

---

---

# Computer Science

Summer Springboard

---

---

# Day 3: The Plot Thickens



**SUMMER**  
SPRINGBOARD  
Look Inward. Go Upward.

# Pop Quiz!

Let's make sure we understand everything that we have covered so far. Take a few minutes to think about the answers to the following questions and then discuss as a class:

1. What is the difference between a float and an int?
2. What are conditionals and what is the correct syntax in Python?
3. What do **and**, **or**, and **not** do?
4. What is the difference between `=` and `==`?
5. What is a type conversion?



## (Slightly) More Advanced Data Types

- Now that we've covered some basic data types, let's build on these and create something more complex
- Namely, lists, tuples, and dictionaries!
- Let's jump into lists first!

## Not so fast!

- There's something we need to talk about before we get into lists and tuples...

# Mutability vs. Immutability

# The Concept of Change in Data Types

- Think of data types as creatures in the Python world. Some like change (mutable), while others resist it (immutable).



# Introducing the Mutables

- Mutable types are like chameleons, they can change their appearance (values).
- Key Players: list, dict, set.
- Characteristics:
  - Lists are like backpacks - items can be added, removed, or changed.
  - Dictionaries are like filing cabinets - you can add new files or rearrange existing ones.
  - Sets are like party guest lists - guests (elements) can come and go.





# Meeting the Immutables

- Immutable types are like mountains, solid and unchanging.
- Key Players: int, float, str, tuple.
- Characteristics:
  - Integers and Floats are like carved stones, their value etched forever.
  - Strings are like stars in the sky, a fixed pattern of characters.
  - Tuples are like time capsules, once sealed, their contents remain the same.

# Why do we care?

- Mutables are versatile but can be tricky; changes affect all references to the object.
- Choose mutable types when you need to change the size or content of your data.
- Immutables are reliable and consistent; they ensure data integrity.
- Opt for immutable types for fixed data where consistency and integrity are key.

# A Real-World Scenario

- Mutable: Like editing a live document, changes are seen by everyone with access.
- Immutable: Like sending a printed letter, to change the content, you must send a new letter.



# Collections



**SUMMER**  
SPRINGBOARD  
Look Inward. Go Upward.

# Back on track: Lists

- Imagine an infinite bookshelf. You can fill this bookshelf how you wish; for example, maybe you put a few of your favorite books, a LEGO set, and a stuffed animal. The items on the shelf have some sort of order - you can count them from left to right - but you can take items down, put new items up, or rearrange them.
- Lists in Python are like this imaginary bookshelf in that lists are capable of storing an ordered collection of items



# Creating a List in Python

- To create a list, use the following syntax:

**list\_name = [item1, item2, ...]**

- Notice that the name of the list is separated by underscores, the items of the list are separated using commas, and we are using square brackets.
- You can have as many items as you would like in your list.
  - If you want an empty list (a list with zero items), simply write:  
**list\_name = []**

# List Characteristics

- Lists in Python are mutable, which means that you can change their content at any time, similar to the bookshelf in our analogy earlier.
- Lists are ordered, which means that the items within the list have a defined order
- Lists are flexible, meaning that they can contain mixed data types; they don't need to just hold “books” or “stuffed animals”

# Adding to a List

- Say we have a list named **favorite\_books** which is created in the following way:

**favorite\_books = ["1984", "To Kill a Mockingbird", "The Great Gatsby"]**

- We can add to our list using the **append()** function:  
**favorite\_books.append("Pride and Prejudice")**
- This function simply adds a new item to the end of the list.



# Accessing Items in a List

- To access an item from a list, use the following notation:  
**`first_book = favorite_books[0]`**
- Remember that lists are ordered. This means that each element in the list has an index (a number representing the position of the element in the list).
- In Python (and most common programming languages), the indices begin with 0, meaning that the first element in the list is index 0, the second element is index 1, etc.
- To access a specific element in a list, enclose the index of the element you want to access in square brackets and put it at the end of your list's name, as we did above.



## Alternate Way to Add to a List

- If you want to add an item to a specific location in your list, use the **insert()** function:  
**favorite\_books.insert(2, "Frankenstein")**
- The above line of code inserts the string "Frankenstein" into our list at index 2, making it the third item in our list.

# Removing Items From a List

- To remove an item from a list, use one of the following syntax options:

**`favorite_books.remove("1984")`**  
**`del favorite_books[0]`**

- The first option removes a specific element from our list, while the second removes an element at a specified index.

# List's Immutable Cousin, Tuple

- Like lists, tuples are ordered collections of items. The items do not need to be of the same type.
- Unlike lists, tuples are immutable, which means that once they are created, you cannot add, remove, or modify its elements.
- To create a tuple, use the following syntax:

**Classic\_movies = ("Casablanca", "Gone with the Wind", "Citizen Kane")**

- An example of a tuple that you have most likely seen before is ordered pairs in math. Coordinate pairs are of the form (x, y) and (at least in the familiar cartesian plane), you cannot add extra elements to this tuple or switch the x and y around (the y value is always on the right, the x on the left)

# Dictionaries

- Dictionaries are Python's efficient filing system, storing data as key-value pairs.
- You can think of a Python dictionary as real-life dictionaries
  - Python dictionary keys -> real-life dictionary words
  - Python dictionary values -> real-life dictionary definitions
- To create a dictionary in Python, use the following syntax:  
**`contact_info = {"Alice": "555-0101", "Bob": "555-0202"}`**



# Dictionary Characteristics

- Dictionaries are mutable.
- Dictionaries are unordered, meaning that there is no fixed order of items; keys retrieve values
- Dictionaries must have unique keys; each key must be distinct



# Navigating Dictionaries

- Accessing Data: **phone\_number = contact\_info["Alice"]**
- Updating Values: **contact\_info["Alice"] = "555-0101"**

# But how do I know which to choose?

- These three collection types can feel very similar and it can be hard to determine when to choose one over another. Here are the general use cases for each:
  - Lists: perfect for sequences where order matters, like steps in a recipe or a to-do list
  - Tuples: generally faster than lists, ensures data integrity, and ideal for use as keys in dictionaries and for returning multiple values from functions (more on this later!)
  - Dictionaries: ideal for lookup tables, associative arrays, and data that's naturally paired (like word definitions or product prices)



# That was a lot!

- We just covered a lot of ground; good work team!
- Take a few minutes to digest what we learned and make sure that you can explain the differences between lists, tuples, and dictionaries.
  - Follow-up: what are some real-life examples of each of these collections?

# Practice Problem Time!

- Open today's Google Colab notebook and follow the instructions up until the line that says "\*\*\*PAUSE\*\*\*"

# Loops

# Captivating Concept

- In Python's world, loops are like mystical paths, allowing you to traverse through a collection or repeat tasks without getting lost in redundancy
- There are two primary pathways for looping: the **for** loop and the **while** loop.

# The 'for' Loop: The Trailblazer

- The **for** loop repeats a block of code a set number of times and is perfect for navigating through each element in a collection (like lists, tuples) or a range of numbers
- Make sure to indent the block of code that you want to repeat.
- For example, the code below prints each element of **treasure\_chest**:

```
for treasure in treasure_chest:  
    print(treasure)
```

- So, the general syntax is:

```
for my_element in (some_collection):  
    doSomething
```

# The 'while' Loop: The Endurance Hiker

- The **while** loop continues as long as a condition is true. It is ideal for repeating a task until a certain state is reached.
- Again, make sure to indent the block of code that you wish to repeat.
- For example, the code below explores the cave until there is no energy left.

```
while energy > 0:  
    explore_cave()
```

- The general syntax is:

```
while (some_condition):  
    doSomething
```



# Controlling Loop Flow

- **break**: the emergency exit from a loop. Stops the loop even if the condition for continuation is true.
- **continue**: the skip button. Moves to the next iteration, bypassing the remaining code in the loop body.
- **pass**: the placeholder. Does nothing, often used as a syntactical requirement.

# Loops: Practical Uses

- **for** loop: reading files line-by-line, processing each item in a list (like sending automated emails to a list of users)
- **while** loop: monitoring real-time data (like a stock price) until a certain condition is met





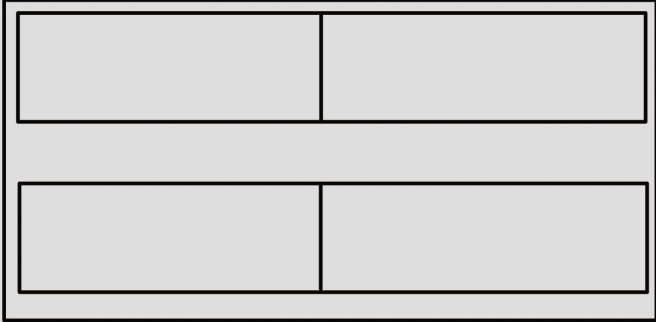
# Looping Wisdom

- Beware of infinite loops; ensure **while** loops have a clear exit.
- Use **for** loops for definite, countable iterations, and **while** loops for more indefinite, condition-based tasks.

# For Loop Animation

- Check out this animation to see a for loop in action!

Computer Memory



→

```
47 treasure_chest = ["gems", "gold", "bones", "diamonds", "rings"]
48
49 for treasure in treasure_chest:
50     print(treasure)
```

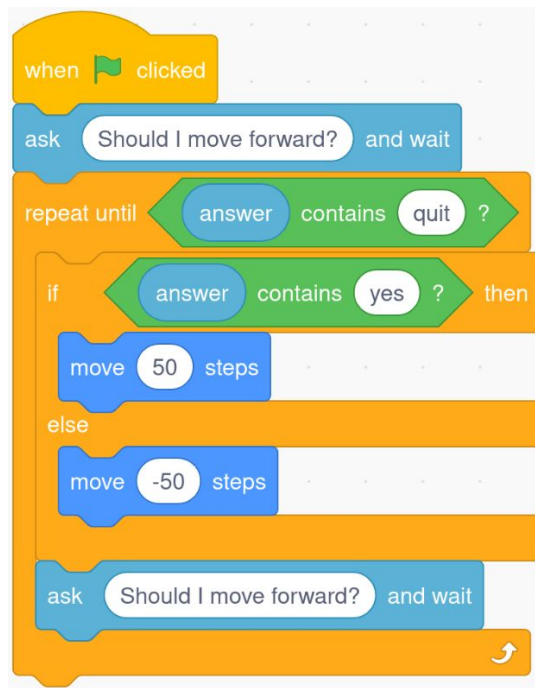
## For Loop Animation (Cont'd)

- Here are some important things to note from the animation in the previous slide:
  - The computer only keeps **treasure** in memory for the life of the **for** loop.
  - After we finish the indented block of code, we hop back up to the start of the loop
  - Once we have exhausted our collection, we exit the loop



# Now, let's do some visual coding practice.

- Return to Scratch: <https://scratch.mit.edu/projects/editor/>
- Edit your code from yesterday and add in a loop block so that you have the following:



# Try your own code!

- Take 10-15 minutes and see if you can implement another way to use loops in Scratch. Share your results with a partner!

## You guessed it... more practice!

- Finish the rest of the exercises in today's notebook.

# Project One Introduction

# Project

- Use at least one collection type, one loop, two conditionals, and 20 lines of code
- Prepare a Google Presentation showcasing your project and a Google Colab notebook to demo your code
- Include the following in your slides:
  - Presentation slide with title and name
  - Description of the project
  - Experience (How difficult was this for you? What did you learn? What do you want to add?)
- You will present and show a demonstration of your code



# Brainstorming

- Spend some time brainstorming ideas for your project either alone, with a partner, or as a group.
- Think about a project that you could expand upon in the future, given more knowledge.