Introduction to Python

Session One

Lorain Public Library System - Avon Branch
October 10, 2023

Agenda

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

Expectations

- New to Python
- Cover core Python concepts
- Combination of slides, discussion and hands-on
- Ask questions!

Python Overview

Brief Python Overview

- Python is a general-purpose, high-level programming language
- Interpreted, meaning that it does not need to be compiled into machine code before it can be run
- Python is known for its simple syntax and readability, making it a popular choice for beginners and experienced programmers alike
- Python was created in the late 1980s by Guido van Rossum, a Dutch computer scientist
  - Named the language after the British comedy troupe Monty Python
- Python was first released in 1991, and it quickly gained popularity among programmers

Brief Python Overview

Python use cases:

- **Web development:** Python can be used to develop both the front-end and back-end of web applications
- **Software development:** Python can be used to develop desktop applications, mobile applications, and server-side applications
- **Data science and machine learning:** Python is a popular choice for data science and machine learning tasks
- **Automation:** Python can be used to automate a variety of tasks, such as system administration, data processing, and web scraping

Brief Python Overview

Advantages of using Python:

- **Simple syntax:** Python has a simple and readable syntax, making it easy to learn and use
- **Versatility:** Python can be used for a wide variety of tasks, from web development to data science to automation
- **Power:** Python is a powerful language that can be used to develop complex applications
- **Large community:** Python has a large and active community of users and developers. This means that there are a number of resources available to help you learn and use the language

Getting Started with Python

Getting Started

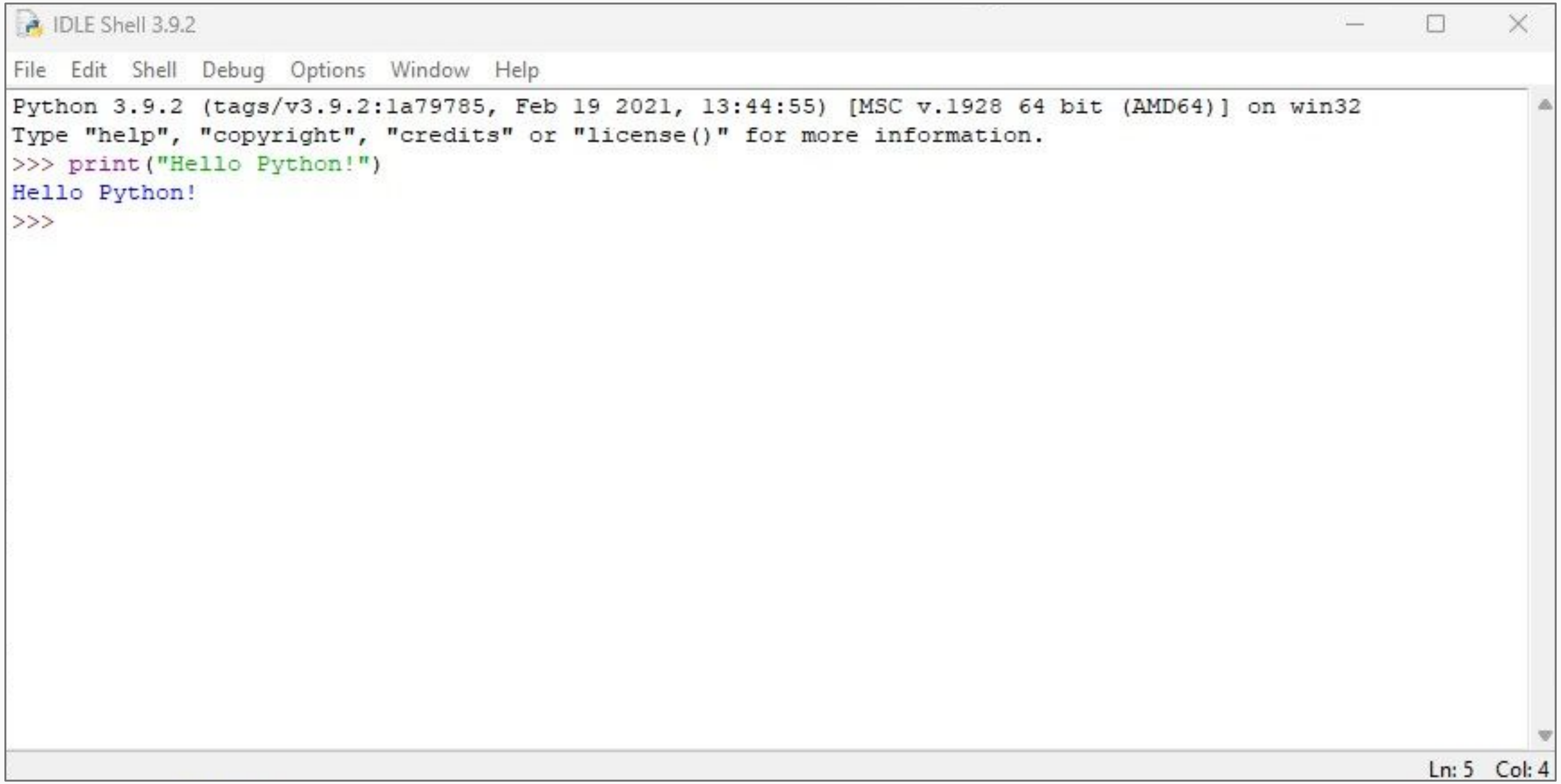
- Installing Python is generally easy, and many Linux and UNIX distributions include Python
- Even some Windows computers now come with Python already installed
- If you need to install Python and aren't confident about the steps, you can find a notes on the BeginnersGuide/Download wiki page, but installation is straightforward on most platforms

Getting Started

Python Development Tools

- Every Python installation comes with an **Integrated Development and Learning Environment**, which you'll see shortened to **IDLE**
- These are a class of applications that help you write code more efficiently
- While there are many IDEs for you to choose from, Python IDLE is very basic, which makes it the perfect tool for a beginning programmer
- Python IDLE is included in Python installations on Windows and Mac
  - If you're a Linux user, then you should be able to find and download Python IDLE using your package manager

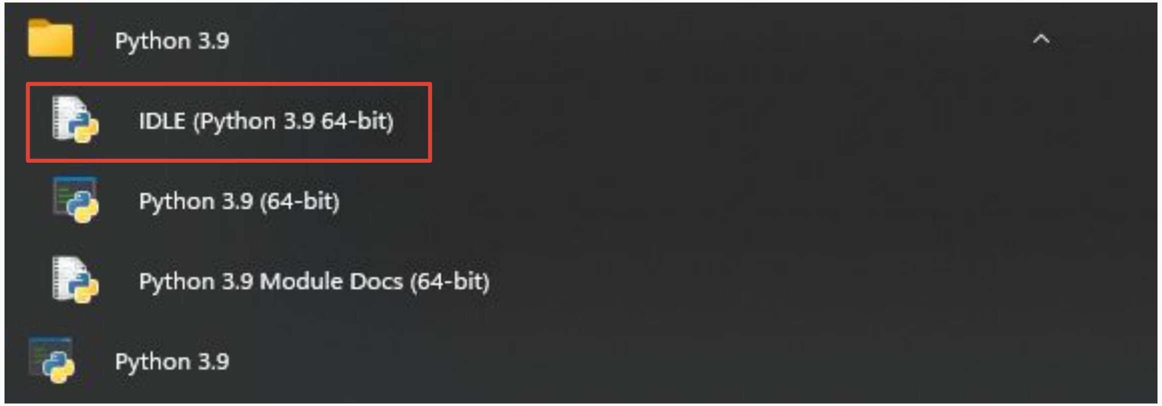
Getting Started



```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello Python!")
Hello Python!
>>>
```

Getting Started

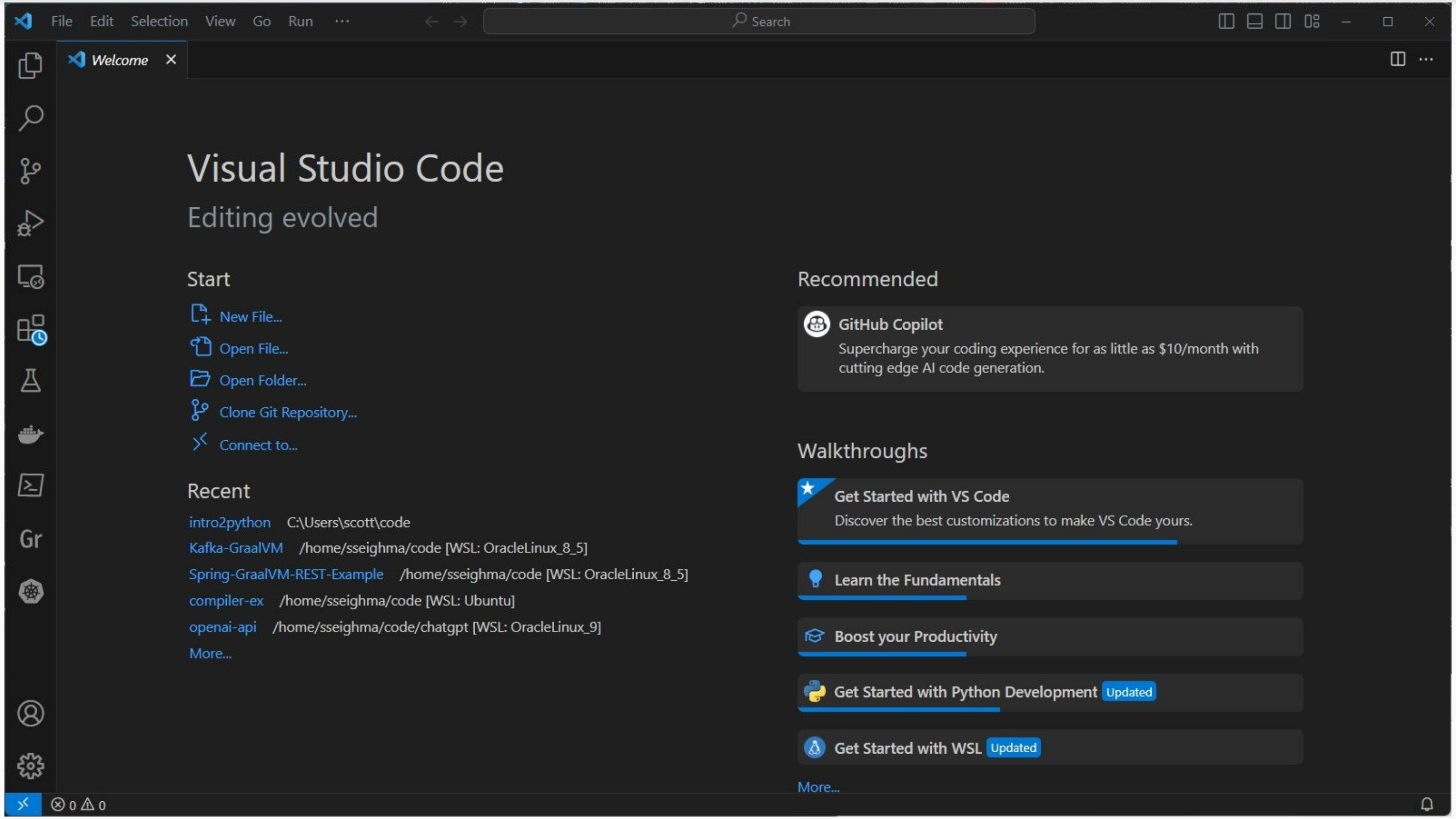
- To start IDLE:
  - Win Key -> All Apps -> Python 3.7



Getting Started

- We'll be using **Visual Studio Code**, a free coding editor that helps you start coding quickly
- Use it to code in any programming language, without switching editors
- Visual Studio Code has support for many languages, including Python, Java, C++, JavaScript, and more
- Visual Studio Code is built with extensibility in mind
- From the UI to the editing experience, almost every part of VS Code can be customized and enhanced through the Extension API

Getting Started



Visual Studio Code

Editing evolved

Start

New File...

Open File...

Open Folder...

Clone Git Repository...

Connect to...

Recent

intro2python	C:\Users\scott\code
Kafka-GraalVM	/home/sseighma/code [WSL: OracleLinux_8_5]
Spring-GraalVM-REST-Example	/home/sseighma/code [WSL: OracleLinux_8_5]
compiler-ex	/home/sseighma/code [WSL: Ubuntu]
openai-api	/home/sseighma/code/chatgpt [WSL: OracleLinux_9]

More...

Recommended

GitHub Copilot

Supercharge your coding experience for as little as $10/month with cutting edge AI code generation.

Walkthroughs

Get Started with VS Code

Discover the best customizations to make VS Code yours.

Learn the Fundamentals

Boost your Productivity

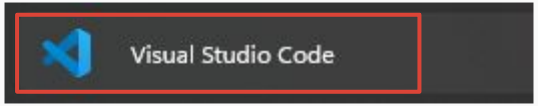
Get Started with Python Development Updated

Get Started with WSL Updated

More...

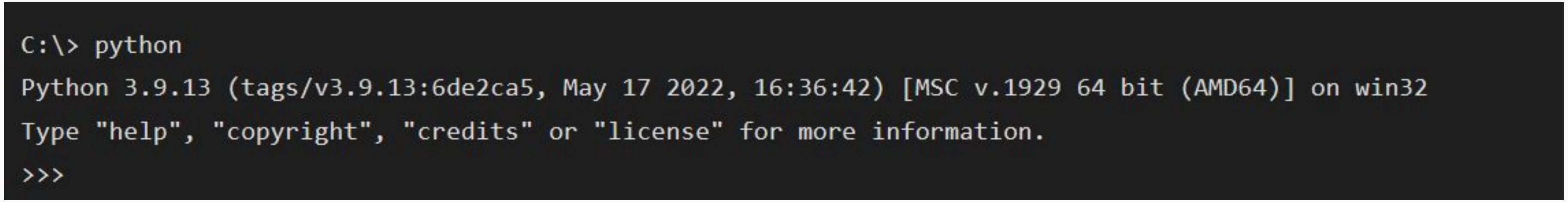
Getting Started

- To start Visual Studio Code:
    - Win Key -> All Apps



Getting Started

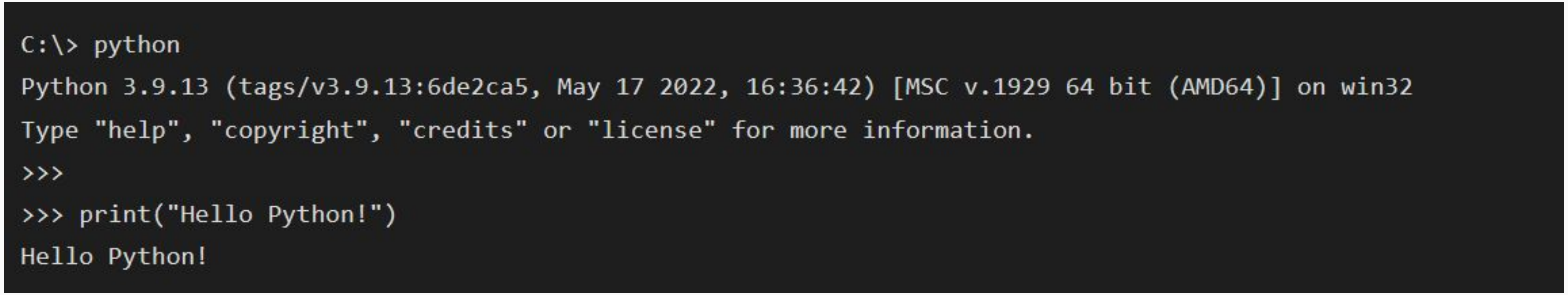
- At a basic level, you can choose to use the **command line interface** (CLI) to write and execute Python scripts in interactive mode
- It's straightforward and simple but can become a bit cumbersome when you begin to write more advanced applications



```
C:\> python
Python 3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022, 16:36:42) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Getting Started

- Open a Windows Terminal:
  - Press **Win + X** to open the menu
  - Scroll down to **Terminal** and click on the Terminal app (or enter "**i**" as a shortcut)
- Type: `python`
- Enter: `print(“Hello Python!”)`



```
C:\> python
Python 3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022, 16:36:42) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print("Hello Python!")
Hello Python!
```

Getting Started

- Open Notepad:
  - Press **Win + R** to open the Run menu
  - Type: `notepad`
  - Click **OK** or press the **Enter** key

- Type: `print("Hello World!")`
- Click **File->Save As** and enter `hello_world.py`

.py is the standard file extension for Python source code files. These files contain Python code that can be executed by a Python interpreter

- Back in Windows Terminal, enter the following command to run **hello_world.py**:

```
python helloworld.py
```

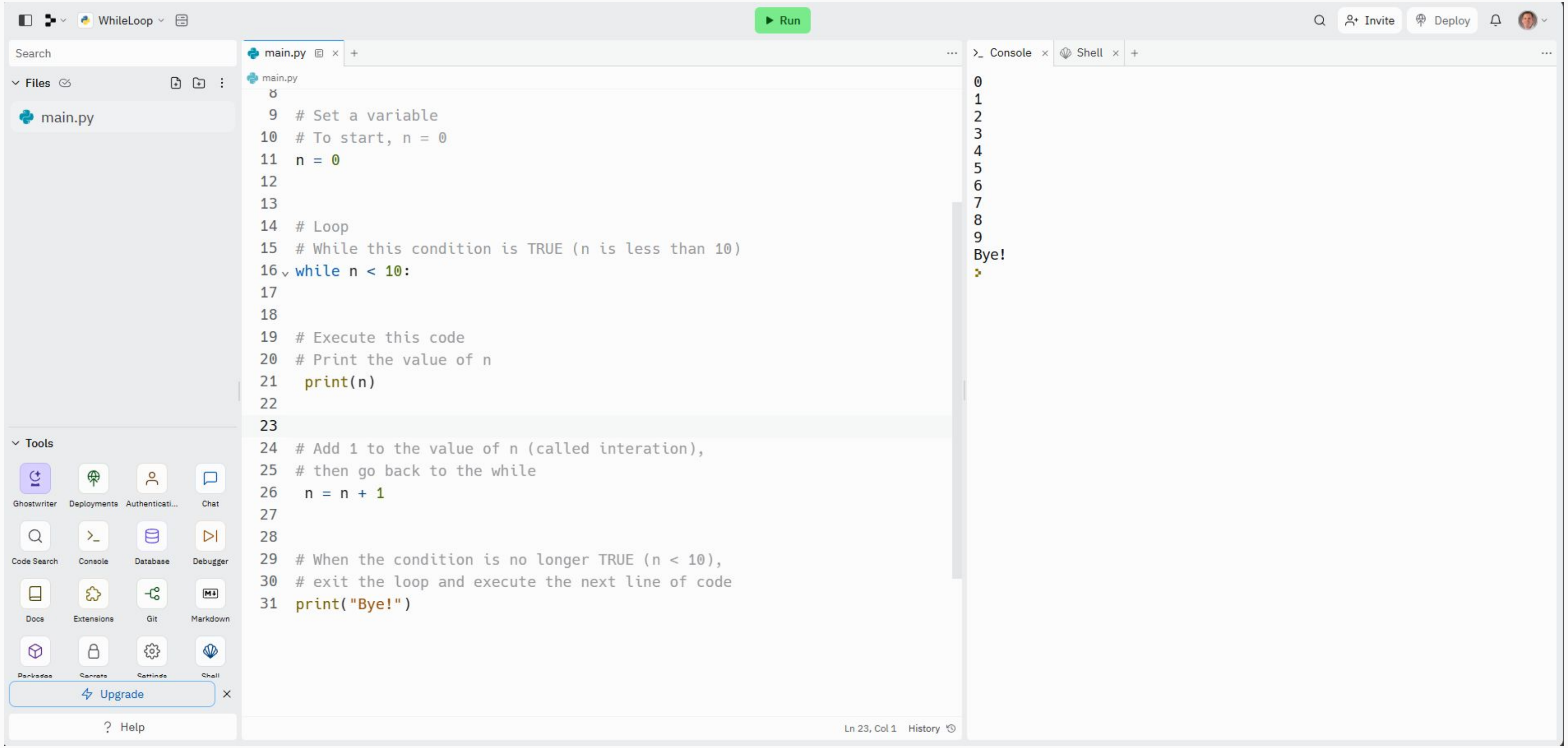
Getting Started

- Replit
  - A Repl (derived from REPL—"read-eval-print loop") is an interactive programming environment where you can write and execute code in real-time
  - From their site:

    “Replit provides a large range of tools and features necessary for software development. It serves as an IDE, a code collaboration platform, a cloud provider, a developer community, and so much more.”

  - The workspace is like an IDE (Integrated Development Environment), providing a comprehensive set of development tools and features for creating, debugging, and managing your software
  - Visit https://replit.com/ to try out the environment

Getting Started



```python
# Set a variable
# To start, n = 0
n = 0


# Loop
# While this condition is TRUE (n is less than 10)
while n < 10:


# Execute this code
# Print the value of n
 print(n)


# Add 1 to the value of n (called interation),
# then go back to the while
 n = n + 1


# When the condition is no longer TRUE (n < 10),
# exit the loop and execute the next line of code
print("Bye!")
```

```
0
1
2
3
4
5
6
7
8
9
Bye!
```

Core Concepts:

Variables

Strings

Numbers

Collections/Lists

User Input

Core Concepts: **Variables**

- Variables are often described as containers where you store values
- Variables can be used to store anything from simple data types (like integers and strings) to more complex data structures (like lists)
- You can think of them as labels you attach to values to keep track of them
- You can also say that a variable references a certain value
- Python uses = to assign values to variables
- Assigning a value to a variable itself declares and initializes the variable with that value (no need to assign a data type to it)

Core Concepts: **Variables**

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number
  - For instance, you can call a variable `message_1` but not `1_message`
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names
  - For example, `greeting_message` works but `greeting message` will cause errors
- Avoid using Python keywords and function names (see Keywords on upcoming slide) as variable names
  - For example, do not use the word `print` as a variable name; Python has reserved it for a particular programmatic purpose

Core Concepts: **Variables**

- Variable names should be short but descriptive
    - For example, `name` is better than `n`, `student_name` is better than `s_n`
- Be careful when using the lowercase letter `l` and the uppercase letter `O` because they could be confused with the numbers `1` and `0`

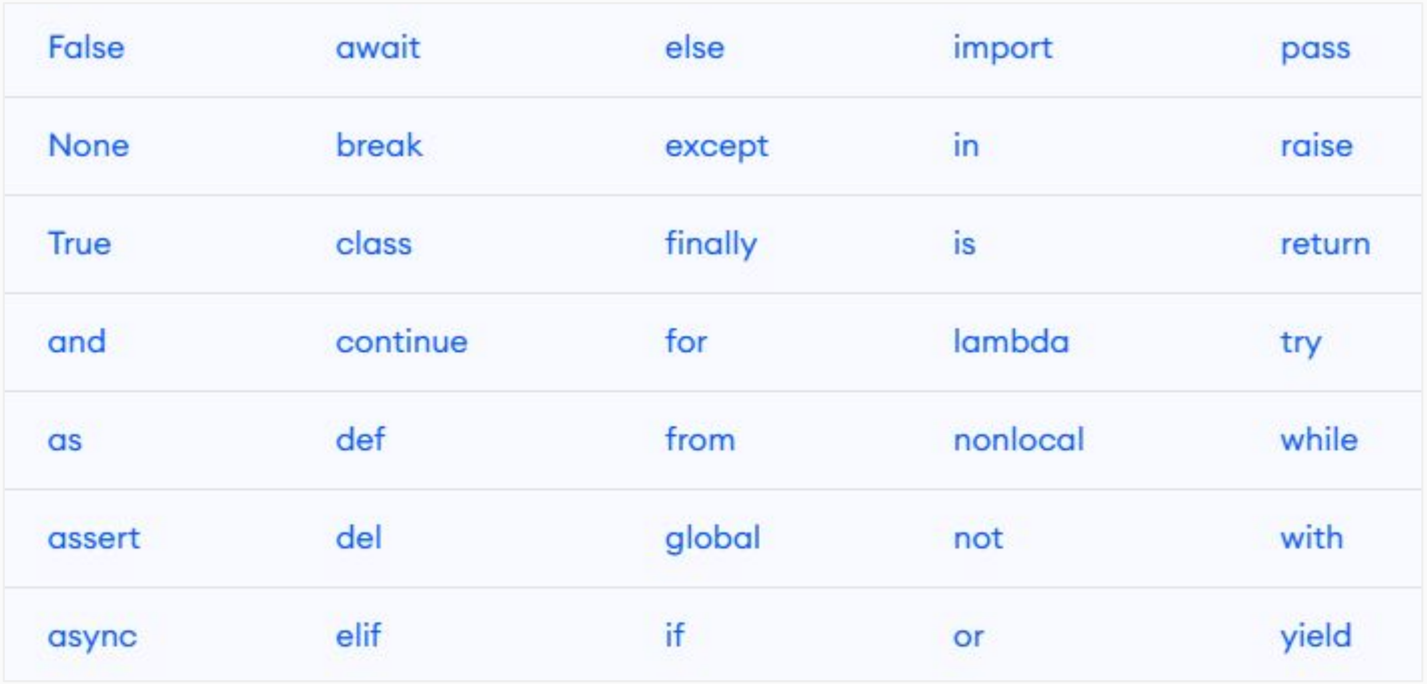
Core Concepts: **Variables**

- Rules for Naming Python Identifiers:
    - It cannot be a reserved Python keyword
    - It should not contain white space
    - It can be a combination of A-Z, a-z, 0-9, or underscore
    - It should start with an alphabet character or an underscore (_)
    - It should not contain any special character other than an underscore (_)

Core Concepts: **Variables**

- Valid identifiers/names:
  - `var1`
  - `_var1`
  - `_1_var`
  - `Var_1`
- Invalid Identifiers/names:
  - `!var1`
  - `1var`
  - `1_var`
  - `var#1`
  - `var 1`

Core Concepts: **Variables**



False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Core Concepts:

Variables

Strings

Numbers

Collections/Lists

User Input

Core Concepts: **Data Types**

- Because most programs define and gather some sort of data and then do something useful with it, it helps to classify different types of data
- **Data types** are the classification or categorization of data items (objects) and include numeric types (int, float, complex), textual types (string), collection types (list, tuple, dict, set), and boolean type (bool)
- The first data type we'll explore is the **string**
- Strings are quite simple at first glance, but you can use them in many different ways

Core Concepts: **Strings**

- A string is a series of characters
- Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
‘This is also a string.’
```

- This flexibility allows you to use quotes and apostrophes within your strings:

```
"The language 'Python' is named after Monty Python, not
the snake."
```

Core Concepts: **Strings**

- You can add whitespace to strings with tabs or newlines
- In programming, whitespace refers to any nonprinting characters, such as spaces, tabs, and end-of-line symbols
- You can use whitespace to organize your output so it's easier for users to read
- To add a tab to your text, use the character combination `\t`:

```
print("\tGo Browns!")
```

- Output:

```
	Go Browns
```

Core Concepts: **Strings**

- To add a newline in a string, use the character combination `\n`:

```
print("Teams:\nBrowns\nSteelers\nBengals\nRavens")
```

- Output:

```
Teams:
Browns
Steelers
Bengals
Ravens
```

Core Concepts: **Strings**

- You can also combine tabs and newlines in a single string
- The string `\n\t` tells Python to move to a new line, and start the next line with a tab

```
print("Teams:\n\tBrowns\n\tSteelers\n\tBengals\n\tRavens")
```

- Output:

```
Teams:
	Browns
	Steelers
	Bengals
	Ravens
```

Core Concepts: **Strings**

- One of the simplest tasks you can do with strings is change the case of the words in a string

```
name = "joe thomas"
print(name.title())
```

`title` is a Python method which are functions that are associated with objects. Methods are used to perform actions on objects

- Save this file as `name.py` and then run it. You should see this output:

```
C:\> python name.py
Joe Thomas
```

Core Concepts: **Strings**

- Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
name = "Joe Thomas"
print(name.upper())
print(name.lower())
```

- This will display the following:

```
JOE THOMAS
joe thomas
```

Core Concepts: **Strings**

- In some situations, you'll want to use a variable's value inside a string
- For example, you might want to use two variables to represent a first name and a last name, respectively, and then combine those values to display someone's full name

Core Concepts: **Strings**

```python
first_name = "joe"
last_name = "thomas"
full_name = f"{first_name} {last_name}"
print(full_name)
```

- To insert a variable's value into a string, place the letter `f` immediately before the opening quotation mark
- Put braces around the name or names of any variable you want to use inside the string
- Python will replace each variable with its value when the string is displayed

Core Concepts: **Strings**

- You can do a lot with f-strings. For example, you can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
first_name = "joe"
last_name = "thomas"
full_name = f"{first_name} {last_name}"
print(f"Hello, {full_name.title()}!")
```

Core Concepts:

Variables
Strings
Numbers
Collections/Lists
User Input

Core Concepts: **Numbers**

- Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on
- Python treats numbers in several different ways, depending on how they're being used
- Let's first look at how Python manages **integers**, because they're the simplest to work with

Core Concepts: **Numbers**

- You can add (**+**), subtract (**-**), multiply (*****), and divide (**/**) integers in Python

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

Core Concepts: **Numbers**

- In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

- Python supports the order of operations too, so you can use multiple operations in one expression
- You can also use parentheses to modify the order of operations so Python can

Core Concepts: **Numbers**

- Python supports the order of operations too, so you can use multiple operations in one expression
- You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify
- For example:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

Core Concepts: **Numbers**

- Python calls any number with a decimal point a **float**
- This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number
- Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately, no matter where the decimal point appears
- For the most part, you can use floats without worrying about how they behave

Core Concepts: **Numbers**

- Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

Core Concepts: **Numbers**

- However, be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

- This happens in all languages and is of little concern
- Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally

Core Concepts: **Numbers**

- When you divide any two numbers, even if they are integers that result in a whole number, you'll always get a float:

```
>>> 4/2
2.0
```

Core Concepts: **Numbers**

- If you mix an integer and a float in any other operation, you’ll get a float as well:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

- Python defaults to a float in any operation that uses a float, even if the output is a whole number

Core Concepts:

Variables
Strings
Numbers
Collections/Lists
User Input

Core Concepts: **Lists**

- A **list** is a collection of items in a particular order
- You can make a list that includes the letters of the alphabet, the digits from 0 to 9, or the names of all the people in your family
- You can put anything you want into a list, and the items in your list don't have to be related in any particular way
- Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names

Core Concepts: **Lists**

- In Python, square brackets (**[ ]**) indicate a list, and individual elements in the list are separated by commas. Here’s a simple example of a list that contains a few kinds of toppings:

```
toppings = ['pepperoni', 'sausage', 'peppers', 'mushrooms']
print(toppings)
```

- If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
['pepperoni', 'sausage', 'peppers', 'mushrooms']
```

- Because this isn’t the output you want your users to see, let’s learn how to access the individual items in a list

Core Concepts: **Lists**

- Lists are ordered collections, so you can access any element in a list by telling Python the position, or index, of the item desired
- To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets.
- For example, let’s pull out the first topping in the list toppings:

```
toppings = ['pepperoni', 'sausage', 'peppers', 'mushrooms']
print(toppings[0])
```

- When we ask for a single item from a list, Python returns just that element without square brackets:

```
pepperoni
```

Core Concepts: **Lists**

- This is the result you want your users to see: clean, neatly formatted output
- You can also use the string methods on any element in this list. For example, you can format the element pepperoni to look more presentable by using the `title()` method:

```
toppings = ['pepperoni', 'sausage', 'peppers', 'mushrooms']
print(toppings[0].title())
```

- This example produces the same output as the preceding example, except pepperoni is capitalized

Core Concepts: **Lists**

- Python considers the first item in a list to be at position 0, not position 1
- This is true of most programming languages, and the reason has to do with how the list operations are implemented at a lower level
- If you’re receiving unexpected results, ask yourself if you’re making a simple but common off-by-one error
- The second item in a list has an index of 1
- Using this counting system, you can get any element you want from a list by subtracting one from its position in the list
- For instance, to access the fourth item in a list, request the item at index 3

Core Concepts: **Lists**

- The following asks for the toppings at index 1 and index 3:

```
toppings = ['pepperoni', 'sausage', 'peppers', 'mushrooms']
print(toppings[1])
print(toppings[3])
```

- This code returns the second and fourth toppings in the list:

```
sausage
mushrooms
```

Core Concepts: **Lists**

- Python has a special syntax for accessing the last element in a list. If you ask for the item at index `-1`, Python always returns the last item in the list:

  ```
  toppings = ['pepperoni', 'sausage', 'peppers', 'mushrooms']
  print(toppings[-1])
  ```

- This code returns the value **`mushrooms`**
- This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is
- This convention extends to other negative index values as well. The index `-2` returns the second item from the end of the list, the index `-3` returns the third item from the end, and so forth

Core Concepts: **Lists**

- You can use individual values from a list just as you would any other variable. For example, you can use f-strings to create a message based on a value from a list
- Let’s try pulling the first bicycle from the list and composing a message using that value:

```
toppings = ['pepperoni', 'sausage', 'peppers', 'mushrooms']
message = f"My favorite pizza topping is {toppings[0].title()}."
print(message)
```

Core Concepts: **Lists**

- We build a sentence using the value at `toppings[0]` and assign it to the variable message
- The output is a simple sentence about the first topping in the list:

```
My favorite pizza topping is Pepperoni.
```

Core Concepts: **Lists**

- Most lists you create will be dynamic, meaning you'll build a list and then add and remove elements from it as your program runs its course
- For example, you might create a game in which a player has to shoot aliens out of the sky
- You could store the initial set of aliens in a list and then remove an alien from the list each time one is shot down
- Each time a new alien appears on the screen, you add it to the list
- Your list of aliens will increase and decrease in length throughout the course of the game

Core Concepts: **Lists**

- The syntax for modifying an element is similar to the syntax for accessing an element in a list. To change an element, use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have
- For example, say we have a list of **`toppings`** and the first item in the list is **`pepperoni`**. We can change the value of this first item after the list has been created:

```
toppings = ['pepperoni', 'sausage', 'peppers']
print(toppings)
toppings[0] = `mushrooms`
print(toppings)
```

Core Concepts: **Lists**

- We define the list **`toppings`**, with **`pepperoni`** as the first element
- Then we change the value of the first item to **`mushrooms`**
- The output shows that the first item has been changed, while the rest of the list stays the same:

```
['pepperoni', 'sausage', 'peppers']
['mushrooms', 'sausage', 'peppers']
```

- You can change the value of any item in a list, not just the first item

Core Concepts: **Lists**

- You might want to add a new element to a list for many reasons
- For example, you might want to make new aliens appear in a game, add new data to a visualization, or add new registered users to a website you’ve built.
- Python provides several ways to add new data to existing lists

Core Concepts: **Lists**

Appending elements to the end of a list

- The simplest way to add a new element to a list is to append the item to the list. When you append an item to a list, the new element is added to the end of the list
- Using the same list we had in the previous example, we'll add the new element mushrooms to the end of the list:

```
toppings = ['pepperoni', 'sausage', 'peppers']
print(toppings)
toppings.append('mushrooms')
print(toppings)
```

Core Concepts: **Lists**

- Here the **`append()`** method adds mushrooms to the end of the list, without affecting any of the other elements in the list:

```
['pepperoni', 'sausage', 'peppers']
['pepperoni', 'sausage', 'peppers', 'mushrooms']
```

- The **`append()`** method makes it easy to build lists dynamically. For example, you can start with an empty list and then add items to the list using a series of **`append()`** calls

Core Concepts: **Lists**

- Using an empty list, let’s add the elements pepperoni, sausage, and peppers to the list:

```
toppings = []
toppings.append('pepperoni')
toppings.append('sausage')
toppings.append('peppers')
print(toppings)
```

- The resulting list looks exactly the same as the lists in the previous examples:

```
['pepperoni', 'sausage', 'peppers']
```

- Building lists this way is very common, because you often won’t know the data your users want to store in a program until after the program is running

Core Concepts: **Lists**

- Building lists this way is very common, because you often won’t know the data your users want to store in a program until after the program is running
- To put your users in control, start by defining an empty list that will hold the users’ values
- Then append each new value provided to the list you just created

Core Concepts: **Lists**

- You can add a new element at any position in your list by using the insert() method. You do this by specifying the index of the new element and the value of the new item:

```
toppings = ['pepperoni', 'sausage', 'peppers']
toppings.insert(0, 'mushrooms')
print(toppings)
```

- In this example, we insert the value mushrooms at the beginning of the list. The `insert()` method opens a space at position 0 and stores the value mushrooms at that location:

```
['mushrooms', 'pepperoni', 'sausage', 'peppers']
```

Core Concepts: **Lists**

Removing Elements from a List

- Often, you'll want to remove an item or a set of items from a list
- For example, when a player shoots down an alien from the sky, you'll most likely want to remove it from the list of active aliens
- Or when a user decides to cancel their account on a web application you created, you'll want to remove that user from the list of active users
- You can remove an item according to its position in the list or according to its value

Core Concepts: **Lists**

Removing an Item Using the **`del`** Statement

- If you know the position of the item you want to remove from a list, you can use the **`del`** statement:

```
toppings = ['pepperoni', 'sausage', 'peppers']
print(toppings)

del toppings[0]
print(toppings)
```

Core Concepts: **Lists**

- Here we use the `del` statement to remove the first item, pepperoni, from the list of toppings:

```
['pepperoni', 'sausage', 'peppers']
['sausage', 'peppers']
```

Core Concepts: **Lists**

- You can remove an item from any position in a list using the `del` statement if you know its index. For example, here's how to remove the second item, sausage, from the list:

```
toppings = ['pepperoni', 'sausage', 'peppers']
print(toppings)

del toppings[1]
print(toppings)
```

Core Concepts: **Lists**

- The second topping is deleted from the list:

```
['pepperoni', 'sausage', 'peppers']
['pepperoni', 'peppers']
```

- In both examples, you can no longer access the value that was removed from the list after the **`del`** statement is used

Core Concepts:

- Variables
- Strings
- Numbers
- Collections/Lists
- **User Input**

Core Concepts: **User Input**

- Most programs are written to solve an end user's problem. To do so, you usually need to get some information from the user
- For example, say someone wants to find out whether they're old enough to vote. If you write a program to answer this question, you need to know the user's age before you can provide an answer
- The program will need to ask the user to enter, or input, their age; once the program has this input, it can compare it to the voting age to determine if the user is old enough and then report the result.
- To do this, you'll use the **`input()`** function

Core Concepts: **User Input**

- You’ll also learn how to keep programs running as long as users want them to, so they can enter as much information as they need to; then, your program can work with that information
- You’ll use Python’s **`while`** loop to keep programs running as long as certain conditions remain true
- With the ability to work with user input and the ability to control how long your programs run, you’ll be able to write fully interactive programs

Core Concepts: **User Input**

- The **`input()`** function pauses your program and waits for the user to enter some text
- Once Python receives the user’s input, it assigns that input to a variable to make it convenient for you to work with
- For example, the following program asks the user to enter some text, then displays that message back to the user:

```
message = input("Tell me something, and I will repeat it: ")
print(message)
```

Core Concepts: **User Input**

- The `input()` function takes one argument: the prompt that we want to display to the user, so they know what kind of information to enter
- In this example, when Python runs the first line, the user sees the prompt **Tell me something, and I will repeat it:**
- The program waits while the user enters their response and continues after the user presses **ENTER**
- The response is assigned to the variable message, then print(message) displays the input back to the user:

```
Tell me something, and I will repeat it: Hello everyone!
Hello everyone!
```

Core Concepts: **User Input**

- Each time you use the `input()` function, you should include a clear, easy-to-follow prompt that tells the user exactly what kind of information you're looking for
- For example:

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

- Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text
- For example:
- Please enter your name: Eric

Core Concepts: **User Input**

- For example:

```
Please enter your name: Eric
Hello, Eric!
```

- Sometimes you’ll want to write a prompt that’s longer than one line. For example, you might want to tell the user why you’re asking for certain input
- You can assign your prompt to a variable and pass that variable to the input() function. This allows you to build your prompt over several lines, then write a clean `input()` statement.
- prompt = "\nIf you share your name, we can personalize the messages you see."
- prompt += "\nWhat is your first name?

Core Concepts: **User Input**

```
prompt = "\nIf you share your name, we can personalize the messages you see."
prompt += "\nWhat is your first name?: "
name = input(prompt)
print(f"\nHello, {name}!\n")
```

- This example shows one way to build a multiline string. The first line assigns the first part of the message to the variable prompt
- In the second line, the operator **+=** takes the string that was assigned to prompt and adds the new string onto the end

Core Concepts: **User Input**

- The prompt now spans two lines, again with space after the question mark for clarity:

```
If you share your name, we can personalize the messages you
see.
What is your first name?: Scott
Hello, Scott!
```

Core Concepts: **User Input**

- When you use the **`input()`** function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

- The user enters the number 21, but when we ask Python for the value of age, it returns '21', the string representation of the numerical value entered
- We know Python interpreted the input as a string because the number is now enclosed in quotes. If all you want to do is print the input, this works well. But

Core Concepts: **User Input**

- We know Python interpreted the input as a string because the number is now enclosed in quotes. If all you want to do is print the input, this works well. But if you try to use the input as a number, you'll get an error:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>=' not supported between instances of 'str' and
'int'
```

- When you try to use the input to do a numerical comparison, Python produces an error because it can't compare a string to an integer: the string '21' that's

Core Concepts: **User Input**

- When you try to use the input to do a numerical comparison, Python produces an error because it can’t compare a string to an integer: the string '21' that’s assigned to age can’t be compared to the numerical value 18
- We can resolve this issue by using the `int()` function, which converts the input string to a numerical value. This allows the comparison to run successfully:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age = int(age)
>>> age >= 18
True
```

Core Concepts: **User Input**

- In this example, when we enter 21 at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by **`int()`**
- Now Python can run the conditional test: it compares age (which now represents the numerical value 21) and 18 to see if age is greater than or equal to 18. This test evaluates to True
- How do you use the int() function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

```
height = input("How tall are you, in inches? ")

height = int(height)
```

Core Concepts: **User Input**

- How do you use the int() function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

```
height = input("How tall are you, in inches? ")
height = int(height)
if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little
older.")
```

- The program can compare height to 48 because height = int(height) converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 48, we tell the user that

Core Concepts: **User Input**

- The program can compare height to 48 because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 48, we tell the user that they're tall enough:

```
How tall are you, in inches? 71
You're tall enough to ride!
```

- When you use numerical input to do calculations and comparisons, be sure to convert the input value to a numerical representation first

Review

Review

- Python is a very popular programming language that’s (fairly) easy to learn
- Variables, Data Types (Strings, Numbers, Lists)
- User Input
- Next:
    - Loops
    - Conditionals
    - Formatting
    - Adding Comments

Contact me: scottseighman@gmail.com