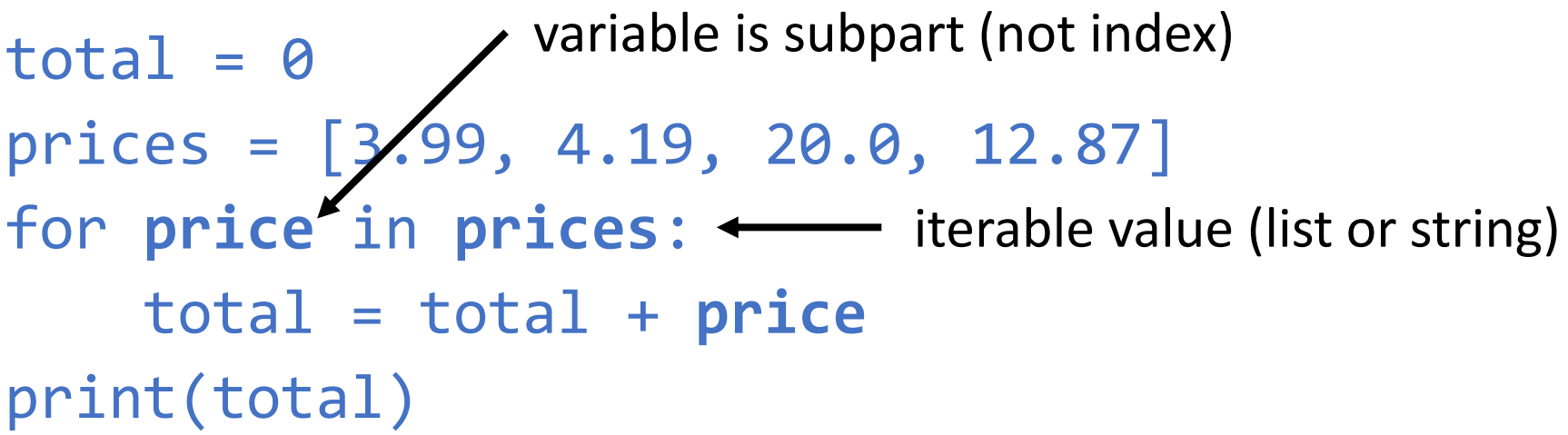
Looping over lists works the same way. Here's a loop that sums all the values in a list of prices.

```
total = 0
prices = [3.99, 4.19, 20.0, 12.87]
for i in range(len(prices)):
    total = total + prices[i]
print(total)
```

Looping Directly Over Lists

But do we actually need to use indexes here? We don't care where in the list a price occurs, we just need to visit each price once.

When we don't care about position, we can use the fact that strings and lists are made of many smaller parts to iterate over them **directly**.




```
total = 0
prices = [3.99, 4.19, 20.0, 12.87]
for price in prices:
    total = total + price
print(total)
```

Which to Use?

Which approach should we use, `range` or iterating directly?

```
total = 0
prices = [3.99, 4.19, 20.0, 12.87]
for i in range(len(prices)):
    total = total + prices[i]
print(total)
```

```
total = 0
prices = [3.99, 4.19, 20.0, 12.87]
for price in prices:
    total = total + price
print(total)
```

It depends on the problem. `range` is best when we need direct information about the position of a value. Otherwise, iterating directly is fine.

Algorithmic Thinking with Strings

If you need to solve a problem that involves doing something with every subpart of a string/list, use a loop.

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 __________

    if s[i] ____________ : # if the character is not a space

        result = result + __________

print(result) # "Wow!Thisissoexciting!"
```

You do: Example: `findMax(nums)`

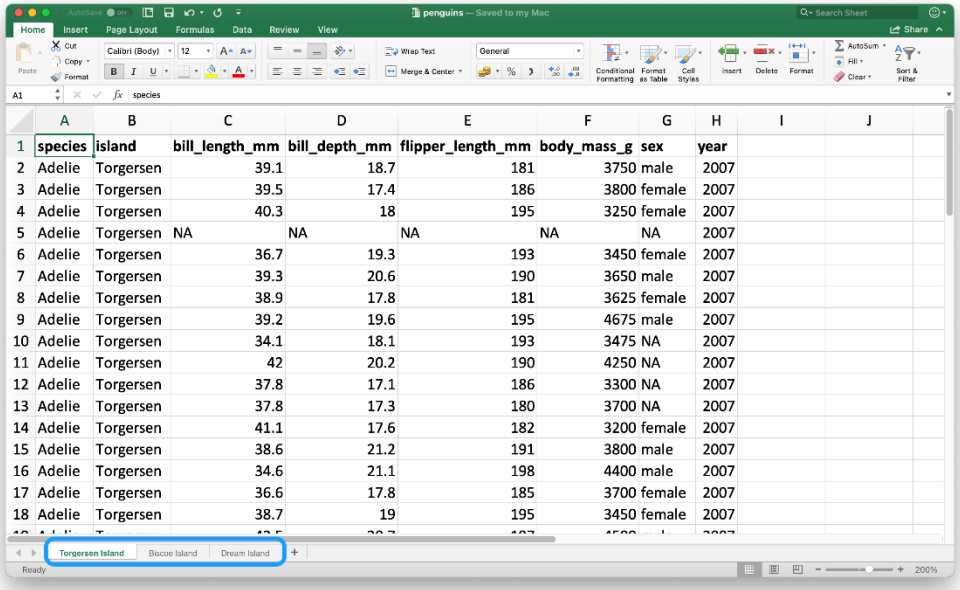
You do: reorder the lines below to make a function that finds the maximum value in a list of integers. We'll often use this algorithmic structure to find the 'biggest' item in a structure.

Correct line number	Code out of order
	`    return biggest`
	`    biggest = nums[0] # why not 0 or -9999?`
	`        biggest = nr`
	`def findMax(nums):`
	`    for nr in nums:`
	`        if nr > biggest:`

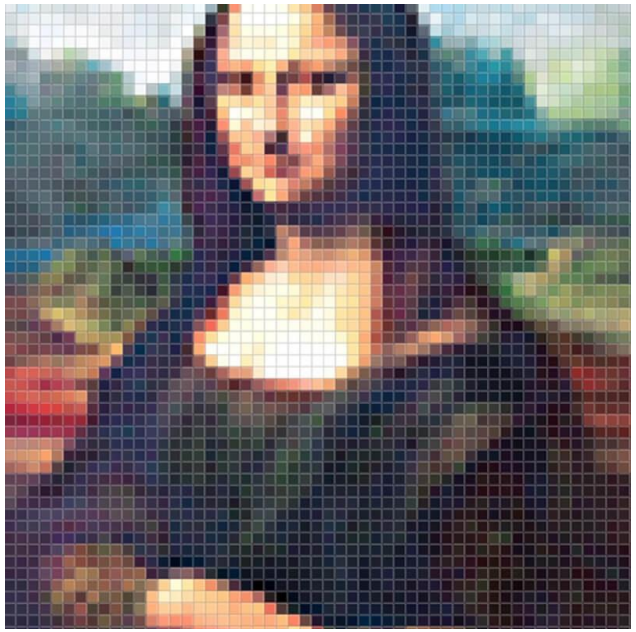
2D Lists

2D Lists are Lists of Lists

We often need to work with data that is **two-dimensional**, such as the coordinates on a grid, values in a spreadsheet, or pixels on a screen. We can store this type of data in a **2D list**, which is a list where the items are themselves lists.



	A	B	C	D	E	F	G	H
1	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
2	Adelie	Torgersen	39.1	18.7	181	3750	male	2007
3	Adelie	Torgersen	39.5	17.4	186	3800	female	2007
4	Adelie	Torgersen	40.3	18	195	3250	female	2007
5	Adelie	Torgersen	NA	NA	NA	NA	NA	2007
6	Adelie	Torgersen	36.7	19.3	193	3450	female	2007
7	Adelie	Torgersen	39.3	20.6	190	3650	male	2007
8	Adelie	Torgersen	38.9	17.8	181	3625	female	2007
9	Adelie	Torgersen	39.2	19.6	195	4675	male	2007
10	Adelie	Torgersen	34.1	18.1	193	3475	NA	2007
11	Adelie	Torgersen	42	20.2	190	4250	NA	2007
12	Adelie	Torgersen	37.8	17.1	186	3300	NA	2007
13	Adelie	Torgersen	37.8	17.3	180	3700	NA	2007
14	Adelie	Torgersen	41.1	17.6	182	3200	female	2007
15	Adelie	Torgersen	38.6	21.2	191	3800	male	2007
16	Adelie	Torgersen	34.6	21.1	198	4400	male	2007
17	Adelie	Torgersen	36.6	17.8	185	3700	female	2007
18	Adelie	Torgersen	38.7	19	195	3450	female	2007



2D List Example

The table below shows cities in Pennsylvania, the counties they're in, and their population.

City	County	Population
Pittsburgh	Allegheny	303,255
Philadelphia	Philadelphia	1,567,258
Allentown	Lehigh	124,880
Erie	Erie	92,957
Scranton	Lackawanna	75,805

In Python, we could represent this using the 2D list to the right. Each of the five elements of the list **is itself a list**!

Population List

0.
```
0. "Pittsburgh"
1. "Allegheny"
2. 303255
```

1.
```
0. "Philadelphia"
1. "Philadelphia"
2. 1567258
```

2.
```
0. "Allentown"
1. "Lehigh"
2. 124880
```

3.
```
0. "Erie"
1. "Erie"
2. 92957
```

4.
```
0. "Scranton"
1. "Lackawanna"
2. 75805
```

Syntax of 2D Lists

Setting up a 2D list is no different than setting up a 1D list; each inner list is one data value.

```
cities = [ ["Pittsburgh", "Allegheny", 303255],
           ["Philadelphia", "Philadelphia", 1567258],
           ["Allentown", "Lehigh", 124880],
           ["Erie", "Erie", 92957],
           ["Scranton", "Lackawanna", 75805] ]
```

This is across multiple lines but treated as one line because each part ends with a comma.

The length of a 2D list is the number of lists in the outer list.

```
len(cities)  # 5
```

Indexing of 2D Lists

```
cities = [ ["Pittsburgh", "Allegheny", 303255],
           ["Philadelphia", "Philadelphia", 1567258],
           ["Allentown", "Lehigh", 124880],
           ["Erie", "Erie", 92957],
           ["Scranton", "Lackawanna", 75805] ]
```

When indexing into a 2D list, the **first square brackets index into a row** and the **second index into a column**.

```
cities       # the list of lists
```

```
cities[2]    # [ "Allentown", "Lehigh", 124880 ]
```

```
cities[2][1] # "Lehigh"
```

Learning Goals

- Identify common properties of **strings** and **lists**
- **Index** and **slice** into strings and lists to break them up into parts
- Use for loops to loop over strings and lists by **index**
- Trace code using **strings**, **1D lists,** and **2D lists**

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.