

Lists and Methods

15-110 – Friday 02/06

Announcements

- Hw2 due Monday at noon

Reminder: Help Available for Homework

As we keep making progress, we will work with more complex topics in the assignments. You may find some of the problems difficult and may get stuck while working.

When you need help, use resources that can give you feedback to help you learn.

Collaboration, Piazza, and Office Hours (TA and Instructor) can all make solving problems much easier and more enjoyable. If you need to, **submit partial work**, then use the revision deadline to fix your work once you get feedback.

Do not use approaches that give you the solution directly, like asking to see a friend's solution or asking an AI agent to solve a problem. This entirely skips the learning process and is also an academic integrity violation.

Learning Goals

- Read and write code using **1D** and **2D lists**
- Use string/list **methods** to call functions directly on values

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: lists, dictionaries, trees, graphs

Efficiency: search algorithms, Big-O, tractability

Lists

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
```

Core List/String Operations

Lists share most of their core operations with strings. You can **concatenate** lists together, just like strings.

```
[ 1, 2 ] + [ 3, 4 ] # concatenation - [ 1, 2, 3, 4]
```

And you can **repeat** lists an integer number of times, again like strings.

```
[ "a", "b" ] * 2 # repetition - [ "a", "b", "a", "b" ]
```

We learned about **indexing**, **slicing**, and **membership checks** last time- those work on lists too.

```
lst = [ "a", "b", "c", "d" ]  
lst[1] # indexing - "b"  
lst[2:] # slicing - [ "c", "d" ]  
"c" in lst # membership - True
```

Sidebar: Built-in List Functions

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

```
len(lst) # length of a list
```

```
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
```

Activity: Evaluate the Code

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

```
[ 5 ] * 3
```

```
["a", "b", "c"][1]
```

```
min([5, 1, 8, 2])
```

Looping Over Lists

Looping over lists works the same way as with strings. We can use a for loop over the indexes of the list to access each item. 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)
```

Example: findMax(nums)

Let's write a function that finds the maximum value in a list of integers.

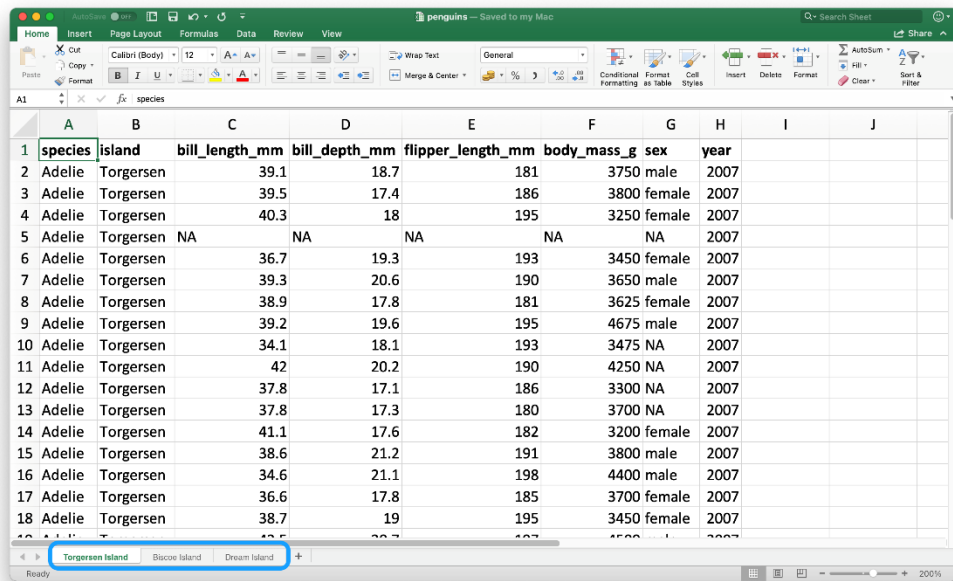
```
def findMax(nums):  
    biggest = nums[0] # why not 0? Negative numbers!  
    for i in range(len(nums)):  
        if nums[i] > biggest:  
            biggest = nums[i]  
    return biggest
```

We'll often use this algorithmic structure to find the 'biggest' item in a structure.

2D Lists

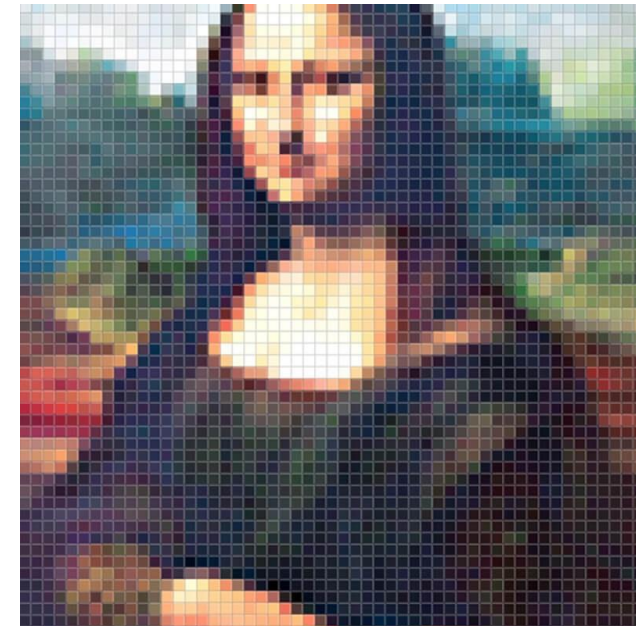
2D Lists are Lists of Lists

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 screenshot of a Google Sheet titled "penguins" showing a table of penguin data. The table has columns for species, island, bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g, sex, and year. The data is organized into rows, with the first row being the header. The table is displayed in a grid format, with the first row highlighted in green.

	A	B	C	D	E	F	G	H	I	J
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"`



Looping Over 2D Lists

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

We can loop over a 2D list the same way we loop over any other list. Indexing into a list once will produce an **inner list**. We'll need to index a second time to get a value.

```
def getCounty(outerList, cityName):  
    for i in range(len(outerList)):  
        innerList = outerList[i]  
        if innerList[0] == cityName:  
            return innerList[1]  
    return None # city not found
```

Looping Over All 2D List Elements

When you loop over a 2D list and want to access *every* element, you need to use **nested for loops**. Often, the outer loop iterates over the indexes of the outer list (**rows**) and the inner loop iterates over the indexes of the inner list (**columns**).

```
gameBoard = [ ["X", " ", "0"], [" ", "X", " "], [" ", " ", "0"] ]
for row in range(len(gameBoard)): # each row is a list
    boardString = ""
    for col in range(len(gameBoard[row])): # each col is a string
        boardString = boardString + gameBoard[row][col]
    print(boardString) # separate rows on separate lines
```

Activity: getTotalPopulation(cities)

Fill in the blanks for the function `getTotalPopulation(cities)` that takes the city-information 2D list from before and finds the total population of all cities in the list.

```
def getTotalPopulation(cities):  
    _____ = 0  
    for row in range(_____):  
        pop = _____  
        total = _____  
    return total
```

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

Hint: note that the population is in the third column. Which index corresponds to that?

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()
```

Some Methods Create New Values

Other string methods return a new value 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"
```

`s.strip()` returns a new string with excess whitespace (spaces, tabs, newlines) at the front and back removed.

```
d = "    Hi there    ".strip() # d = "Hi there"
```

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
```

Some Methods Change Data Types

Finally, some methods let you convert between strings and lists as needed.

`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"
```

[if time] 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.

Learning Goals

- Read and write code using **1D** and **2D lists**
- Use **list methods** to change lists without variable assignment