Introduction to Python

Session Two

Lorain Public Library System - Avon Branch
October 17, 2023

Agenda

- Brief Python Overview
- Getting Started
- Python Features/Concepts
  - Variables
  - Data Types
    i. Strings
    ii. Numbers
    iii. Collections (Lists)
  - User Input
  - Loops
    i. For Loops
    ii. While Loops
  - Conditionals
  - Formatting
  - Code Comments

Core Concepts:

for Loops
while Loops
Conditionals
Formatting
Code Comments

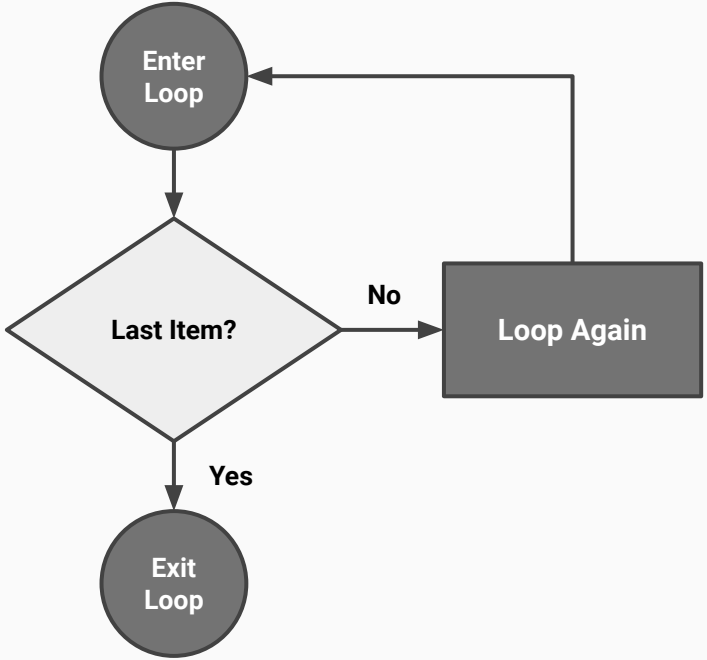
Core Concepts: **Loops**

- Loops are important because they’re one of the most common ways a computer automates repetitive tasks
- When you’re using loops for the first time, keep in mind that the set of steps is repeated once for each item in the list, no matter how many items are in the list
- If you have a million items in your list, Python repeats these steps a million times—and usually very quickly
- There are two ways to create loops in Python: with the **for-loop** and the **while-loop**

Core Concepts: **for Loops**

- **for-loops** are used when you have a block of code which you want to repeat a fixed number of times
- The **for-loop** is always used in combination with an iterable object, like a list
- When you want to do the same action with every item in a list, you can use a **for-loop**

Core Concepts: **for Loops**



Core Concepts: **for Loops**

- Say we have a list of toppings, and we want to print out each topping in the list
- We could do this by retrieving each topping from the list individually, but this approach could cause several problems
- For one, it would be repetitive to do this with a long list of toppings
- Also, we'd have to change our code each time the list's length changed
- Using a for loop avoids both of these issues by letting Python manage these issues internally

Core Concepts: **for Loops**

```
toppings = ['pepperoni', 'sausage', 'cheese']
for topping in toppings: # Temporary variable name; Add ":"
    print(toppings)      # Indent 4 spaces
```

- We begin by defining a list, then we define a **for loop**
- This line tells Python to pull a topping from the list **toppings**, and associate it with the temporary variable **topping**
- Next, we tell Python to print the topping that's just been assigned to **topping**
- Python then repeats these last two lines, once for each topping in the list
- It might help to read this code as "*For every topping in the list of toppings, print the topping's topping.*"

Core Concepts: **for Loops**

- Let's build on the previous example by printing a message for each topping, telling them how much you like each topping:

```
toppings = ['pepperoni', 'sausage', 'cheese']

for topping in toppings:
    print(f"{topping.title()} is the best!")
```

Core Concepts: **for Loops**

- What happens once a for loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish
- Any lines of code after the for loop that are not indented are executed once without repetition
- Let's write a thank you to the group of toppings as a whole, thanking the chef for a great pizza …

Core Concepts: **for Loops**

- To display this group message after all of the individual messages have been printed, we place the thank you message after the for loop, without indentation

```
toppings = ['pepperoni', 'sausage', 'cheese']

for topping in toppings:
    print(f"{topping.title()} is the best!")
    print(f"I can't wait to try a {topping.title()} pizza.\n")

print("Thank you chef, that pizza was great!\n")
```

Core Concepts: **for Loops**

```
Pepperoni is the best!
I can't wait to try a Pepperoni pizza.

Sausage is the best!
I can't wait to try a Sausage pizza.

Cheese is the best!
I can't wait to try a Cheese pizza.

Thank you chef, that pizza was great!
```

Core Concepts: **for Loops**

- Keep in mind when writing your own for loops that you can choose any name you want for the **temporary variable** that will be associated with each value in the list
- However, it's helpful to choose a meaningful name that represents a single item from the list
- For example, here's a good way to start a for loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

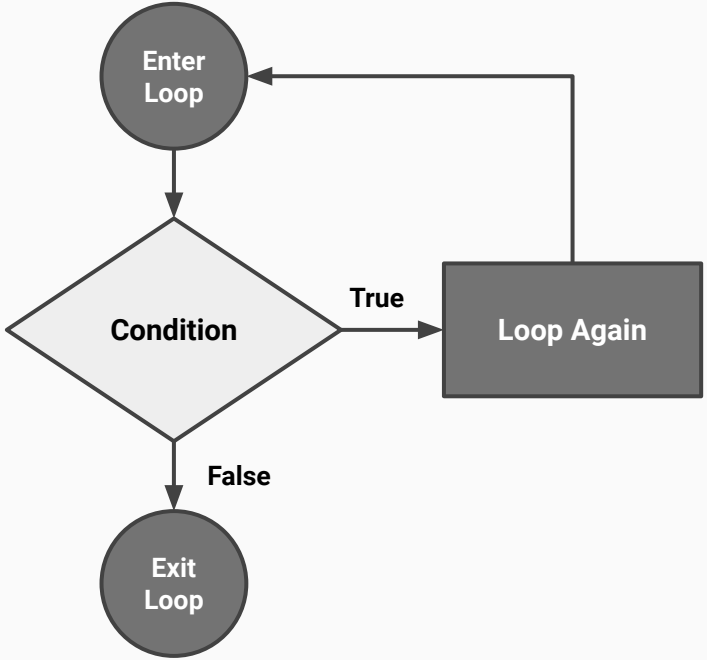
Core Concepts:

for Loops
while Loops
Conditionals
Formatting
Code Comments

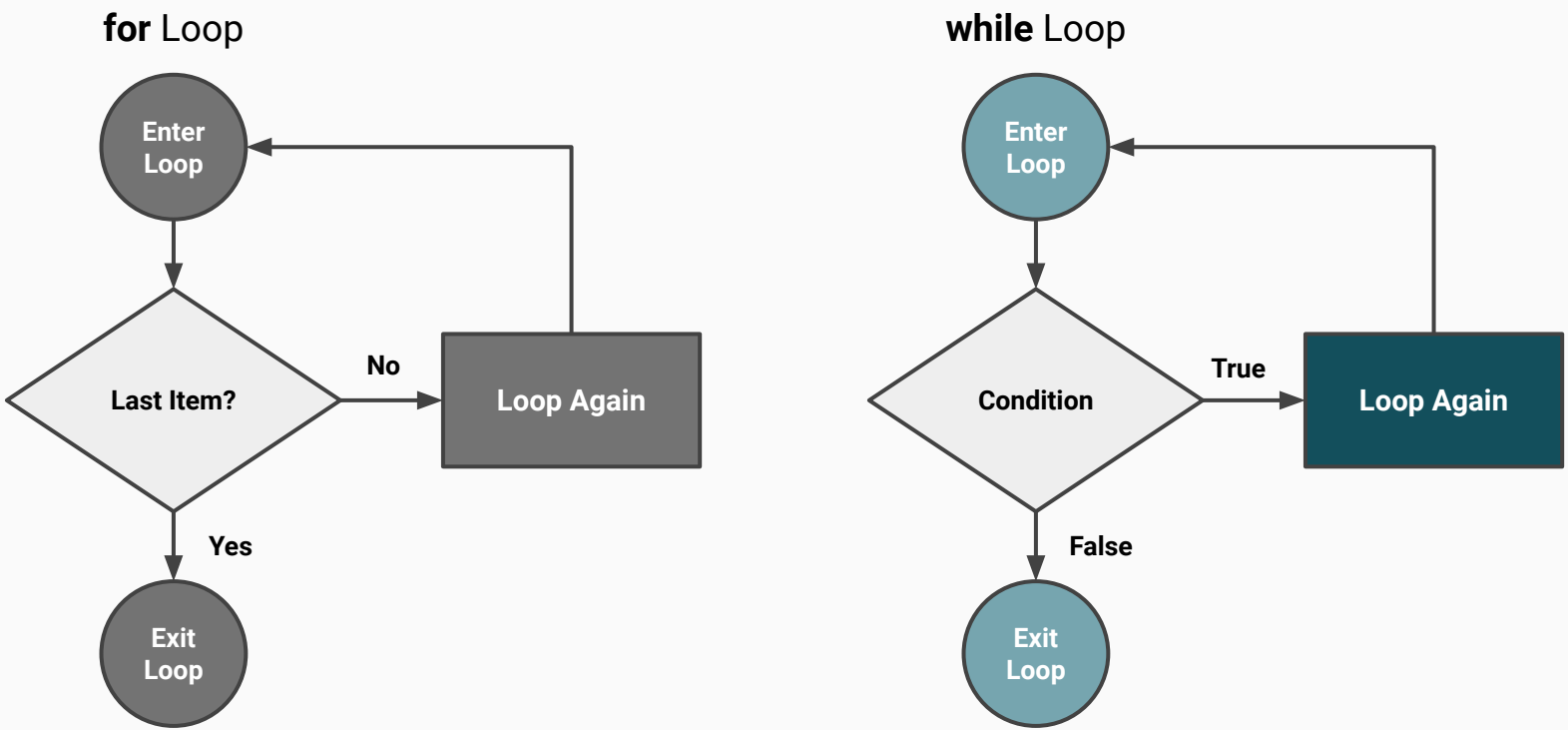
Core Concepts: **while Loops**

- The **for loop** takes a collection of items and executes a block of code once for each item in the collection
- In contrast, the **while loop** runs as long as, or while, a certain condition is **True**

Core Concepts: **while Loops**



Core Concepts: **while Loops**



Core Concepts: **while Loops**

- You can use a while loop to count up through a series of numbers
- For example, the following while loop counts from 1 to 5:

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

- The last line adds 1 to that value, it can also be written as:

```
current_number = current_number + 1
```

Core Concepts: **while Loops**

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

- Output:

```
1
2
3
4
5
```

Core Concepts: **while Loops**

- The programs you use every day most likely contain **while loops**
- For example, a game needs a **while loop** to keep running as long as you want to keep playing, and so it can stop running as soon as you ask it to quit
- Programs wouldn’t be fun to use if they stopped running before we told them to or kept running even after we wanted to quit, so **while loops** are quite useful

Core Concepts: **while Loops**

- We can make the program run as long as the user wants by putting most of the program inside a while loop
- We'll define a **quit** value and then keep the program running as long as the user has not entered the quit value:

```
prompt = "\nTell me something, and I will repeat it back to
you or enter 'quit' to end the program."
prompt += "\nEnter something: "
message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

Core Concepts: **while Loops**

Let’s switch gears for a moment and discuss **conditionals …**

Core Concepts: **while Loops**

- In the previous examples, we had the program perform certain tasks while a given condition was **True**
- But what about more complicated programs in which many different events could cause the program to stop running?
- For example, in a game, several different events can end the game. When the player runs out of ships, their time runs out, or the cities they were supposed to protect are all destroyed, the game should end
- It needs to end if any one of these events happens. If many possible events might occur to stop the program, trying to test all these conditions in one **while** statement becomes complicated and difficult

Core Concepts: **while Loops**

- For a program that should run only as long as many conditions are **True**, you can define one variable that determines whether or not the entire program is active
- This variable, called a **flag**, acts as a signal to the program
- We can write our programs so they run while the flag is set to **True** and stop running when any of several events sets the value of the flag to **False**
- As a result, our overall while statement needs to check only one condition: whether the flag is currently **True**
- Then, all our other tests (to see if an event has occurred that should set the flag to False) can be neatly organized in the rest of the program

Core Concepts: **while Loops**

- Let's add a flag to the code from the previous section. This flag, which we'll call **active** will monitor whether or not the program should continue:

```
prompt = "\nTell me something, and I will repeat it back to
you or enter 'quit' to end the program."
prompt += "\nEnter something: "
active = True
while active:
    message = input(prompt)
    if message == 'quit':
        active = False
    else:
        print(message)
```

Core Concepts: **while Loops**

- To exit a while loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the break statement

```
prompt = "\nPlease enter the name of a city you have visited.
Enter 'quit' when you are finished: "

while True:
    city = input(prompt)
    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

Core Concepts: **Nested Loops**

- A **nested loop** is a loop inside the body of the outer loop
- The **inner** or **outer loop** can be any type, such as a **while loop** or **for-loop**
- For example, the **outer for-loop** can contain a **while loop** and vice versa
- The **outer loop** can contain more than one **inner loop**
- There is no limitation on the chaining of loops

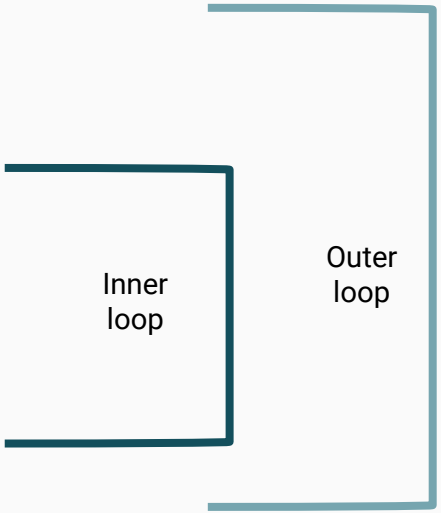
Core Concepts: **Nested Loops**

```
toppings = ['Pepperoni', 'Sausage', 'Peppers']

# outer for-loop
for topping in toppings:

# inner while loop
    count = 0
    while count < 3:
        print(topping)

# increment counter
        count = count + 1
    print()
```



Core Concepts: **Nested Loops**

```
toppings = ['Pepperoni', 'Sausage', 'Peppers']

# outer for-loop
for topping in toppings:

# inner while loop
    count = 0
    while count < 3:
        print(topping)

# increment counter
        count = count + 1
    print()
```

Output:

```
Pepperoni
Pepperoni
Pepperoni

Sausage
Sausage
Sausage

Peppers
Peppers
Peppers
```

Core Concepts:

for Loops
while Loops
Conditionals
Formatting
Code Comments

Core Concepts: **Conditionals**

- Programming often involves examining a set of conditions and deciding which action to take based on those conditions
- Python's **if** statement combined with **Conditional** and **Logical operators** allows you to examine the current state of a program and respond appropriately to that state

Core Concepts: **Conditionals**

- At the heart of every **if** statement is an expression that can be evaluated as **True** or **False** and is called a **conditional test**
- Python uses the values **True** and **False** to decide whether the code in an **if** statement should be executed
- If a conditional test evaluates to **True**, *Python executes the code following the **if** statement*
- If the test evaluates to **False**, *Python ignores the code following the **if** statement*

Core Concepts: **Comparison Operators**

- Python Comparison Operators:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Core Concepts: **Comparison Operators**

- Most conditional tests compare the current value of a variable to a specific value of interest
- The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
>>> car = `bmw`
>>> car == `bmw`
True
```

Core Concepts: **Comparison Operators**

```
>>> car = 'bmw'
>>> car == 'bmw'
True
```

- The first line sets the value of car to `**bmw**` using a single equal sign, as you've seen many times already
- The next line checks whether the value of car is `**bmw**` by using a double equal sign (**==**)
- This equality operator returns **True** if the values on the left and right side of the operator match, and **False** if they don't match. The values in this example match, so Python returns **True**

Core Concepts: **Comparison Operators**

- When the value of car is anything other than `**bmw**`, this test returns **False**:

```
>>> car = `audi`
>>> car == `bmw`
False
```

Core Concepts: **Comparison Operators**

```
>>> car = `audi`
>>> car == `bmw`
False
```

- A **single equal sign** is really a statement; you might read the first line of code as:
  - **“Set the value of car equal to `audi`.”**
- On the other hand, a **double equal sign** asks a question:
  - **“Is the value of car equal to `bmw`?”**
- Most programming languages use equal signs in this way

Core Concepts: **Comparison Operators**

- Testing for equality is **case sensitive** in Python
- For example, two values with different capitalization are not considered equal:

```
>>> car = `Audi`
>>> car == `audi`
False
```

Core Concepts: **Comparison Operators**

- If case matters, this behavior is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

- This test will return **True** no matter how the value `**Audi**` is formatted because the test is now case insensitive

Core Concepts: **Comparison Operators**

- When you want to determine whether two values are not equal, you can use the inequality operator (**!=**)
- We'll store a **requested_pizza** topping in a variable and then print a message if the person did not order anchovies:

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

Core Concepts: **Comparison Operators**

```
requested_topping = 'mushrooms'
if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

- This code compares the value of **requested_topping** to the value `**anchovies**`
- If these two values do not match, Python returns **True** and executes the code following the **if** statement
- If the two values match, Python returns **False** and does not run the code following the **if** statement

Core Concepts: **Logical Operators**

- You may want to check multiple conditions at the same time
- For example, sometimes you might need two conditions to be **True** to take an action
- Other times, you might be satisfied with just one condition being **True**
- The keywords **and** and **or** can help you in these situations

Core Concepts: **Logical Operators**

There are three possible **logical operators** in Python:

- **and** returns **True** *if both statements are true*
- **or** returns **True** *if at least one of the statements is true*
- **not** reverses the Boolean value:
    - returns **False** *if the statement is true*, and **True** *if the statement is false*

Core Concepts: **Logical Operators**

- To check whether two conditions are both **True** simultaneously, use the keyword **and** to combine the two conditional tests
- If each test passes, the overall expression evaluates to **True**
- If either test fails or if both tests fail, the expression evaluates to **False**

```
temperature = 78
humidity = 90

if (temperature > 75) and (humidity > 85): # () are a best practice
    print("Hot and humid")
Hot and humid
```

Core Concepts: **Logical Operators**

- Remember, **or** returns **True** *if at least one of the statements is true*

```
temperature = 75
humidity = 90

if (temperature > 80) or (humidity > 85):
    print("Hot, humid or both!")
Hot, humid or both!
```

Core Concepts: **Logical Operators**

- Remember, **not** reverses the Boolean value
- We can use it to test if the temperature is colder (i.e. not hotter) than a threshold:

```
temperature = 45

if not temperature > 50:
    print("Cool")
Cool
```

Core Concepts: **Conditionals**

- Sometimes it's important to check whether a list contains a certain value before taking an action
- For example, you might want to check whether a pizza topping is available in a list of current toppings before completing someone's order:

```
>>> requested_toppings = ["mushrooms", "onions", "pineapple"]
>>> "mushrooms" in requested_toppings
True

>>> "pepperoni" in requested_toppings
False
```

Core Concepts: **Conditionals**

- Often, you'll want to take one action when a conditional test passes and a different action in all other cases
- Python's **if-else** syntax makes this possible
- An **if-else** block is similar to a simple **if** statement, but the **else** statement allows you to define an action or set of actions that are executed when the conditional test fails

Core Concepts: **Conditionals**

- We'll display the same message we had previously, but this time we'll add a message for anyone not old enough to vote:

```
age = 17

if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

- If the conditional test passes (**True**), the first block of indented **print()** calls is executed. If the test evaluates to **False**, the **else** block is executed

Core Concepts: **Conditionals**

- Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `**if-elif-else**` syntax
- Python executes only one block in an `**if-elif-else**` chain. It runs each conditional test in order, until one passes
- When a test passes, the code following that test is executed and Python skips the rest of the tests

Core Concepts: **Conditionals**

- Many real-world situations involve more than two possible conditions
- For example, consider an amusement park that charges different rates for different age groups:

```
Admission for anyone under age 4 is free
Admission for anyone between the ages of 4 and 18 is $25
Admission for anyone age 18 or older is $40
```

Core Concepts: **Conditionals**

```
Admission for anyone under age 4 is free
Admission for anyone between the ages of 4 and 18 is $25
Admission for anyone age 18 or older is $40
```

- The following code tests for the age group of a person and then prints an admission price message:

```
age = 12
if age < 4:
    print("Your admission cost is $0.")
elif age < 18:
    print("Your admission cost is $25.")
else:
    print("Your admission cost is $40.")
```

Core Concepts: **Conditionals**

- The `**if-elif-else**` chain is powerful, but it's only appropriate to use when you just need **one test to pass**
- As soon as Python finds one test that passes, it skips the rest of the tests. This behavior is beneficial, because it's efficient and allows you to test for one specific condition
- However, sometimes it's important to check all conditions of interest. In this case, you should use a series of simple **if** statements with no `**elif**` or **else** blocks
- This technique makes sense when more than one condition could be **True**, and you want to act on every condition that is **True**

Core Concepts: **Conditionals**

- Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

Core Concepts: **Conditionals**

- This code would not work properly if we used an `**if-elif-else**` block, because the code would stop running after only one test passes:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

Core Concepts: **Conditionals**

- In summary:
  - If you want only one block of code to run, use an `**if-elif-else**` chain
  - If more than one block of code needs to run, use a series of independent `**if**` statements

Core Concepts:

for Loops
while Loops
Conditionals
Formatting
Code Comments

Core Concepts: **Formatting**

- Python uses indentation to determine how a line, or group of lines, is related to the rest of the program
- Python's use of indentation makes code very easy to read
- Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure
- In longer Python programs, you'll notice blocks of code indented at a few different levels
- These indentation levels help you gain a general sense of the overall program's organization

Core Concepts: **Formatting**

- You'll need to watch for a few common indentation errors
- For example, people sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented
- Seeing examples of these errors now will help you avoid them in the future and correct them when they do appear in your own programs

Core Concepts: **Formatting**

- Always indent the line after the `**for**` statement in a loop. If you forget, Python will remind you:

```
toppings = [‘pepperoni’, ‘sausage, ‘peppers’]
for topping in toppings:
print(topping)
```

Core Concepts: **Formatting**

```
toppings = [‘pepperoni’, ‘sausage, ‘peppers’]
for topping in toppings:
print(topping)
```

- The call to `**print()**` should be indented, but it’s not. When Python expects an indented block and doesn’t find one, it lets you know which line it had a problem with:

```
 File "toppings.py", line 3
    print(topping)
    ^
IndentationError: expected an indented block after `for` statement on line 2
```

Core Concepts: **Formatting**

```
message = "Hello Python world!"
    print(message)
```

- We don't need to indent the `**print()**` call, because it isn't part of a loop; hence, Python reports that error:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

Core Concepts: **Formatting**

- The colon at the end of a for statement tells Python to interpret the next line as the start of a loop.

```
toppings = [‘pepperoni’, ‘sausage, ‘peppers’]
for topping in toppings
    print(topping)
```

- If you accidentally forget the colon, you’ll get a syntax error because Python doesn’t know exactly what you’re trying to do:

```
 File "toppings.py", line 2
    for topping in toppings
                           ^
SyntaxError: expected `:`
```

Core Concepts:

for Loops
while Loops
Conditionals
Formatting
Code Comments

Core Concepts: **Code Comments**

- Comments are an extremely useful feature in most programming languages
- Everything you've written in your programs so far is Python code
- As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving
- A comment allows you to write notes in your spoken language, within your programs

Core Concepts: **Code Comments**

- In Python, the hash mark (#) indicates a comment
- Anything following a hash mark in your code is ignored by the Python interpreter. For example:

```
# Say hello to everyone.
print("Hello Python people!")
```

- Python ignores the first line and executes the second line

```
Hello Python people!
```

Core Concepts: **Code Comments**

- The primary reason to write comments is to explain what your code is supposed to do and how you are making it work
- When you’re in the middle of working on a project, you understand how all of the pieces fit together
- But when you return to a project after some time away, you’ll likely have forgotten some of the details
- You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach clearly

Review

Resources

- Python Documentation
- Python Cheat Sheets
- Python Beginners Guides
- Code Academy, Coursera, edX, Free Code Camp, Udemy, etc

Contact me:

Scott Seighman
scottseighman@gmail.com

Final Project

Final Project: **Pizza Ordering Script**

- As a ‘final project’, we’ll write a Python script for ordering pizza!
- Prompt the user asking if they want to order a pizza
  - If Yes, ask for toppings until finished (append to the list)
  - If No, leave an exit message
- A plain pizza is $5.00 and each topping is $.75
  - Display how many toppings were added (Hint: You’ll need the length of the list)
  - Calculate total price (including toppings) and display total