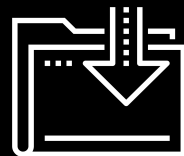




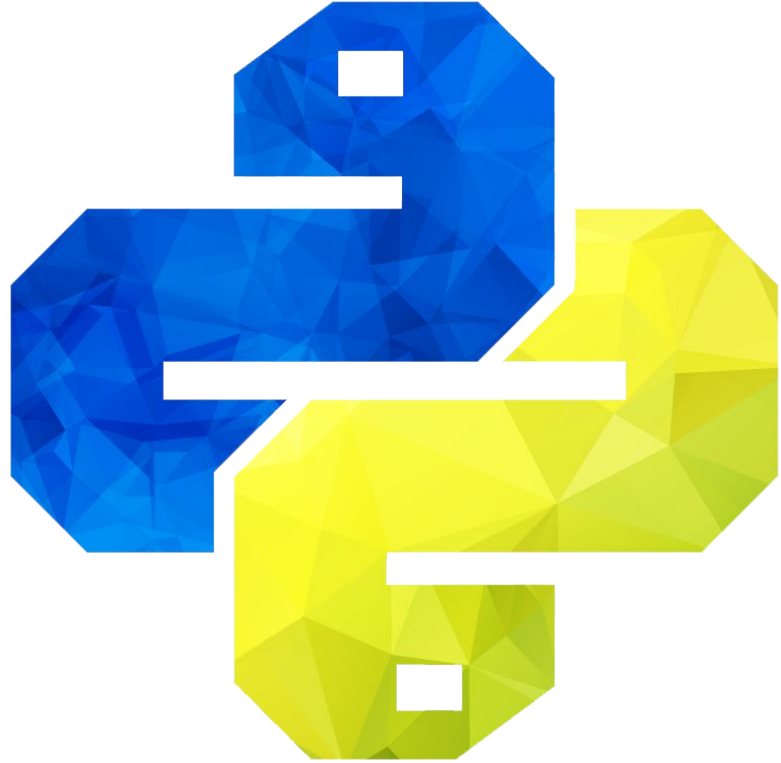
Picking Up the Python Pace

Cybersecurity Boot Camp
Python 1: Day 2



Python: Conditionals and Loops

Today we will pick up the Python programming pace by introducing conditionals and loops.



Class Objectives

By the end of class today, you will be able to:

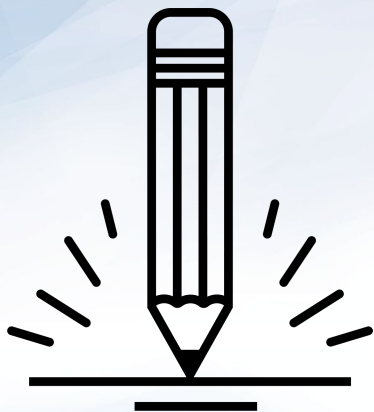
- ❑ Use simple and complex conditionals to create branching paths for programs to execute.
- ❑ Build a command-line application using conditionals and user input.
- ❑ Use `for` and `while` loops to iterate through lists and dictionaries to perform basic operations on collections of data.
- ❑ Use the `range` function to loop through a list with defined ranges.
- ❑ Use the `enumerate` function to loop through a list with a built-in counter.
- ❑ Use `key`, `value`, and `item` methods with loops to convert data from dictionaries into lists.

Last Class we Covered:

Data	Logic
1. Numbers	1. Operators
2. Strings	2. Conditionals
3. Booleans	3. Loops
4. Lists	4. Functions
5. Dictionaries	5. Modules

Today's Class:

Data	Logic
1. Numbers	1. Operators
2. Strings	2. Conditionals
3. Booleans	3. Loops
4. Lists	4. Functions
5. Dictionaries	5. Modules



Activity: Doling Out Data Types

In this activity, you will create a wide variety of variables and print out the information contained within them to the screen.

Instructions sent via Slack.

Suggested Time:
12 minutes



Your Turn: *Doling Out Data Types*

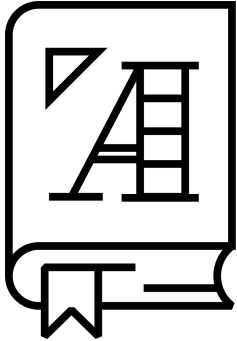
Instructions:

1. **Declare** a variable called `name` and store your name inside it.
2. **Print** the message: "My name is *YOUR NAME HERE*."
3. **Declare** a variable called `age` and store your age inside it.
4. **Print** the message: "I am *YOUR AGE HERE* years old."
5. **Declare** a variable called `pizza_price` and store a number inside it.
6. **Declare** a variable called `number_of_pizzas` and store a number inside it.
7. **Store** the total price for all the pizzas in a variable called `total_price_of_pizzas`.
8. **Print** the message: "The price of pizza is *PRICE OF PIZZA* dollars."
9. **Print** the message: "We are buying *NUMBER OF PIZZAS* pizzas."
10. **Print** the message: "The total price for all the pizzas is *TOTAL PRICE OF PIZZAS*."
11. **Create** a list, called `favorite_countries`. Store the name of four favorite countries inside it.
12. **Print** the message: "My favorite countries are *LIST OF COUNTRIES HERE*."
13. **Create** a dictionary called `contact_information` and store a home phone number, a cell phone number, and an email address inside of it.
14. **Print** out the message: "Please contact me at *EMAIL* or call me at *HOME PHONE*"
15. **Print** out the message: "In case of an emergency call *CELL NUMBER*"
16. **Use** the script file provided to help you through your solution. We filled in the first few answers for you.



• Conditionals


Conditionals



Conditionals are statements ubiquitous in coding that allow programs to perform more complex computations and actions based on `if` statements and boolean conditions.

Conditionals

Conditionals are used to run assigned code only when certain conditions are met. Nearly every conditional starts with **if**.

Keyword	Condition to be tested
<code>if</code>	<code>(x == 1):</code>  The parentheses and colons are syntax to let Python know you're building a conditional statement
<code># code to run if condition is true</code>	

Conditionals

The condition within the parentheses is evaluated as either **true** or **false**.



If condition is **true**, then the code block will run.



If the condition is **false**, the code block associated with that **if** statement will not run and the program continues running through the code.

```
if (x == 1):
```

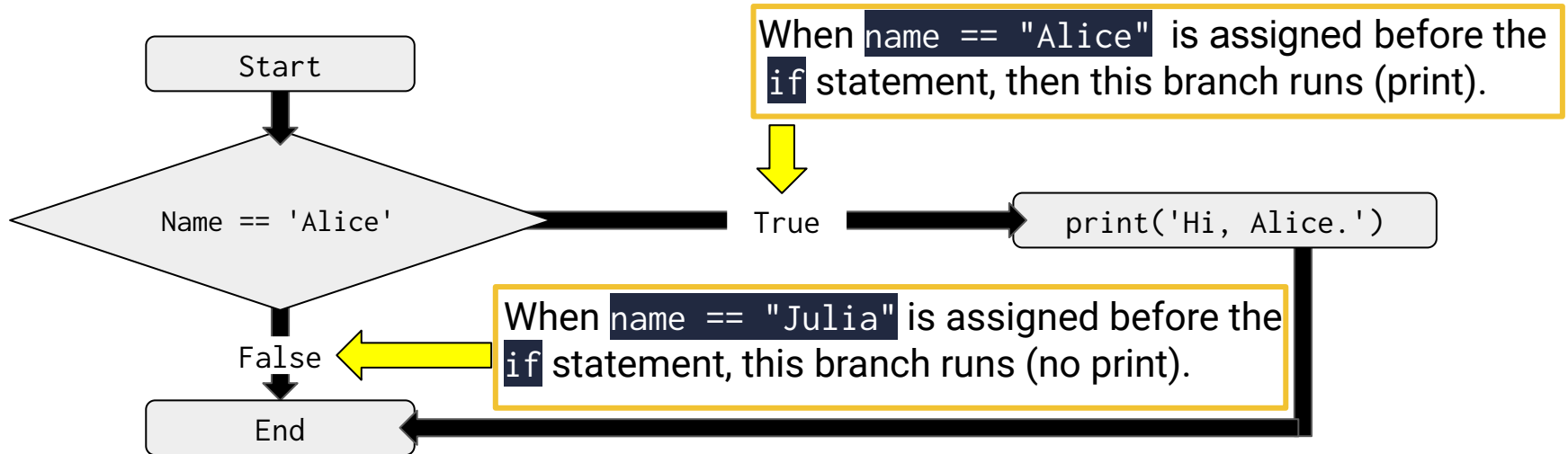
```
# code to run if condition is true
```

Conditionals

```
if
```

```
(name == "Alice"):
```

```
    print("Hi, Alice.")
```



Conditionals Format



The **colon** after the condition is required in Python. It separates the condition from the body of code that follows.



The lines of code after the colon must be **indented**. It is standard in Python to use four spaces for indenting, but any kind of indentation will work so long as it is consistent.

```
if (x == 1):
```

```
    # code to run if condition is true
```

Conditional Syntax

Operators

The `==` operator indicates that the value of one variable is equal to another. So if the values are equal then the conditional statement will evaluate as True.

The `!=` operator indicates that the value of one variable is NOT equal to another. So if the values are NOT equal then the conditional statement will evaluate as True.

Other operators for conditional statements:

The `<` operator indicates that the value of one variable is less than another.

The `>` operator indicates that the value of one variable is greater than another.

The `<=` operator indicates that the value of one variable is less than or equal to another.

The `>=` operator indicates that the value one variable is greater than or equal to another.

Conditionals



`if` statements get more complicated: `if...else`



`else`—if none of the above is true, run this block of code.

```
if
```

```
(x == 1):
```

```
# code to run if condition is true
```

```
else
```

```
# code to run when none of the above conditions are  
true
```

Conditionals

```
princess = "Moana"
```

This code runs

```
if (princess == "Moana"):
    print("The Best 'Princess'")
else:
    print("Not the best princess :(")
```

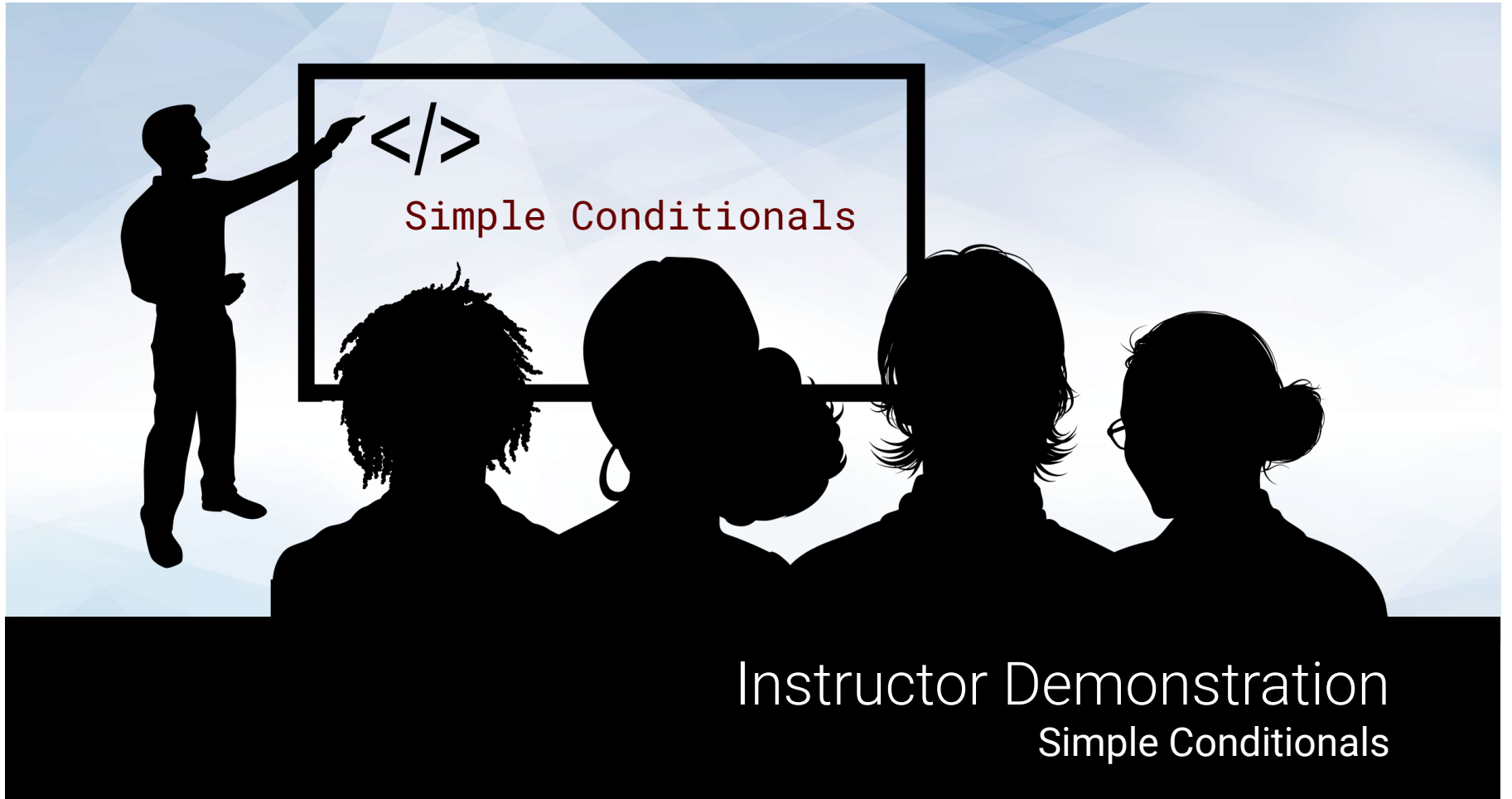
This code doesn't run

```
princess = "Aurora"
```

Now this code doesn't run

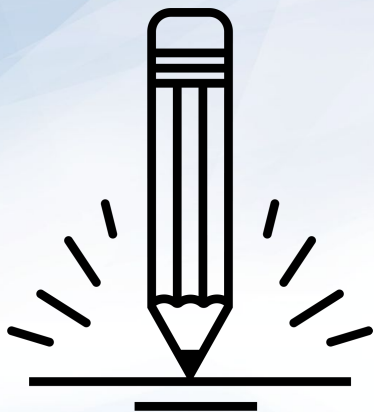
```
if (princess == "Moana"):
    print("The Best 'Princess'")
else:
    print("Not the best princess :(")
```

Now this code runs



Instructor Demonstration

Simple Conditionals



Activity: Password Check

In this activity, you will create a command-line application that will ask users for their password and will check it against the correct one.

Instructions sent via Slack.

Suggested Time:
7 minutes

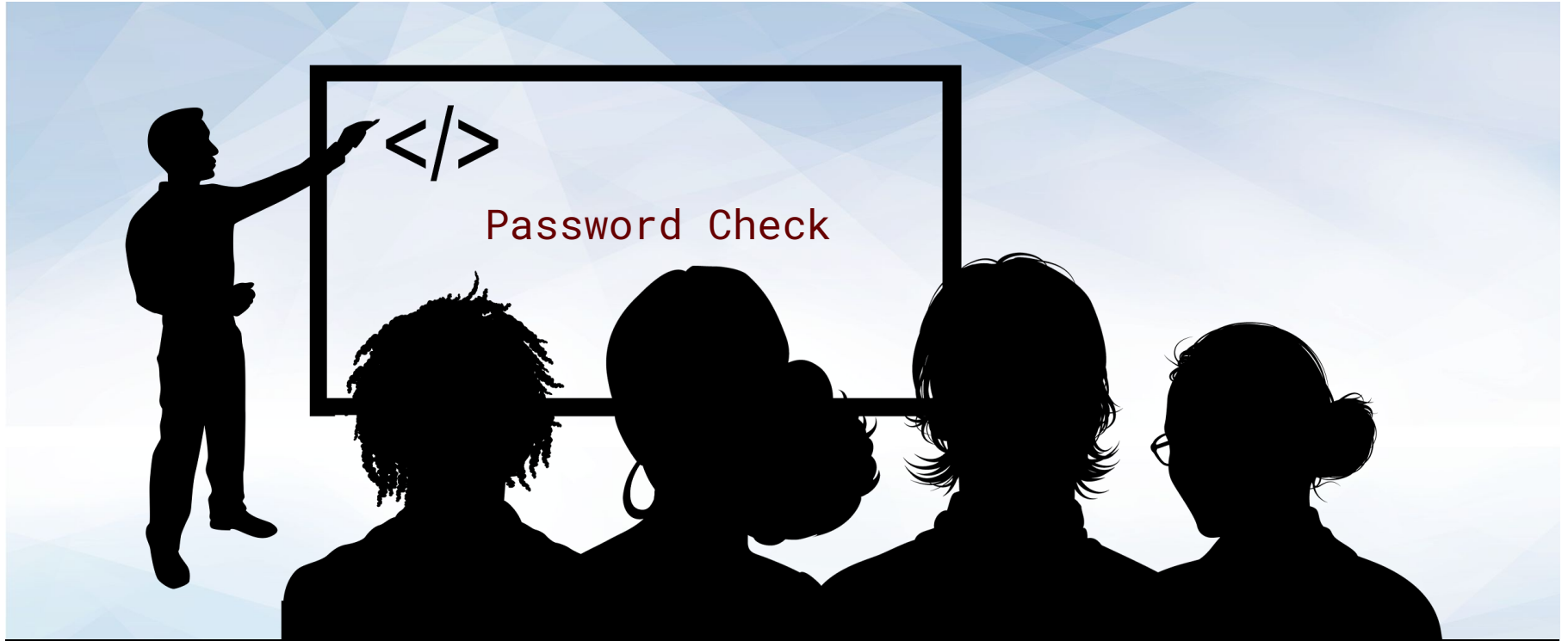


Your Turn: Password Check

Instructions:

Use the **PasswordCheck.py** file to create a command-line application that asks users for their password.

- If the password matches the value stored within the `master_password` variable, alert the user with a message stating: "You have been granted access!"
- If the password does not match the value stored within the `master_password`, alert the user with a message stating: "You have been denied access!"



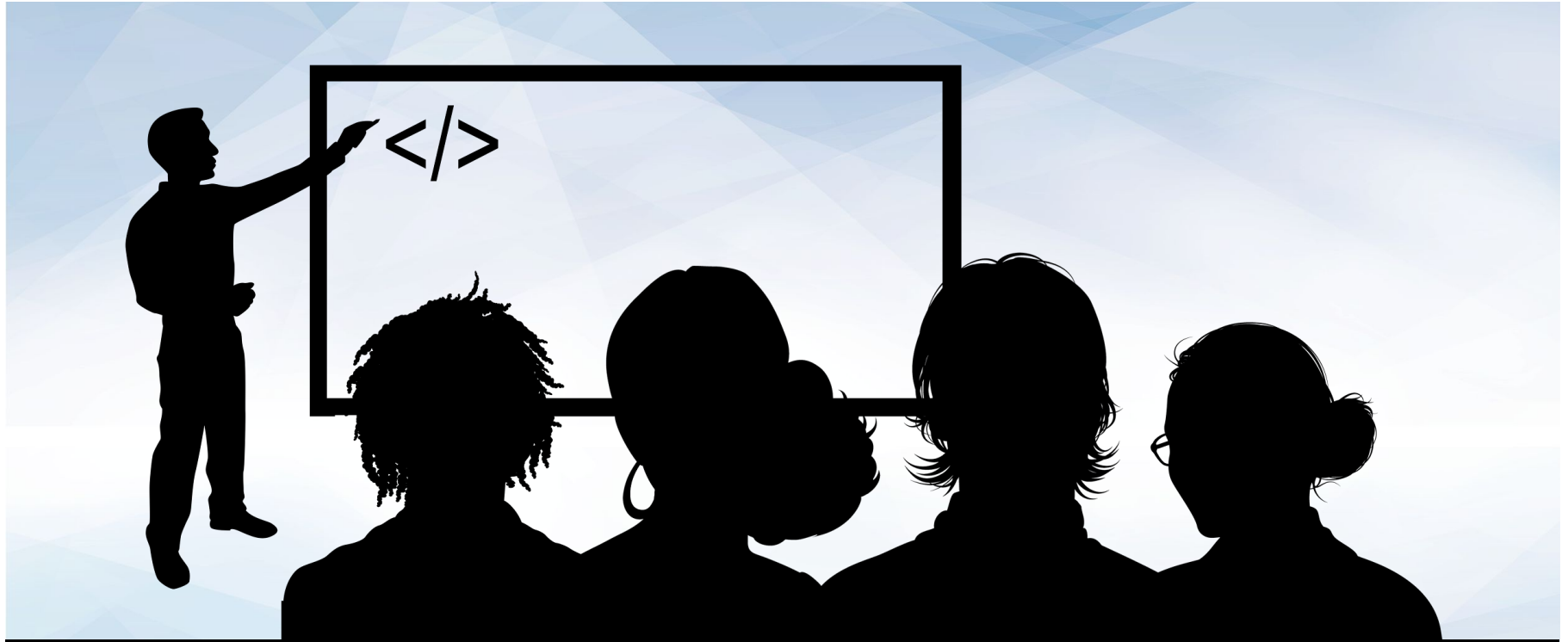
Instructor Demonstration

Password Check Review

Complex Conditionals

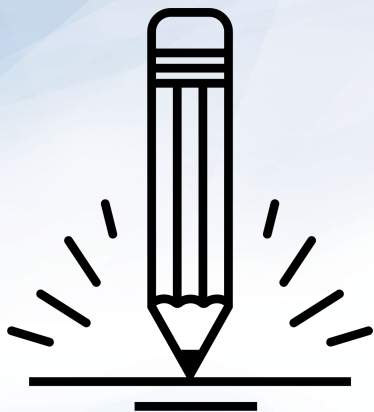


While we can make plenty of applications that run on simple if...else conditionals, there will be times we want our programs to run more **complex** behavior.



Instructor Demonstration

Complex Conditionals



Activity: Bad Bartending

In this activity, you will create a digital bartender.

Instructions sent via Slack

Suggested Time:
15 minutes



Your Turn: Bad Bartending

Instructions:

Set a variable called `drinking_age` to 21.

Prompt the user for their age in years and then check if the user is 21 or older.

- If the user is 21 or older, create a list called `drinks` and store the names of 4 cocktails inside of it. Then prompt the user for the drink they want, check if the user's selection is in the `drinks` list, and print "Cheers!" to the terminal if it is.
- If the user is not 21 or older, print "your fake looks really fake" to the terminal instead.

Use the script file provided to get started. We have provided you with the initial code as well as some important pieces throughout. Use the comments to help you with the rest of the code.

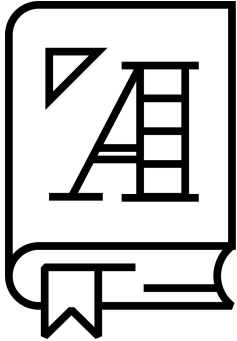


Instructor Demonstration

Bad Bartending Review

Loops

Loops



Loops allow programmers to run a block of code multiple times without having to repeatedly type that block of code.

Why use loops?

For example, let's say...

We have a list of zoo animals and want to print them to the terminal one at a time.

We could write out individual print statements for each element in the list, but that would take up a lot of space and be inefficient.

The next three slides indicate this inefficiency...

Back to The Zoo Pen

List Name: zoo_animals

Zebra

Rhino

Giraffe

Owl

Index 0

Index 1

Index 2

Index 3

Coded in Python using a **List:**

```
zoo_animals = ["Zebra", "Rhino", "Giraffe", "Owl"]
```

Back to The Zoo Pen (Logging)

List Name: zoo_animals

Zebra

Rhino

Giraffe

Owl

Index 0

Index 1

Index 2

Index 3

```
zoo_animals = ["Zebra", "Rhino", "Giraffe", "Owl"]
```

```
print(zoo_animals[0])
```

```
print(zoo_animals[1])
```

```
print(zoo_animals[2])
```

```
print(zoo_animals[3])
```



```
nbart:Unit 3$ python zoo.py
```

```
Zebra
```

```
Rhino
```

```
Giraffe
```

```
Owl
```

99 Loop Problems

What's wrong here?

```
zoo_animals = ["Zebra", "Rhino", "Giraffe", "Owl"]  
  
print(zoo_animals[0])  
print(zoo_animals[1])  
print(zoo_animals[2])  
print(zoo_animals[3])
```



```
nbart:Unit 3$ python zoo.py  
Zebra  
Rhino  
Giraffe  
Owl
```

Repeated Code! We can be more efficient.

Why use loops?

For example, let's say...

We have a list of zoo animals and want to print them to the terminal one at a time.


Instead, we can use a **for loop** in order to iterate through the list one element at a time and run each element through the same block of code...

Loops are critical in programming

Loops allow you to run a block of code multiple times.

Before:


```
zoo_animals = ["Zebra", "Rhino", "Giraffe", "Owl"]  
  
print(zoo_animals[0])  
print(zoo_animals[1])  
print(zoo_animals[2])  
print(zoo_animals[3])
```



```
nbart:Unit 3$ python zoo.py  
Zebra  
Rhino  
Giraffe  
Owl
```

After:

```
zoo_animals = ["Zebra", "Rhino", "Giraffe", "Owl"]  
  
for animal in zoo_animals:  
    print(animal)
```



```
nbart:Unit 3$ python zoo.py  
Zebra  
Rhino  
Giraffe  
Owl
```

For Loop Breakdown



for loops are used to iterate over items of any sequence (e.g., list, string) in order.



For each **iteration** of the loop, the iteration variable `(animal)` gets set to the associated element in the list.



Code “inside” the for loop (indented) is the **code block** that gets executed once for each item in the sequence (4 times here).

```
zoo_animals = ["Zebra", "Rhino", "Giraffe", "Owl"]
```

```
for animal in zoo_animals:  
    print(animal)
```

Sequence

Block of Code

Iteration Variable

Enter the For-Loop

The indented code gets repeated for each item in the list



```
# Start with a list
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]

# Loop through each item of the list
for vegetable in vegetables:
    print("I love" + vegetable)

# Logs:
# I Love Carrots
# I Love Peas
# I Love Lettuce
# I Love Tomatoes
```

Enter the For-Loop

Running the code
“loops” through and
prints each item in
the list.



```
# Start with a list
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]

# Loop through each item of the list
for vegetable in vegetables:
    print("I love" + vegetable)

# Logs:
# I Love Carrots
# I Love Peas
# I Love Lettuce
# I Love Tomatoes
```

Run That For Loop


```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```

For the first iteration, the loop sets `vegetable = "Carrots"`



Run That For Loop

```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```



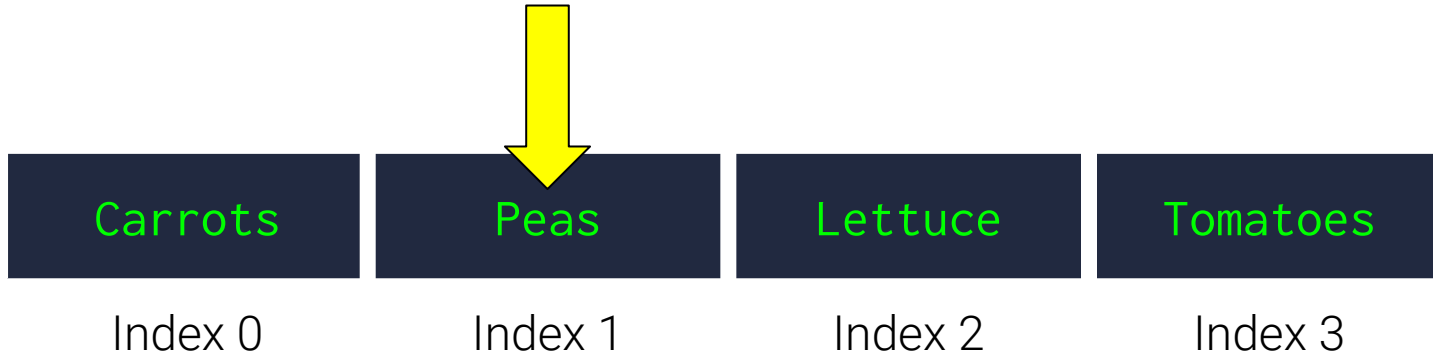
For the first iteration, the loop sets `vegetable = "Carrots"`
So `print("I love Carrots")`



Run That For Loop


```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```

For the second iteration, the loop sets `vegetable = "Peas"`

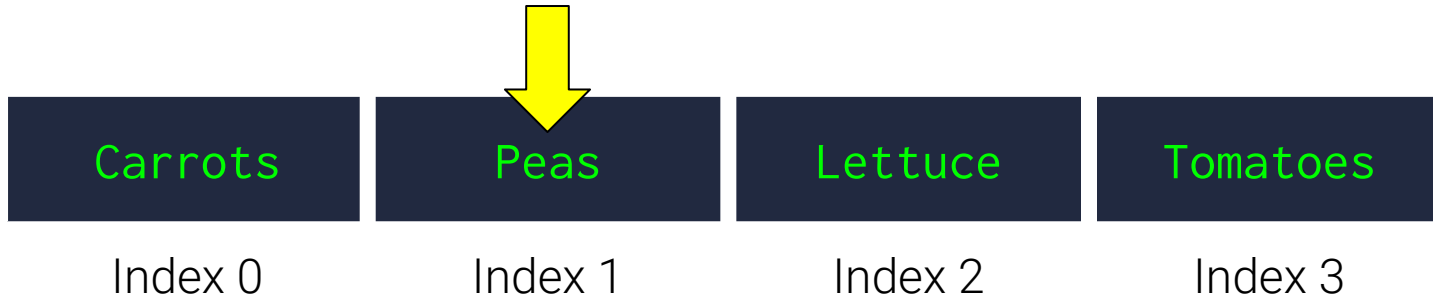


Run That For Loop

```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```



For the second iteration, the loop sets `vegetable = "Peas"`
So `print("I love Peas")`



Run That For Loop


```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```

For the third iteration, the loop sets `vegetable = "Lettuce"`



Run That For Loop

```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```



For the third iteration, the loop sets `vegetable = "Lettuce"`
So `print("I love Lettuce")`



Run That For Loop


```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```

For the fourth iteration, the loop sets `vegetable = "Tomatoes"`



Run That For Loop

```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```

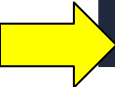


For the fourth iteration, the loop sets `vegetable = "Tomatoes"`
So `print("I love Tomatoes")`

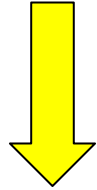


Run That For Loop

```
# Start with a list  
Vegetables = ["Carrots", "Peas", "Lettuce", "Tomatoes"]  
  
# Loop through each item of the list  
for vegetable in vegetables:  
    print("I love" + vegetable)
```

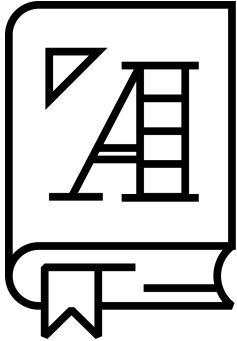


We've now looped through the entire sequence. So the code skips forward to continue running code after the for loop.



Range and Enumerate

Range (x,y)



The **range(x,y)** function allows programmers to loop through a sequence of numbers where **x** is the number to start from and **y** is the number to end before.

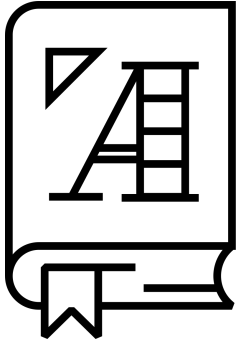
Range (x,y)

```
# range(start, stop) allows you to loop through a range of numbers
# start is the starting value (inclusive), stop is the end value (not inclusive)
for x in range(2, 5):
    print(x)
    # can reference element in list at index x
    print(hobbies[x])
print("-----")
```

Important: The “y” value is **exclusive**, meaning this number will never actually be reached.

The final number reached using the range() function is the number just before the one passed.

Enumerate (x,y)



The **enumerate (x,y)** command allows users to both loop through a list and count the current iteration of said loop.

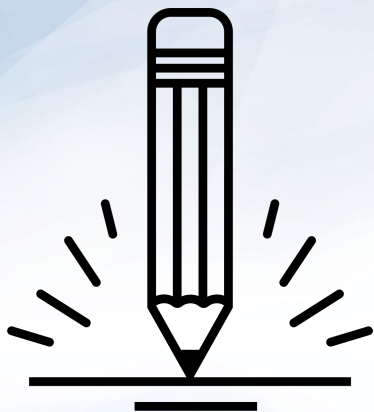
Enumeration

```
# Using enumeration allows programmers to loop through a list and create a counter to reference the current iteration of the loop that they are on
# `index` is an integer which points to the current iteration of the loop
# `hobby` holds the value of the current element in the `hobbies` list
for index, hobby in enumerate(hobbies):
    print(hobby + " is my #" + index + " hobby")
print("-----")

# The enumerate() function also allows programmers to choose what number to start counting from. This means they can count from 1 instead of 0 if they so
desired
for index, hobby in enumerate(hobbies, start=1):
    print(hobby + " is my #" + index + " hobby")
print("-----")
```

Two temporary variables have to be created when using `enumerate()`.

The first variable is an integer that points to the current iteration of the loop while the second variable is the value of the current element in the list being looped through.



Activity: Vulnerable List

In this activity, you will create two `for` loops to navigate through a list of vulnerable IP addresses.

Instructions sent via Slack

Suggested Time:
10 minutes



Your Turn: Vulnerable List

Instructions:

Using the file provided, complete the following:

- Create a loop using `range()` that moves through only the first 5 `IP_addresses` and prints them in order to the screen with their rank.
- Create a loop using `enumerate()` that goes through all the `IP_addresses` and prints out the vulnerability ranking for each one.

Hint: The *ranking* for an IP address is determined by its position in the list. The first element has the ranking of 1, the second has the rank of 2, and so on.



Instructor Demonstration

Vulnerable List Review

Take a Break!



Nested for Loops

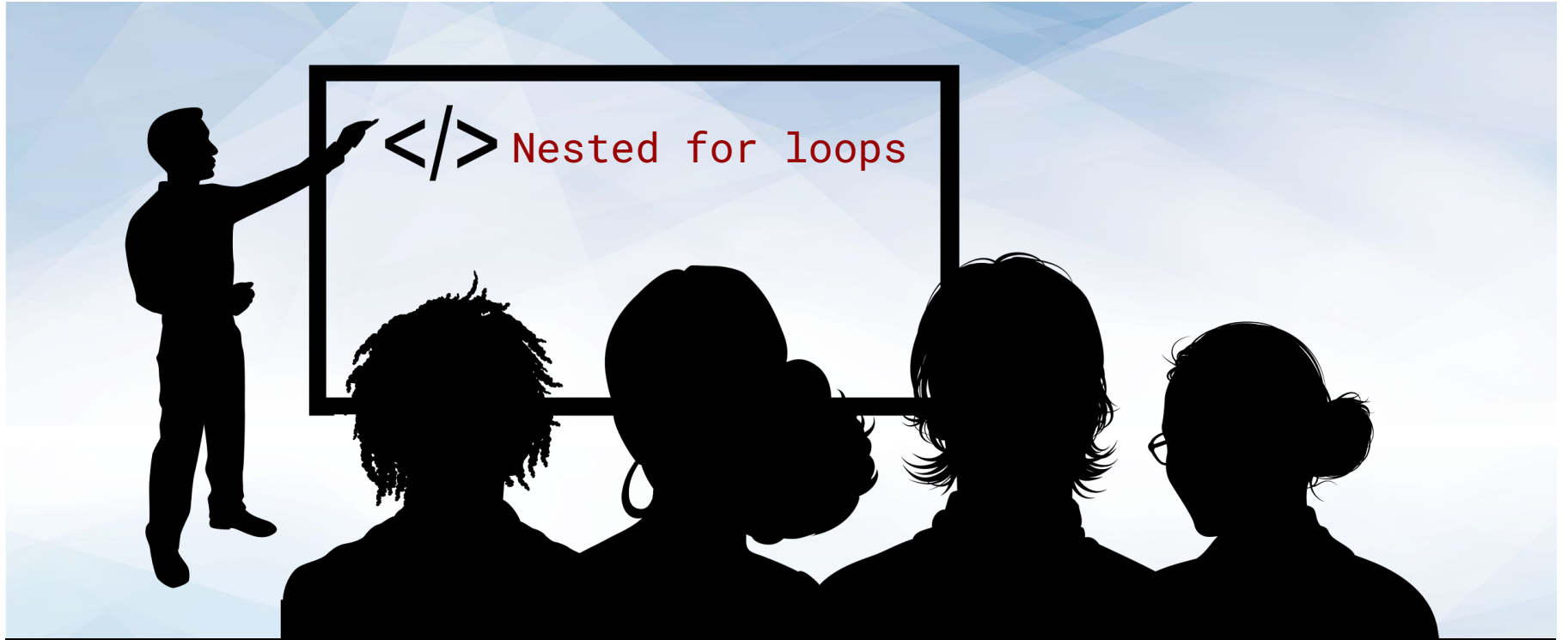
Nested for Loops

While looping through a single list is pretty simple, things get a bit more difficult when multiple lists or more complex data types, like dictionaries, have to be iterated through each loop at the same time.

Nested loops are loops that contain other loops inside of them.

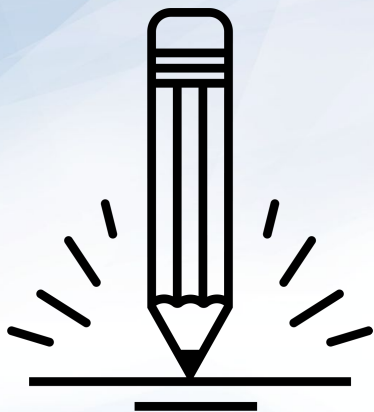
Through the use of **nested for loops**, programmers can easily compare or combine the values of two lists with different lengths.

Dictionaries can also store vast quantities of data within them. Unlike lists, however, the data stored within dictionaries are not stored with numeric indexes and require a bit more effort to loop through.



Instructor Demonstration

Nested For Loops



Activity: Programmer Loop

In this activity, you will be given a short list of dictionaries of people who work as programmers and must print out all the key/value pairs for each programmer using a series of nested for loops.

Instructions sent via Slack.

Suggested Time:
7 minutes



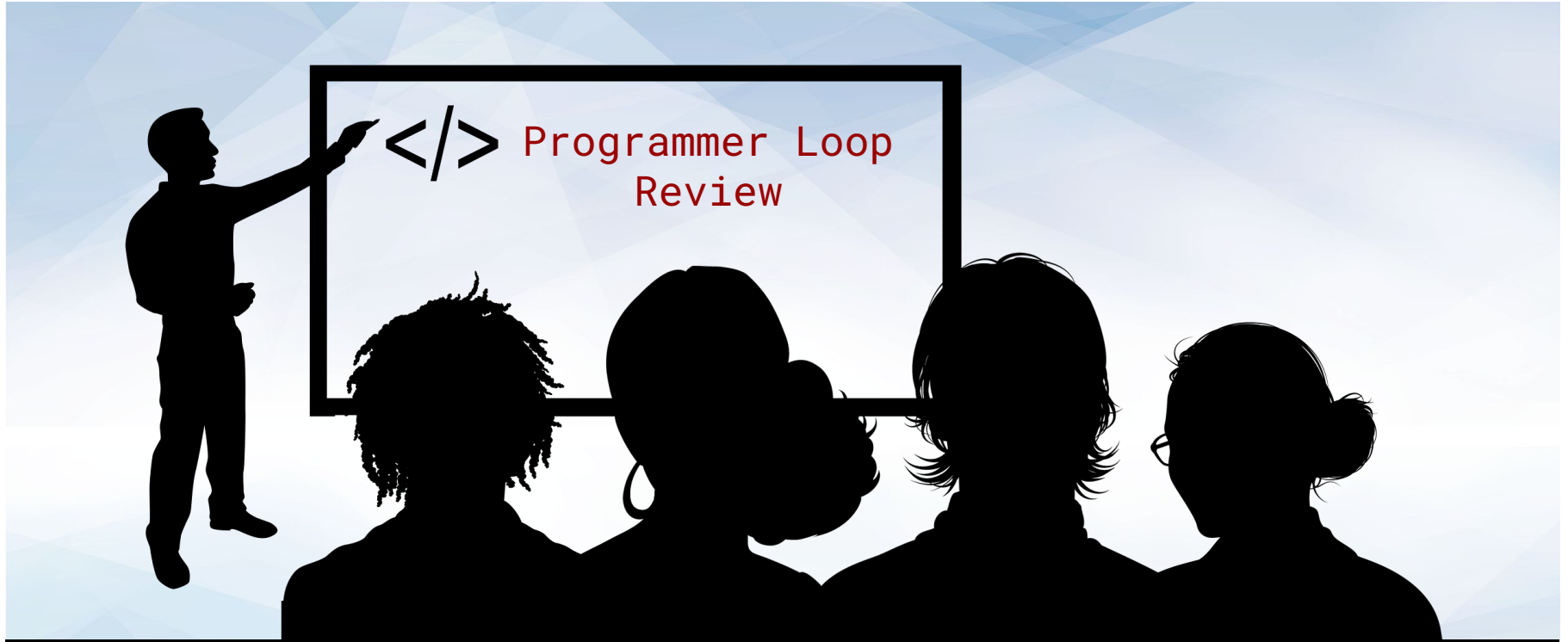
Your Turn: Programmer Loop

Instructions

Using the list of dictionaries in the file provided, do the following:

- Loop through the `programmers_list` one dictionary at a time.
- Loop through each of the dictionary's keys and values.
- Print out each key and its associated value to the terminal.
- Print out a line which will separate each programmer from the next.

We added a lot of the code for you in the script file. Use the comments to help you with the rest.



Instructor Demonstration

Programmer Loop Review

While Loops

For vs While

For loops allow you run a block of code **for** each item in a sequence.

```
sequence = ["a", "b", "c"]
```

```
for item in sequence:  
    print(item)
```



a
b
c

While loops allow you to run **while** some condition is true

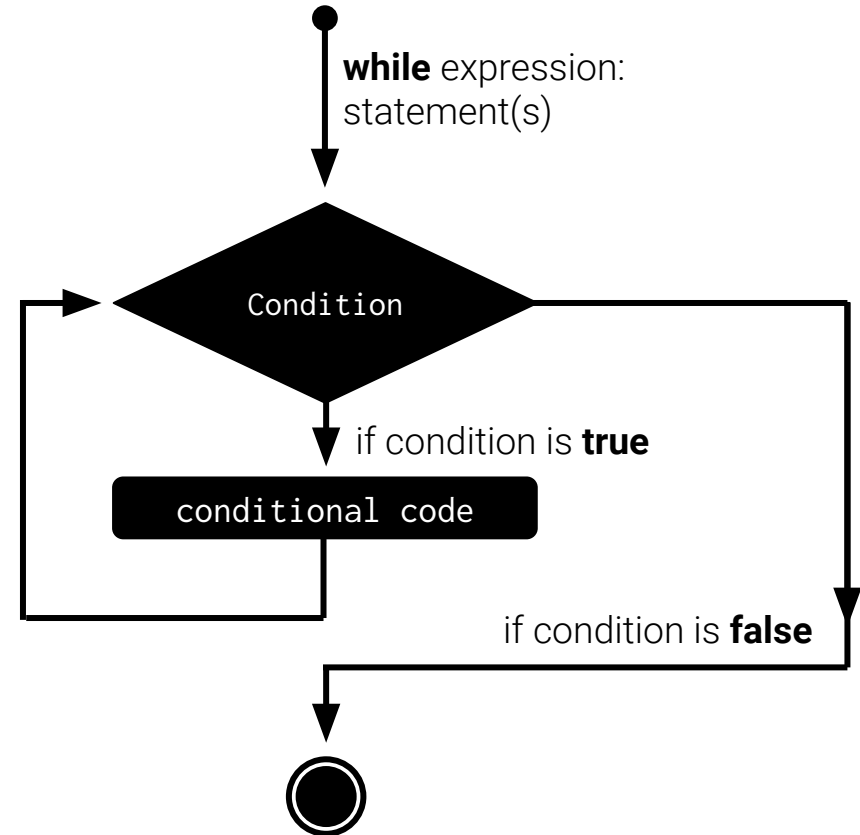
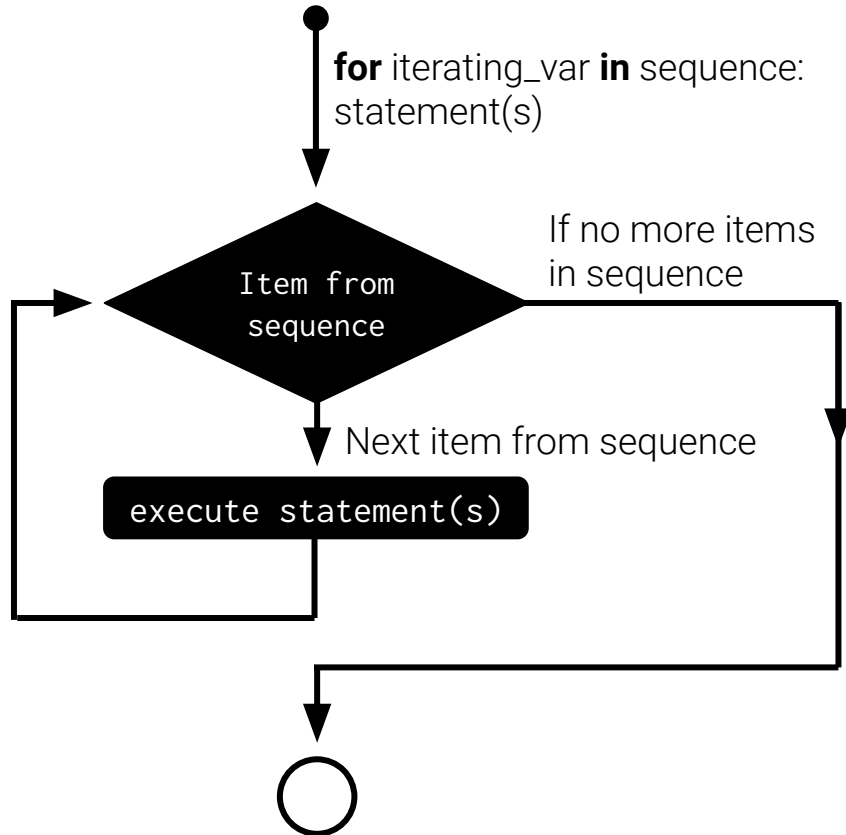
```
i = 0
```

```
while i < 5:  
    print(i)  
    i = i + 1
```



0
1
2
3
4

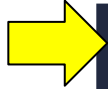
For vs While



Run That While Loop

```
i = 1
```

i starts at 1



```
i = 1
```

```
# while some condition is true, run this block of code
```

```
while i < 5:
```

```
    print(f"I have {i} dollar(s)")
```

```
    i = i + 1
```

```
# Logs:
```

```
# I have 1 dollar(s)
```

Run That While Loop

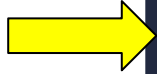
We check the condition to see if the block of code will run.

Because `i < 5`, we will run the block of code.

`i = 1`



`i < 5 -> True`



```
i = 1
```

```
# while some condition is true, run this block of code
```

```
while i < 5:
```

```
    print(f"I have {i} dollar(s)")
```

```
    i = i + 1
```

```
# Logs:
```

```
# I have 1 dollar(s)
```

Run That While Loop

The block of code prints and then **increments** our iterator variable *i*



```
i = 1
```

```
i < 5 -> True
```

```
print
```

```
i = 1
```

```
# while some condition is true, run this block of code
```

```
while i < 5:
```



```
    print(f"I have {i} dollar(s)")
```

```
    i = i + 1
```

```
# Logs:
```

```
# I have 1 dollar(s)
```

Run That While Loop

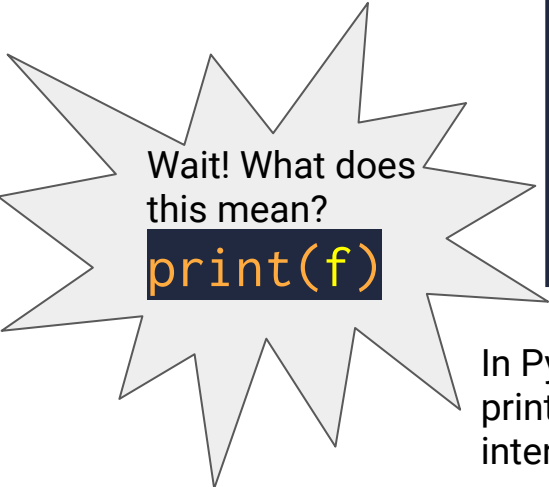
The block of code prints and then **increments** our iterator variable *i*

```
i = 1
```



```
i < 5 -> True
```

```
print
```



Wait! What does
this mean?

```
print(f)
```

```
i = 1
```

```
# while some condition is true, run this block of code
```

```
while i < 5:
```

```
    print(f"I have {i} dollar(s)")
```

```
    i = i + 1
```

```
# Logs:
```

```
# I have 1 dollar(s)
```

In Python, the **f** is a shortcut to format other data types into a string for ease of printing. In this example, we want to print a variable **i** in our string, but we can't interpolate an integer into a string without formatting it. The **f** does that for us.

Run That While Loop

The block of code prints and then **increments** our iterator variable ***i***.



```
i = 1
```

```
i < 5 -> True
```

```
print
```

```
i = i + 1 -> 2
```



```
i = 1
```

```
# while some condition is true, run this block of code
```

```
while i < 5:
```

```
    print(f"I have {i} dollar(s)")
```


```
    i = i + 1
```

```
# Logs:
```


```
# I have 1 dollar(s)
```

Run That While Loop

At the end of the code block, we go back to the top of the while loop.



```
i = 1  
i < 5 -> True  
print  
i = 2
```




```
i = 1  
  
# while some condition is true, run this block of code  
while i < 5:  
    print(f"I have {i} dollar(s)")  
    i = i + 1  
  
# Logs:  
# I have 1 dollar(s)
```


Run That While Loop

Now `i` = 2, which is still less than 5, so the code block gets executed again and `i` is incremented to 3.

This process continues until the condition is False, when `i` \geq 5.



```
i = 2
2 < 5 -> True
print
i = 3
```




```
i = 1

# while some condition is true, run this block of code
while i < 5:
    print(f"I have {i} dollar(s)")
    i = i + 1


# Logs:
# I have 1 dollar(s)
# I have 2 dollar(s)
```

Run That While Loop

This process continues until the condition is False, i.e., when `i >= 5`.



```
i = 3
3 < 5 -> True
print
i = 4
```




```
i = 1

# while some condition is true, run this block of code
while i < 5:
    print(f"I have {i} dollar(s)")
    i = i + 1

# Logs:
# I have 1 dollar(s)
# I have 2 dollar(s)
# I have 3 dollar(s)
```


Run That While Loop

This continues until the condition is False, when `i >= 5`.



```
i = 4
4 < 5 -> True
print
i = 5
```


```
i = 1

# while some condition is true, run this block of code
while i < 5:
    print(f"I have {i} dollar(s)")
    i = i + 1


# Logs:
# I have 1 dollar(s)
# I have 2 dollar(s)
# I have 3 dollar(s)
# I have 4 dollar(s)
```

Run That While Loop

At the end of this loop iteration, **i** will be 5.



```
i = 4
4 < 5 -> True
print
i = 5
```



```
i = 1

# while some condition is true, run this block of code
while i < 5:
    print(f"I have {i} dollar(s)")
    i = i + 1

# Logs:
# I have 1 dollar(s)
# I have 2 dollar(s)
# I have 3 dollar(s)
# I have 4 dollar(s)
```

Run That While Loop

Now `i` is 5, which is **not** less than 5, so the code block in the while loop won't get executed again.

At this point, the code would continue running any code after the while loop and our loop iteration is finished.

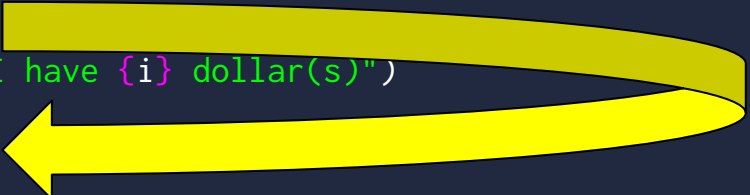
`i = 5`

 `5 < 5 - >False`

```
i = 1

# while some condition is true, run this block of code
while i < 5:
    print(f"I have {i} dollar(s)")
    i = i + 1

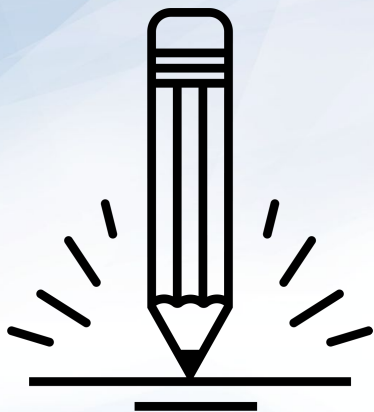
# Logs:
# I have 1 dollar(s)
# I have 2 dollar(s)
# I have 3 dollar(s)
# I have 4 dollar(s)
```





Instructor Demonstration

While Loops



Activity: The Number Chain

In this activity, you will build a command-line game that will ask the user for a number and then will print all the numbers from 0 up until that number.

Suggested Time:
15 minutes



Your Turn: The Number Chain

Instruction:

Ask the user "How many numbers?" and then print out a chain of ascending numbers from 0 up to, but not including, the number input.

After the results have printed, ask the user if they would like to continue. If "y" is entered, keep the chain running by inputting a new number and starting a new count from 0 to the number input. If "n" is entered, exit the application.

We added some of the initial code for you in the script file. Use the comments to help you with the rest.

Hint: You will need to use both `for` and `while` loops for this activity. We provided the start of the `while` loop for you in the script file.

Bonus: Rather than just displaying numbers starting at 0, have the numbers begin at the end of the previous chain.



Instructor Demonstration

The Number Chain Review



Challenge: Kid in a Candy Store

In this challenge, pretend you're a kid going to the candy store with your parents. After pestering your parents for a while, they finally let you pick out some candy to take home.

Instructions sent via Slack

Suggested Time:
20 minutes





Instructor Demonstration

Kid in a Candy Store Review

Python topics we've covered so far:

Data	Logic
1. Numbers	1. Operators
2. Strings	2. Conditionals
3. Booleans	3. Loops
4. Lists	4. Functions
5. Dictionaries	5. Modules

Class Objectives

By the end of class today, you will be able to:

- ✓ Use simple and complex conditionals to create branching paths for programs to execute.
- ✓ Build a command-line application using conditionals and user input.
- ✓ Use `for` and `while` loops to iterate through lists and dictionaries to perform basic operations on collections of data.
- ✓ Use the `range` function to loop through a list with defined ranges.
- ✓ Use the `enumerate` function to loop through a list with a built-in counter.
- ✓ Use `key`, `value`, and `item` methods with loops to convert data from dictionaries into lists.



Questions?