FOR PRESENTER USE ONLY

Data Structures

Overview

In this lesson, students will learn how to use lists and dictionaries to organize data.

Duration

120 minutes

Learning Objectives

In this lesson, students will:

- Use lists and list methods to manage collections of data.
- Use index-based retrieval to access and manipulate list items.
- Use dictionaries to represent data with multiple properties.

Pre-Class Materials and Preparation

For remote classrooms: Virtual breakout rooms and Slack may be needed to facilitate the partner exercise and discussions. As you plan for your lesson:

- Consider how you'll create pairs for the partner exercise (randomly, or with pre-assigned partners).
- Determine how (if at all) exercise timing may need to be adjusted.
- For helpful tips, keep an eye out for the **For remote classrooms** tag in the speaker notes.
- Prepare screenshots and answers to exercises in advance so that they can be easily shared in Slack during your lecture.

Pre-Class Materials and Preparation (Cont.)

Pre-Work Review: Many of the concepts covered in this class will recap what students learned in the pre-work; in particular, these myGA lessons:

- Manipulating Variables in Python
- Data Types in Python

We recommend reviewing the pre-work lessons to understand what students will come to class knowing. Click here to see the study guides for these lessons. You can also view the pre-work materials on myGA.

For review or extension topics, review the notes in the pre-work for recommendations on how to teach the concept. Leverage the discussions, exercises, and knowledge checks to gauge students' understanding of concepts. You can adjust your approach to the lesson overall based on how students do with these exercises.

FOR PRESENTER USE ONLY

Suggested Agenda

Time	Activity
0:00–0:15	**Welcome + Introduction to Data Structures**
0:15–0:55	**Lists and Tuples**
0:55–1:05	**Break**
1:05-1:30	**Dictionaries**
1:30–1:50	**Combining Data Structures**
1:50–2:00	**Wrapping Up, Q&A, and Exit Ticket Completion**

Jupyter Notebook

The exercises referenced in this lesson can be found in the Python Workbooks + Data folder.

Our first few lessons begin by reviewing the notebook, as the same material was covered in the pre-work.

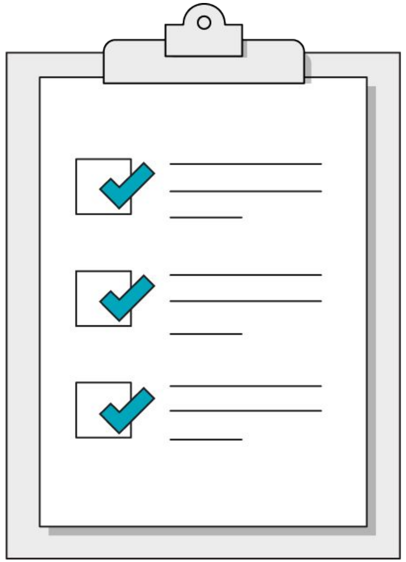
If students feel that they are able to confidently solve the challenges in the workbook, you can quickly skim through the lesson content and have them only complete the challenges.

Data Structures

GENERAL ASSEMBLY

Our Learning Goals

- Use lists and list methods to manage collections of data.
- Use index-based retrieval to access and manipulate list items.
- Use dictionaries to represent data with multiple properties.



What We'll Practice Today

This class is a **blended learning experience**. It connects to and reinforces topics that you encountered in the myGA pre-work.

We're going to return to topics covered in the pre-work and build upon them:

- Data structures like **lists, tuples, and dictionaries**.
- **Combining** data structures.
- **List methods**.

Solo Exercise:

Jupyter Notebook Review

10 minutes

Let's dive right in and use what we learned in the pre-work! We want to understand where you are in your learning journey so that we can give the best possible experience in class.

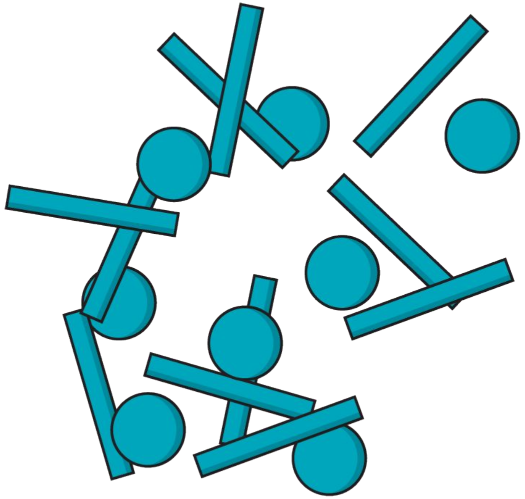
Look over the exercises in today's Jupyter Notebook and attempt any that seem immediately doable to you.

Then, **rate your confidence level** on today's subjects from 1–5.

Our Sad, Unstructured Data

So far, our variables have only stored a single piece of information, or data. Imagine trying to represent a data set using this method:

```
customer_one = "Anees Rosario"
customer_two = "Alya Pham"
customer_three = "Marc Wormald"
customer_four = "Ellie-Mai Muir"
```

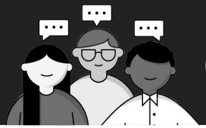


You won't get very far using a new variable for every new piece of data!

Data Structures to the Rescue!

Fortunately, Python provides us with some options for compiling data into a single structure:

- **Lists** are exactly what they sound like: comma-separated lists of values.
- **Tuples** are like lists but more strict. You can't update the values in a tuple!
- **Dictionaries** allow us to associate multiple properties together, much like a single row of a spreadsheet can contain many columns.

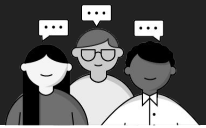


Discussion:

Guess the Data Structure!

Before we get into the specifics, let’s think about **which of our three data structures (lists, tuples, and dictionaries) makes the most sense when representing the following information:**

- The five continents on Earth.
- An item for sale in our store.
- All transactions conducted in our store.
- A single transaction conducted in our store.
- A weather forecast for today.
- The three possible quality rankings for produce.



Discussion:

Data Structures Revealed

Tuples hold values that will not change, while lists can be updated more easily. Dictionaries represent a single, but complex, piece of information.

- The five continents on Earth: **Tuple**.
- An item for sale in our store: **Dictionary**.
- All transactions conducted in our store: **List**.
- A single transaction conducted in our store: **Dictionary**.
- A weather forecast for today: **Dictionary**.
- The three possible quality rankings for produce: **Tuple**.

Data Structures

Lists and Tuples

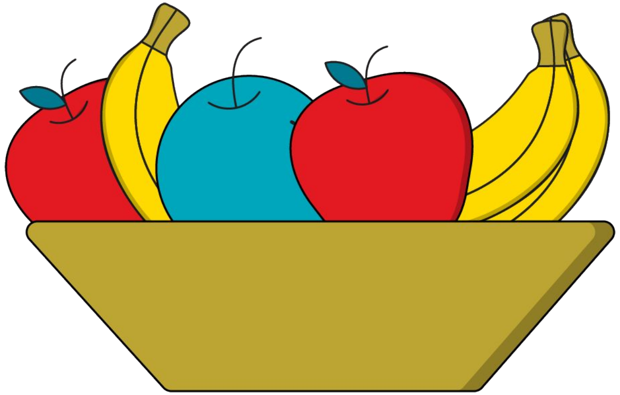
Lists

A **list** is an ordered collection of data combined into one variable.

Each item in an list is assigned an **index** value based on its position. These index values allow us to access individual elements within the list.

```
["banana", "orange", "apple"]
```

0 **1** **2**



Syntax

```
fruits = ["banana", "orange", "apple"]
```

You create a list using a set of square brackets.

Inside the brackets, each value must be separated by a comma.

Although it most often makes sense for all values to be the same data type, there’s no rule against using a list to store data of varying types.

Accessing List Values

```
fruits = ["banana", "orange", "apple"]
# fruits[0] will access "banana"
# fruits[1] will access "orange"
# fruits[2] will access "apple"
```

Access list items using square brackets around their index values. It’s pretty simple — just remember that the first index value is always zero!

Updating List Values

```
fruits = ["banana", "orange", "apple"]
fruits[0] = "kiwi"
fruits[1] = "strawberry"
# The fruits list now looks like ["kiwi", "strawberry", "apple"]
```

We can update the items in a list using the same index syntax. The same assignment operator that we used with individual variables can also be used to set the value of a specific item in a list.

Solo Exercise:

2.1 Colors of the Rainbow

15 minutes

Let's practice creating a list, along with accessing and updating values using the specific list index syntax.

List Methods

In addition to using index syntax to refer to specific items, we can also use **methods** of lists to perform specific actions on that list. The syntax looks like:

```
list_name.method_name( any_inputs_here )
```

The parentheses might remind you of the print statements we’ve used thus far. That’s because, just like print(), methods can also accept specific input, or **arguments**, for their operations.

Adding Items With `.append()`

```
    fruits = ["orange"]

    fruits.append("kiwi")
    # fruits is now: ["orange", "kiwi"]
```

The **`.append()`** method adds the item inside the parentheses to the **end** of the list.

Adding Items With `.insert()`

```
fruits = ["orange", "kiwi"]

fruits.insert(1, "lemon")
# fruits is now: ["orange", "lemon", "kiwi"]
```

The **`.insert()`** method starts by taking an index number, then adds the second argument in the parentheses to the given index.

Note the original item at that index is simply bumped to the next spot in the list.

Removing Items With `.pop()`

```
fruits = ["orange", "kiwi", "lime"]
fruits.pop()
# fruits is now: ["orange", "kiwi"]
fruits.pop(0)
# fruits is now: ["kiwi"]
```

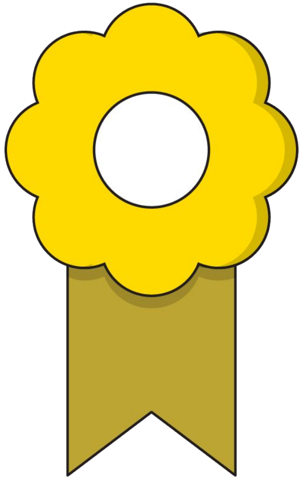
The **`.pop()`** method can be used to remove the item at a specific index, or, if you don't provide any index, it will simply remove the last item in the list.

Partner Exercise:

2.2 The Wait List

20 minutes

Let's practice these list methods and more by managing the waiting list for the Python Academy, the most exclusive private boarding school in all of East Python-shire.



More List Functions

	Explanation	Example

`len()`	Produces the length of the list.	`fruits = ["apples", "bananas", "oranges"]`
		`len(fruits)`
		`# 3`

`min() and max()`	Produces the minimum or maximum.	`scores = [10, 20, 30]`
		`min(scores)`
		`# 10`
		`max(scores)`
		`# 30`

`sum()`	Produces the total sum of all items in the list.	`sales = [25, 15, 10]`
		`sum(sales)`
		`# 50`

Tuples: You Already Know Them!

The good news about tuples is that they work exactly the same way as lists, but simply use parentheses instead of square brackets.

```
valid_statuses = ("operational", "faulty", "non-operational")
```

The difference is that tuple values are **immutable**, meaning you cannot change values in a tuple once it’s been created; thus, tuples are mostly used for reference information that will not change.

Sets

A **set** is an unordered collection of unique values. You can’t use an index to access a specific element in a set. Instead, sets have methods for accessing or manipulating collection members:

```
primary_colors = { "red", "blue", "yellow" }

primary_colors.add("green")

primary_colors.remove("yellow")
```

Data Structures

Dictionaries

Dictionary Definition

A dictionary is a collection of associated **keys** and **values**. Think of a dictionary like a row of data in a spreadsheet — you have column names, which are your keys, and then you have the actual values inside of the columns.

item_name	category	price
tomatoes	food	1.99

```
# The above row would be translated into this dictionary

{ "item_name": "tomatoes", "category": "food", "price": 1.99 }
```

Syntax for Dictionaries (Cont.)

```
item = { "item_name": "tomatoes", "category": "food", "price": 1.99 }
```

Curly braces are used to start and end the dictionary.

Syntax for Dictionaries (Cont.)

```
item = { "item_name": "tomatoes", "category": "food", "price": 1.99 }
```

Key names are provided first and surrounded by quotation marks.

Syntax for Dictionaries (Cont.)

```
item = { "item_name": "tomatoes", "category": "food", "price": 1.99 }
```

Values are provided after a colon and can be of any data type.

Syntax for Dictionaries (Cont.)

```
item = { "item_name": "tomatoes", "category": "food", "price": 1.99 }
```

Finally, note the **commas** separating each key-value pair.

Accessing Values in a Dictionary

```
item = { "item_name": "tomatoes", "category": "food", "price": 1.99 }
```

Accessing specific values in a dictionary works a lot like accessing specific items in a list. This time, instead of numbered indices, we use the name of the key.

```
item["category"]

# This accesses the value "food"
```

Adding Values to a Dictionary

One of the most powerful features of a dictionary is the ability to add new keys and values whenever necessary.

```
item["fresh"] = True
```

```
item["discount"] = .15
```

The same syntax can update a given property as well.

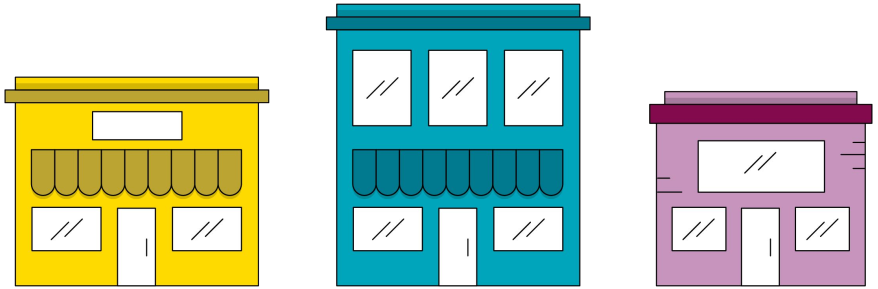
```
item["fresh"] = False
```

Solo Exercise:

2.3 Key Value Properties

15 minutes

Practice accessing and updating properties in a dictionary by modifying a real estate listing in Section 2.3 of the Jupyter Notebook.

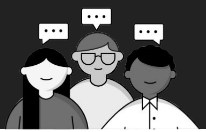


Discussion:

Representing a Data Set

Based on what we know about data structures, **how would we represent an entire data set (like a .csv file)**, with many rows and columns of information, **in Python**?

item_name	category	price
tomatoes	food	1.99
mango	food	3.00
journal	office	15.00



Discussion:

Representing a Data Set

Answer: A list of dictionaries! Each dictionary is a specific row, and the list is our “table” collecting all of the rows together.

```
[ {"item_name" : "tomatoes", "category": "food", "price" : 1.99},

{"item_name" : "mango", "category": "food", "price" : 3.00},

{"item_name" : "journal", "category": "office", "price" : 15.00} ]
```

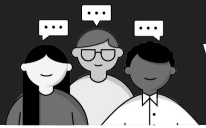
Data Structures

Combining Data Structures

Nested Data Structures

So far, we've seen lists and dictionaries that organize simpler data types, like strings and numbers. However, it's common to see the more complex data structures nested within each other, creating elaborate mazes of data.

When getting data from outside sources, for example, you might be given a dictionary that contains another dictionary that contains a list containing another dictionary that **finally** contains the information you're looking for!



Discussion:

What Is This Accessing?

Think about what the following line of code is saying, one layer at a time:

```
authors[0]["books"][1]["title"]
```

Without seeing the exact data structure in question, what is this line attempting to access?

Discussion:

Peeling Back the Layers

```
authors[0]

# There is a list called authors and we want the first thing in it.
```

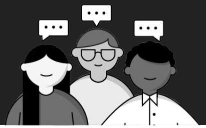
Discussion:

Peeling Back the Layers

```
authors[0]["books"]

# There is a list called authors and we want the first thing in it.

# That first thing is a dictionary and we want the "books" property.
```

Discussion:

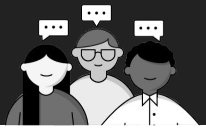
Peeling Back the Layers



```
authors[0]["books"][1]

# There is a list called authors and we want the first thing in it.

# That first thing is a dictionary and we want the "books" property.

# Turns out "books" is a list and we want the second thing in it.
```

Discussion:

Peeling Back the Layers



```
authors[0]["books"][1]["title"]

# There is a list called authors and we want the first thing in it.

# That first thing is a dictionary and we want the "books" property.

# Turns out "books" is a list and we want the second thing in it.

# That second thing is another dictionary with a "title" property.
```

Keep Calm and print() On

When dealing with a complex, multi-level data structure, you can lean on print statements to help peel back the layers one at a time. The syntax for accessing elements doesn’t change, you just simply need more layers:

```
print(authors[0])

print(authors[0]["books"])

print(authors[0]["books"][1])

print(authors[0]["books"][1]["title"])
```

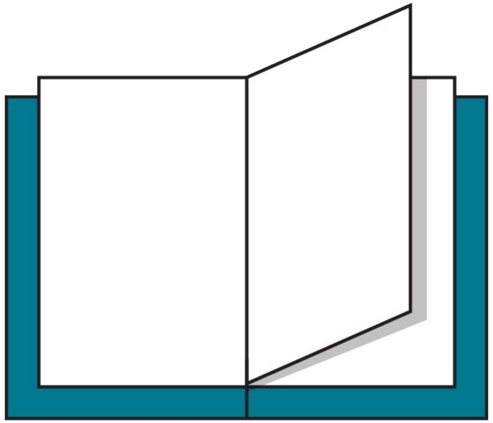
Partner Exercise:

2.4 Python Librarian

15 minutes

Managing nested data structures can be difficult!

Work through the challenges of accessing and modifying the “authors” dictionary found in Section 2.4 of the workbook.



Data Structures

Wrapping Up

Recap

In today's class, we...

- Used lists and list methods to manage collections of data.
- Used index-based retrieval to access and manipulate list items.
- Used dictionaries to represent data with multiple properties.

Looking Ahead

On your own:

- Ensure that you've completed the Python pre-work and pre-work quiz.

Next Class:

Conditionals

Don’t Forget: Exit Tickets!