PYTHON FUNDAMENTALS

UNB-GGE PYTHON WORKSHOP SERIES 1

William Liu
Department of Geodesy and Geomatics Engineering
University of New Brunswick
July 2023

Agenda

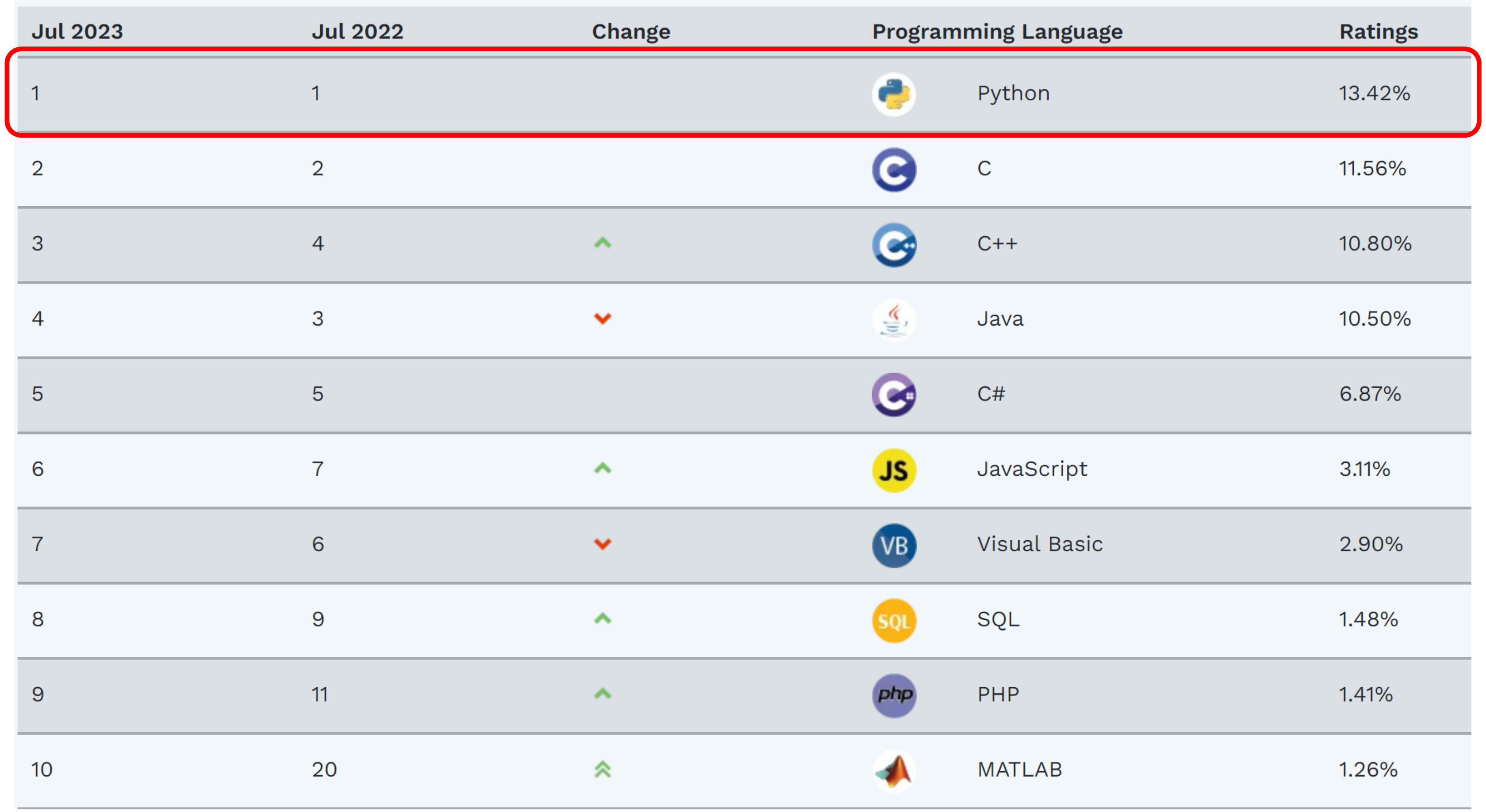
1. Why Python?
2. Python Overview
3. Setting up Your Development Environment
4. Python Basics
    - 4.0 Comments
    - 4.1 Literals
    - 4.2 Variables
    - 4.3 Operators
    - 4.4 Control Structures
    - 4.5 Data Types
        - NoneType
        - Numeric Types: `int`, `float`, `bool`, `complex`
        - Sequential Types: `str`, `list`, `tuple`,
        - Mapping and Set Types: `set`, `dict`
    - 4.6 Functions
    - 4.7 Classes, Objects, and Methods
    - 4.8 Packages and Modules
5. Creating Your First Python Program
6. Assignment

1. Why Python?

Understand why we choose Python

1. Why Python?

TIOBE Programming Community index: an indicator of the popularity of programming languages.
The index can be used to check whether your programming skills are still up to date or to decide strategically what programming language should be adopted when starting to build a new software system.



Jul 2023	Jul 2022	Change	Programming Language	Ratings
1	1		Python	13.42%
2	2		C	11.56%
3	4	▲	C++	10.80%
4	3	▼	Java	10.50%
5	5		C#	6.87%
6	7	▲	JavaScript	3.11%
7	6	▼	Visual Basic	2.90%
8	9	▲	SQL	1.48%
9	11	▲	PHP	1.41%
10	20	▲▲	MATLAB	1.26%

Figure source: *https://www.tiobe.com/tiobe-index/*

1. Why Python?

A recent survey (link) by the “Stack Overflow” website reveals Python’s popularity among professional developers.

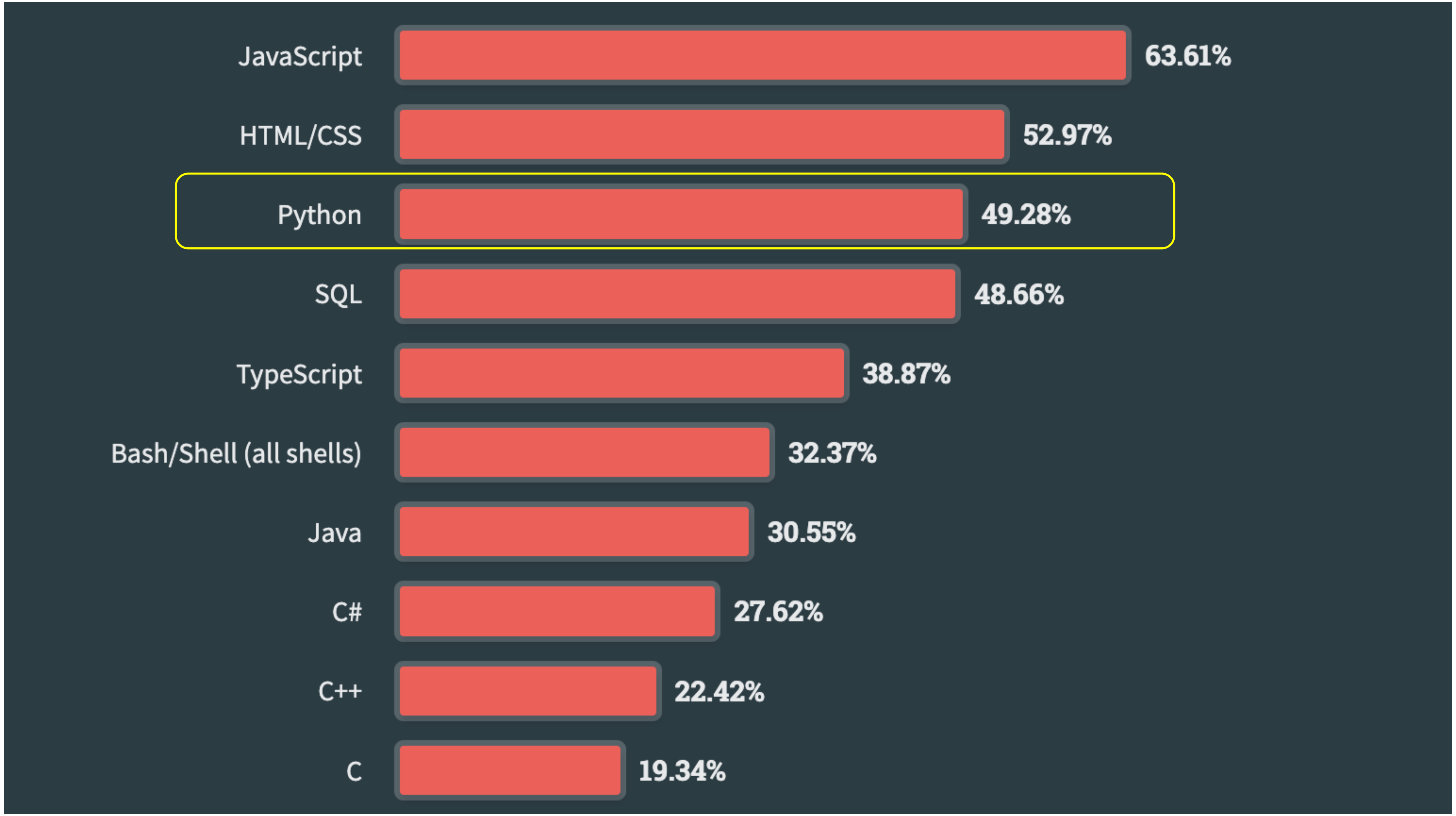
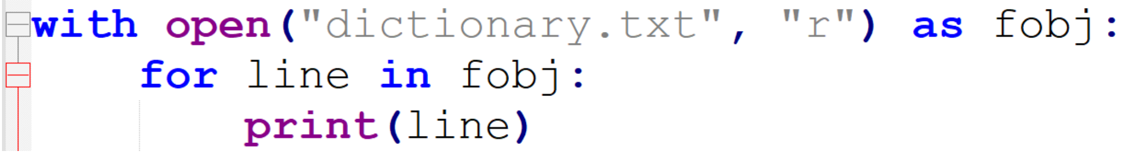


Figure source: *https://stackoverflow.blog/2023/06/13/developer-survey-results-are-in/*

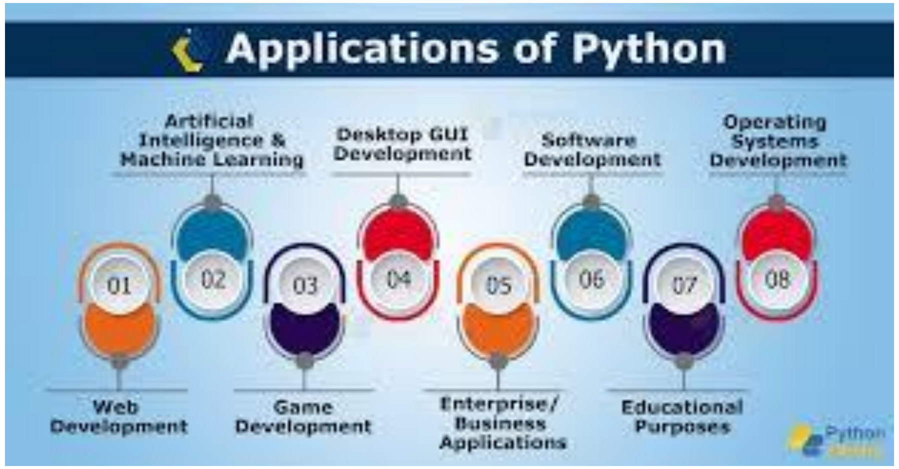
1. Why Python?

Why is Python so popular?

- **Readability:** Python’s syntax is easy to read and understand, making it a brilliant choice for all levels of programmers.
- **Simplicity:** Python allows developers to express ideas using fewer lines of code than other languages.
- **Versatility:** Python is multi-functional, with applications for web development, data analysis, AI, scientific computing, automation, etc.
- **Large and Active Community:** The Python programming language benefits from a diverse and active community of developers who contribute a wealth of resources, including libraries, frameworks, and support. This fosters a culture of collaboration, knowledge sharing, and ongoing improvement of the language.



```
with open("dictionary.txt", "r") as fobj:
    for line in fobj:
        print(line)
```



1. Why Python?

Why is Python so popular?

- **Cross-platform Compatibility:** Python is available on various platforms, such as Windows, macOS, Linux, and has consistent behavior across different operating systems.
- **Extensive Library Ecosystem:** Python offers a vast collection of libraries and frameworks, such as NumPy, Pandas, Django, TensorFlow, and Flask, which simplify development tasks and accelerate the creation of complex applications.
- **Integration Capabilities:** Python can easily integrate with other languages and systems, allowing developers to leverage existing code and infrastructure seamlessly.
- **Career Opportunities:** Python's popularity has led to high demand for Python developers in the job market, making it a valuable skill for career advancement.

1. Why Python?

Python in the Field of Geomatics

An article (link) in the Gogeomatics website ranked Python as the No 1 programming language in the GIS world:
"Python can be compared to the Swiss army knife for GIS."

Python is the leading programming language in the GIS world and here are some reasons why:

- **Wide Range of Geospatial Libraries:** Python has many powerful libraries for working with geospatial data, including GDAL, Fiona, GeoPandas, Shapely, and PyProj. These tools allow for reading and writing spatial file formats, performing spatial analysis, handling projections, visualizing geospatial data, and processing remote sensing imagery.
- **Data Analysis and Visualization:** Python libraries like NumPy, Pandas, and Matplotlib are great for geospatial data analysis, manipulation, and visualization.
- **Integration with GIS Software:** Python works well with GIS software like QGIS and ArcGIS, allowing users to create custom tools, automate workflows, and access APIs.
- **Open Source and Community Support:** Python is open-source and has a strong community of geospatial professionals who contribute to its development and provide support through various online resources.
- **Cross-Domain Integration:** Python is a popular choice for interdisciplinary projects involving geospatial data as it easily integrates with remote sensing, data science, machine learning, and web development.

2. Python Overview

Understand basic Python concepts

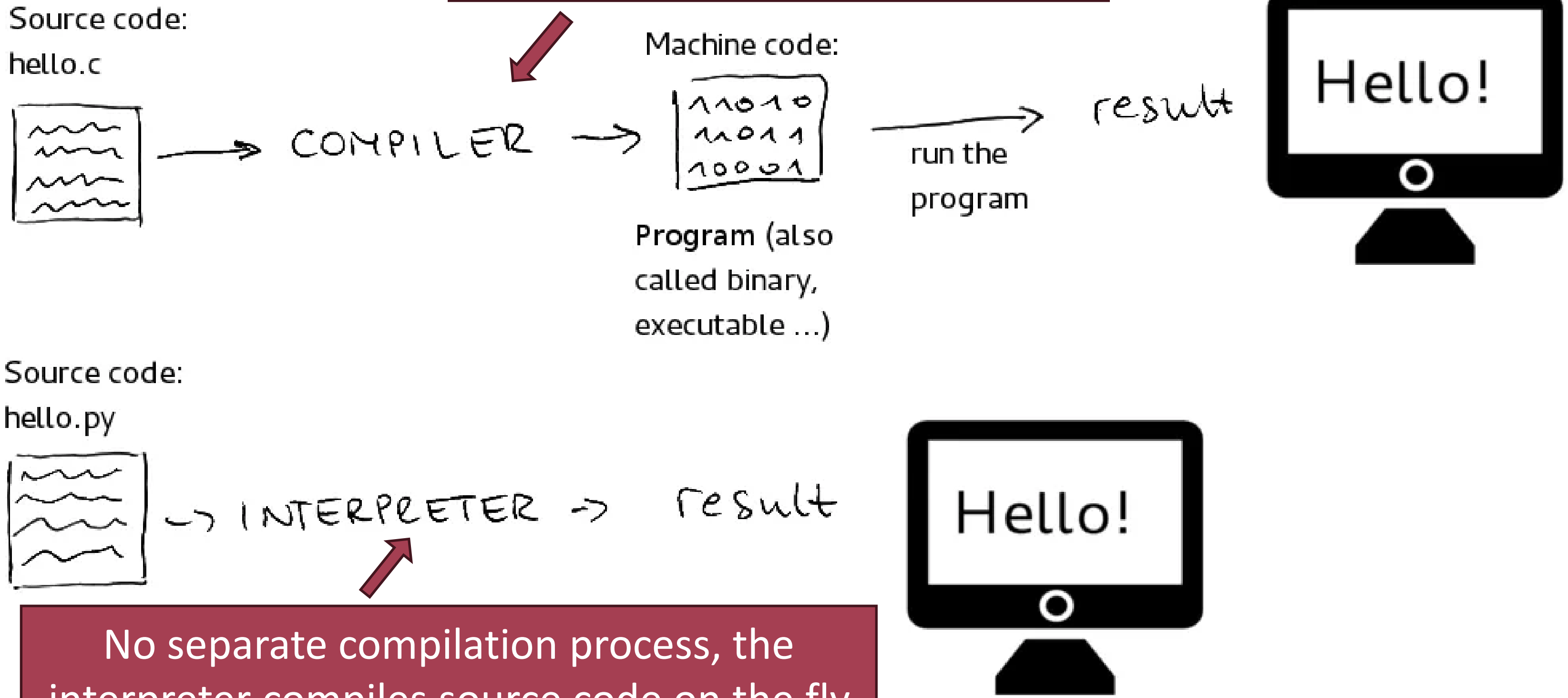
2. Python Overview

Python is an interpreted programming language that differs from the traditional compiled programming language such as C/C++, C#, and Java.

From a user's point of view:

Compiled language

Interpreted language



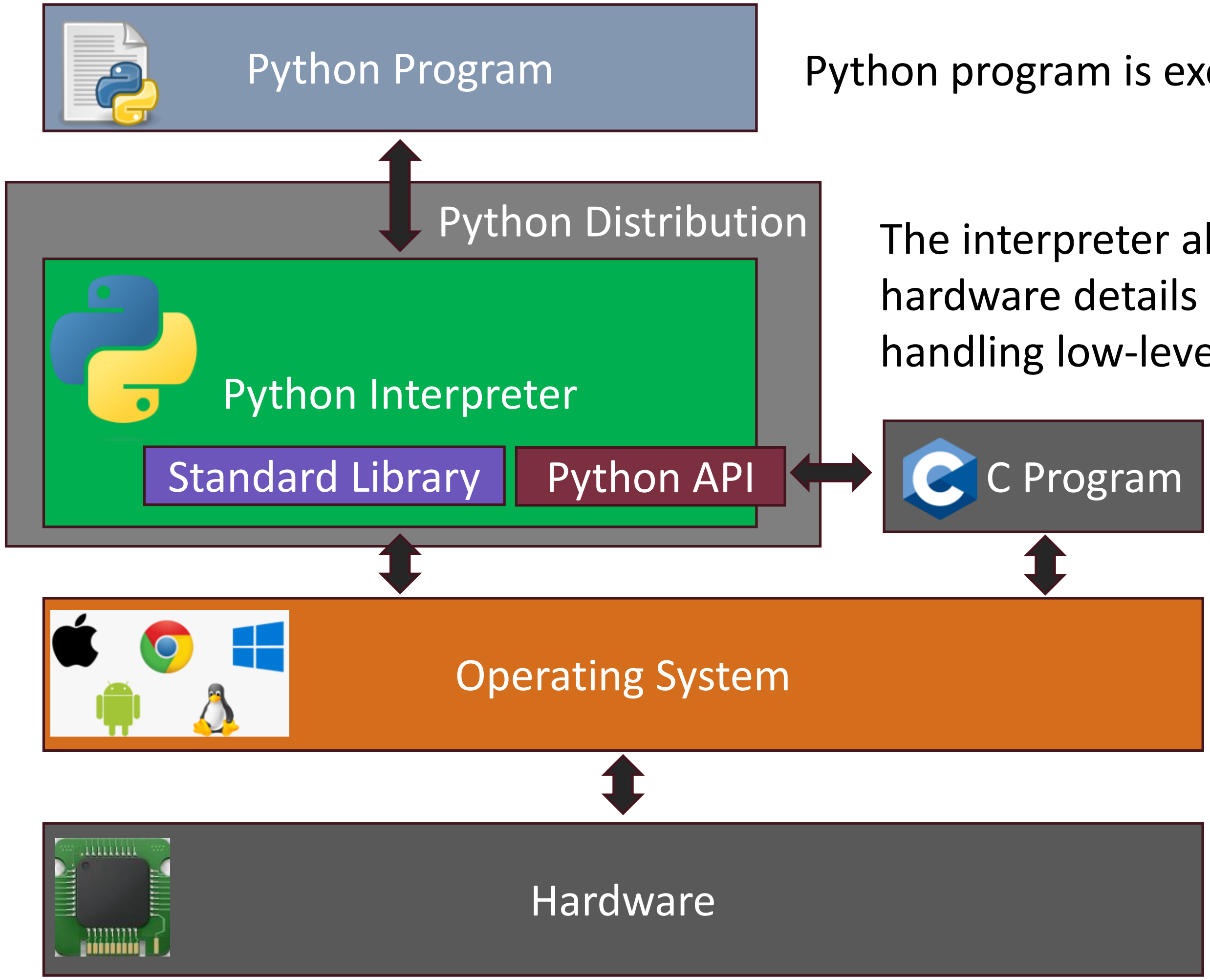
Separate compilation process

No separate compilation process, the interpreter compiles source code on the fly

Figure source: *https://medium.com/from-the-scratch/stop-it-there-are-no-compiled-and-interpreted-languages-512f84756664*

2. Python Overview

Python Concepts



Python program is executed by the interpreter

The interpreter abstracts away from the operating system and the hardware details by providing a uniform programming interface, handling low-level operations, and ensuring platform independence

The interpreter can be extended with C programs by using the Python API for time-critical applications

The standard library provides powerful and easy-to-use functionalities to the Python program

2. Python Overview

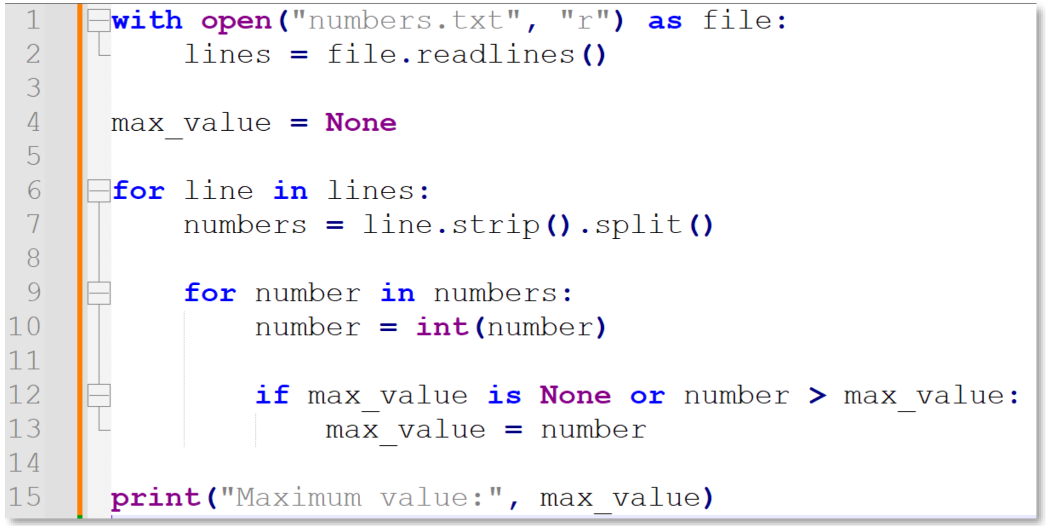
Python Program

Each Python program may contain one or many Python script files. Let’s examine a simple example that only consists of one Python file:

“Suppose we have a text file named numbers.txt containing multiple lines of integer numbers. We separate the numbers from others with a space in the file. Write a Python script file to find the maximum number from the file and then output it to the screen.”

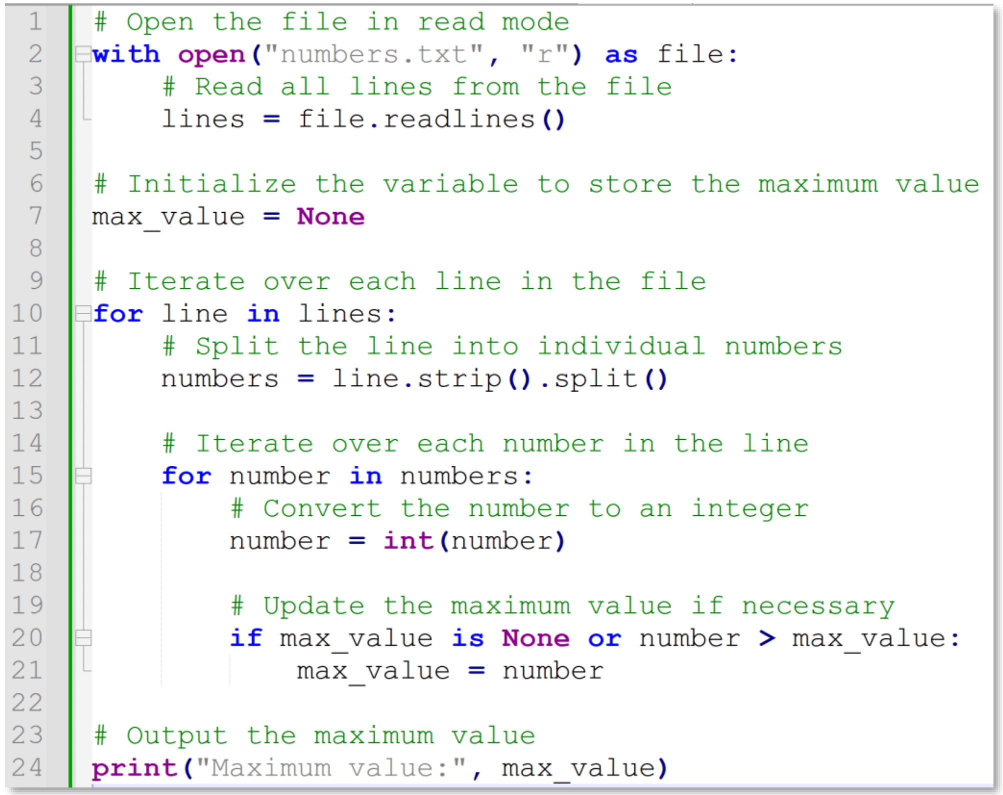
Read the left script first. If you really need some explanations, read the right one with comments

(Note: anything following the # symbol on the same line will be considered a comment and will not be executed by the Python interpreter).



```
with open("numbers.txt", "r") as file:
    lines = file.readlines()

max_value = None

for line in lines:
    numbers = line.strip().split()

    for number in numbers:
        number = int(number)

        if max_value is None or number > max_value:
            max_value = number

print("Maximum value:", max_value)
```

Without comments

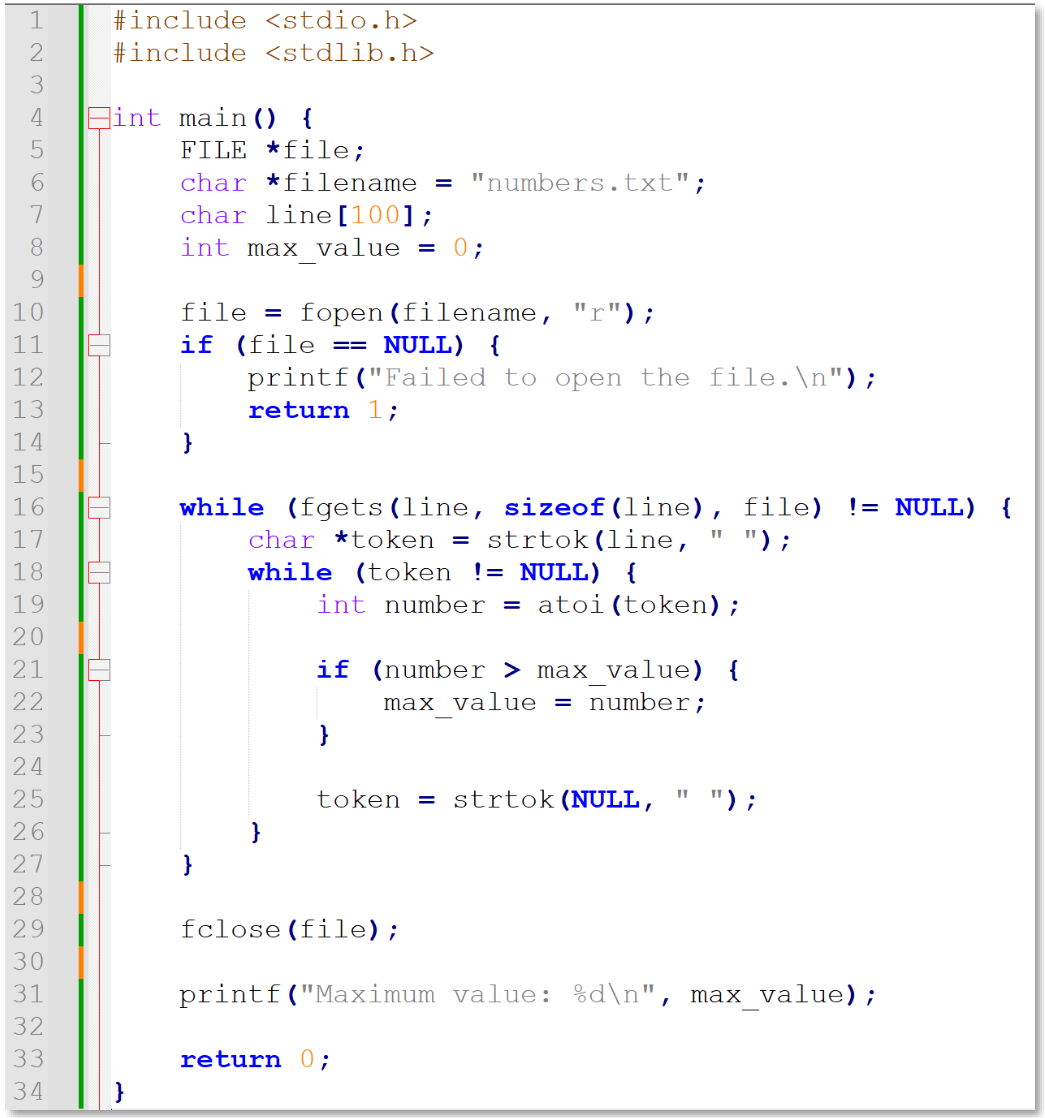


```
# Open the file in read mode
with open("numbers.txt", "r") as file:
    # Read all lines from the file
    lines = file.readlines()

# Initialize the variable to store the maximum value
max_value = None

# Iterate over each line in the file
for line in lines:
    # Split the line into individual numbers
    numbers = line.strip().split()

    # Iterate over each number in the line
    for number in numbers:
        # Convert the number to an integer
        number = int(number)

        # Update the maximum value if necessary
        if max_value is None or number > max_value:
            max_value = number

# Output the maximum value
print("Maximum value:", max_value)
```

With comments

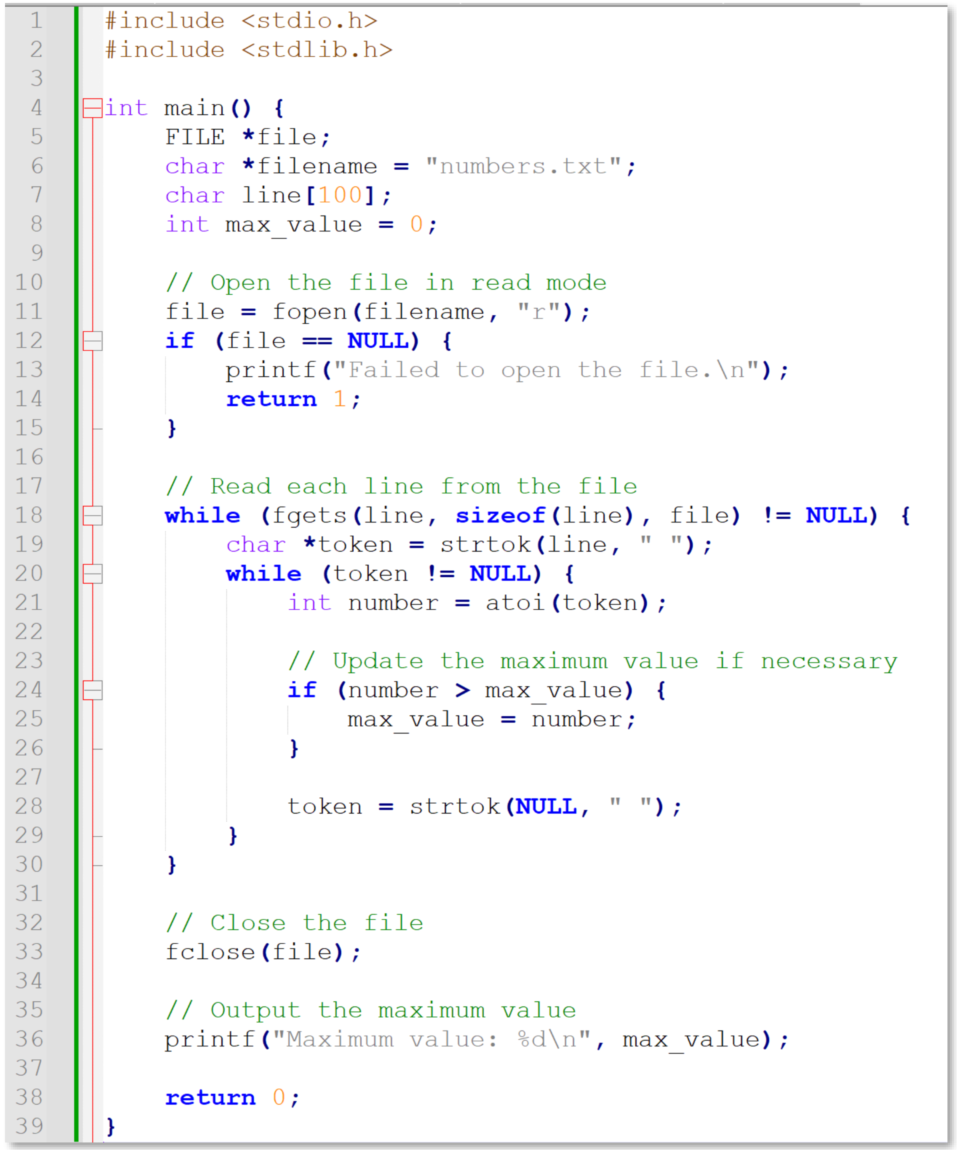
2. Python Overview

Compare the corresponding C program implementing the same functionality



```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file;
    char *filename = "numbers.txt";
    char line[100];
    int max_value = 0;

    file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 1;
    }

    while (fgets(line, sizeof(line), file) != NULL) {
        char *token = strtok(line, " ");
        while (token != NULL) {
            int number = atoi(token);

            if (number > max_value) {
                max_value = number;
            }

            token = strtok(NULL, " ");
        }
    }

    fclose(file);

    printf("Maximum value: %d\n", max_value);

    return 0;
}
```

Without comments



```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file;
    char *filename = "numbers.txt";
    char line[100];
    int max_value = 0;

    // Open the file in read mode
    file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 1;
    }

    // Read each line from the file
    while (fgets(line, sizeof(line), file) != NULL) {
        char *token = strtok(line, " ");
        while (token != NULL) {
            int number = atoi(token);

            // Update the maximum value if necessary
            if (number > max_value) {
                max_value = number;
            }

            token = strtok(NULL, " ");
        }
    }

    // Close the file
    fclose(file);

    // Output the maximum value
    printf("Maximum value: %d\n", max_value);

    return 0;
}
```

With comments

2. Python Overview

Python Program

Now, think about the **Readability** and **Simplicity** of the Python language**.** Are you convinced of what we claimed before?

2. Python Overview

Python Interpreter

Popular Python interpreter implementations

- **CPython**: The default and widely used Python interpreter, known for its performance, stability, and extensive library support.
- **PyPy**: A faster and more memory-efficient alternative to CPython, featuring a Just-in-Time (JIT) compiler for performance improvements.
- **Jython**: Python implementation running on the Java Virtual Machine (JVM), allowing seamless integration with Java code and libraries.
- **IronPython**: Python implementation targeting the .NET Framework, enabling Python and other .NET languages to work together.
- **MicroPython**: Lightweight Python implementation for microcontrollers and embedded systems with a small footprint.

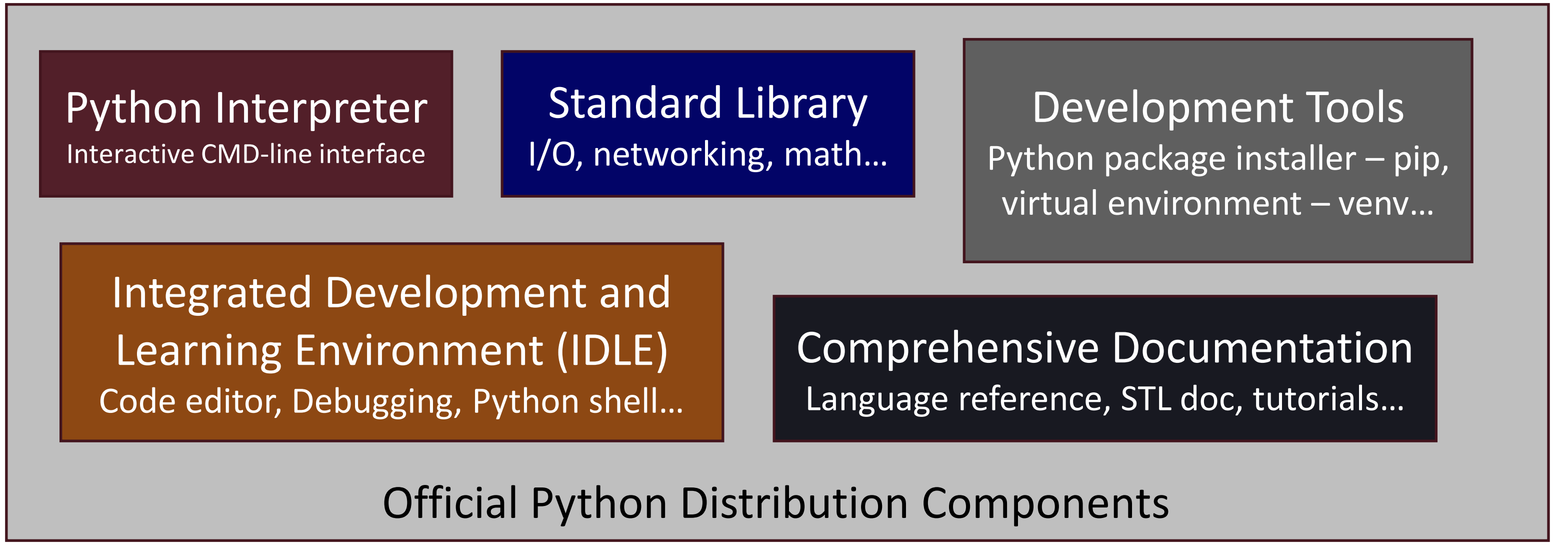
This workshop will focus on the default and reference implementation of the Python interpreter – CPython.

2. Python Overview

Python Distribution

A pre-packaged bundle of the Python programming language, along with additional tools, libraries, and resources that make it easier to develop, run, and distribute Python applications.

The official Python installation file provided by *http://www.python.org* commonly referred to as "CPython," can be considered a minimal Python distribution. It includes the core Python interpreter, the standard library, and essential tools for running Python code. It serves as the base from which other distributions are built.



Python Interpreter
Interactive CMD-line interface

Standard Library
I/O, networking, math…

Development Tools
Python package installer – pip, virtual environment – venv…

Integrated Development and Learning Environment (IDLE)
Code editor, Debugging, Python shell…

Comprehensive Documentation
Language reference, STL doc, tutorials…

Official Python Distribution Components

2. Python Overview

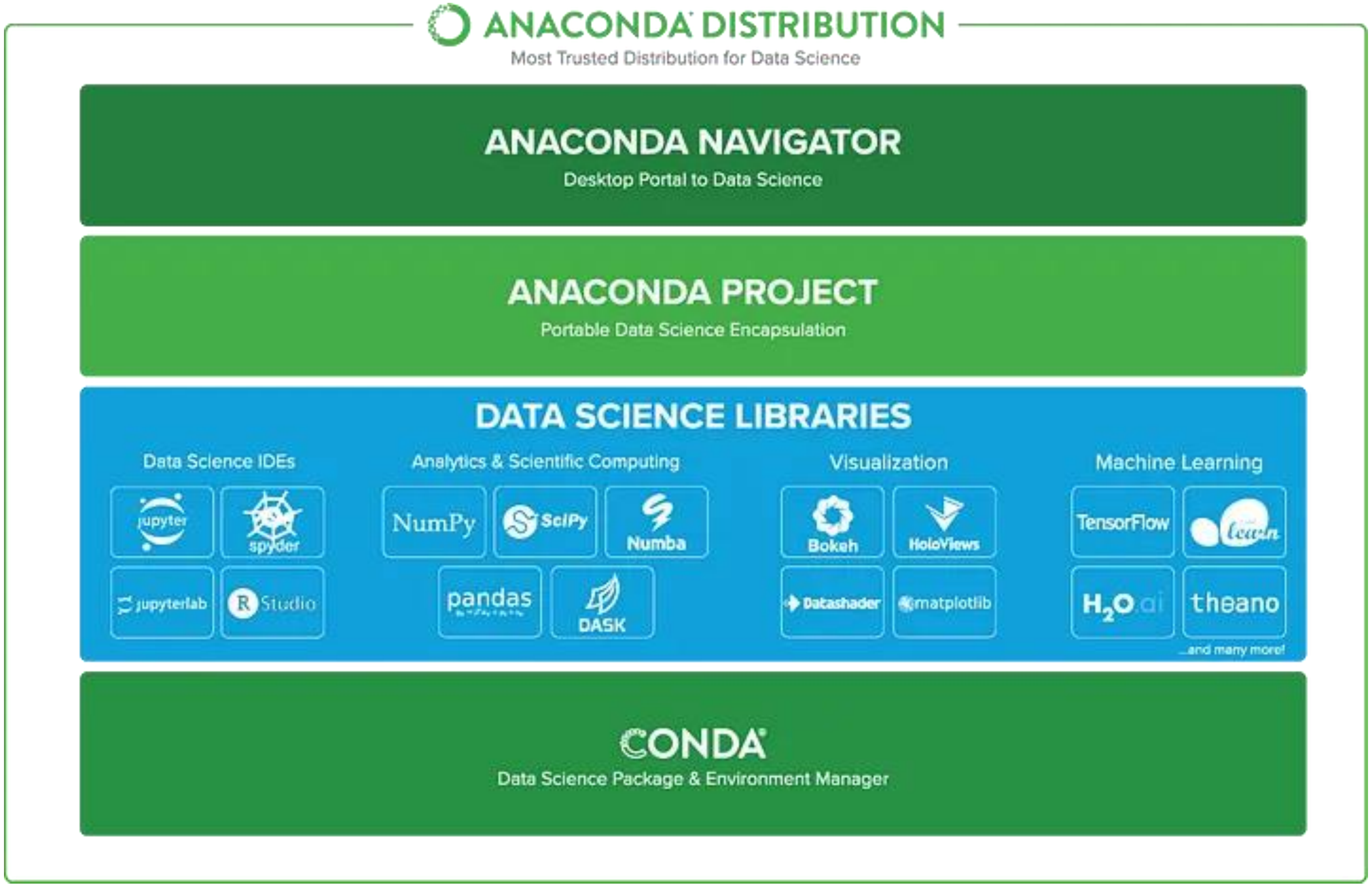
Popular Python Distributions

- **Anaconda**: Comprehensive data science distribution with package management (conda).
- **Miniconda**: Minimal version of Anaconda with conda package manager.
- **Python(x,y)**: Scientific computing distribution with key packages.
- **Enthought Canopy**: Python distribution for scientific and analytic computing.
- **ActivePython**: Commercial distribution with enterprise support.
- **WinPython**: a free open-source portable Python distribution for Windows and scientific and educational usage.

This workshop will focus on the most popular **Anaconda** distribution for data science

2. Python Overview

Anaconda Components



- A graphical user interface (GUI) for Python package management

- A tool helps manage and reproduce data science projects

- Includes the CPython interpreter
- An interactive web-based environment - Jupyter Notebook
- Spyder IDE
- Popular data science packages
- Additional tools and libraries

- Create isolated environments
- Manage dependencies
- Install/update/remove packages

3. Setting up Your Development Environment

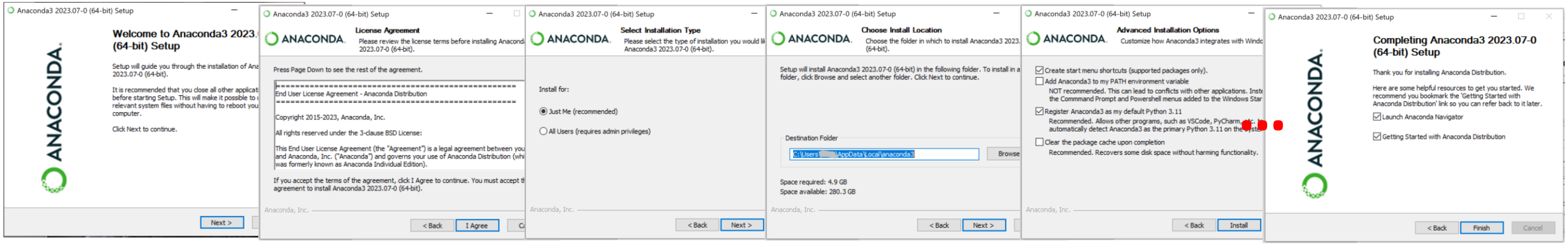
Prepare your coding tools

3. Setting up Your Development Environment

Install Anaconda

Download a version of Anaconda for your computer:
https://www.anaconda.com/download

- Choose the default options to install Anaconda



Welcome to Anaconda3 2023.
(64-bit) Setup

Setup will guide you through the installation of Ana
2023.07-0 (64-bit).

It is recommended that you close all other applicat
before starting Setup. This will make it possible to
relevant system files without having to reboot you
computer.

Click Next to continue.

License Agreement

Please review the license terms before installing Anacond
2023.07-0 (64-bit).

Press Page Down to see the rest of the agreement.

End User License Agreement - Anaconda Distribution

Copyright 2015-2023, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

This End User License Agreement (the "Agreement") is a legal agreement between you
and Anaconda, Inc. ("Anaconda") and governs your use of Anaconda Distribution (whi
was formerly known as Anaconda Individual Edition).

If you accept the terms of the agreement, click I Agree to continue. You must accept th
agreement to install Anaconda3 2023.07-0 (64-bit).

Select Installation Type

Please select the type of installation you would lik
Anaconda3 2023.07-0 (64-bit).

Install for:

● Just Me (recommended)
○ All Users (requires admin privileges)

Choose Install Location

Choose the folder in which to install Anaconda3 2023.
(64-bit).

Setup will install Anaconda3 2023.07-0 (64-bit) in the following folder. To install in a
folder, click Browse and select another folder. Click Next to continue.

Destination Folder: C:\Users\[redacted]\AppData\Local\anaconda3

Space required: 4.9 GB
Space available: 280.3 GB

Advanced Installation Options

Customize how Anaconda3 integrates with Windo

☑ Create start menu shortcuts (supported packages only).
☐ Add Anaconda3 to my PATH environment variable
  NOT recommended. This can lead to conflicts with other applications. Inste
  the Command Prompt and Powershell menus added to the Windows Star
☑ Register Anaconda3 as my default Python 3.11
  Recommended. Allows other programs, such as VSCode, PyCharm, etc. t
  automatically detect Anaconda3 as the primary Python 3.11 on the syste
☐ Clear the package cache upon completion
  Recommended. Recovers some disk space without harming functionality.

Completing Anaconda3 2023.07-0
(64-bit) Setup

Thank you for installing Anaconda Distribution.

Here are some helpful resources to get you started. We recommend you bookmark the 'Getting Started with Anaconda Distribution' link so you can refer back to it later.

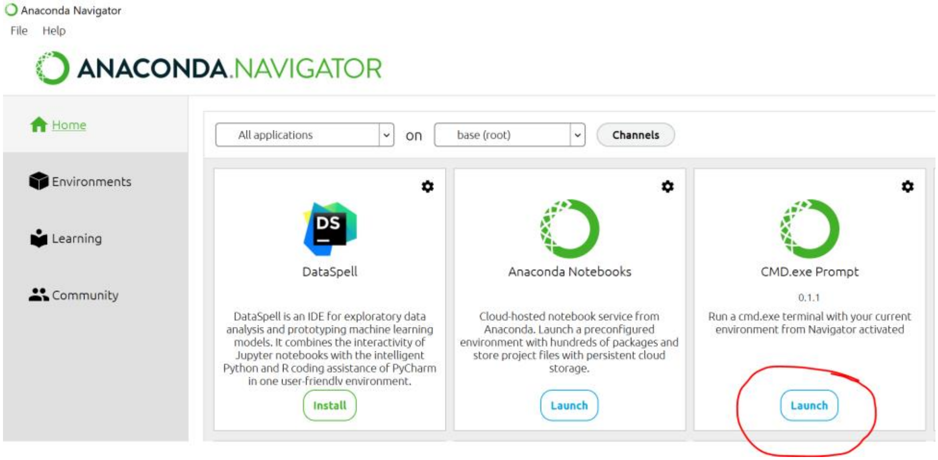
☑ Launch Anaconda Navigator
☑ Getting Started with Anaconda Distribution

Note: this workshop will only focus on the Windows platform

3. Setting up Your Development Environment

Test Anaconda Installation

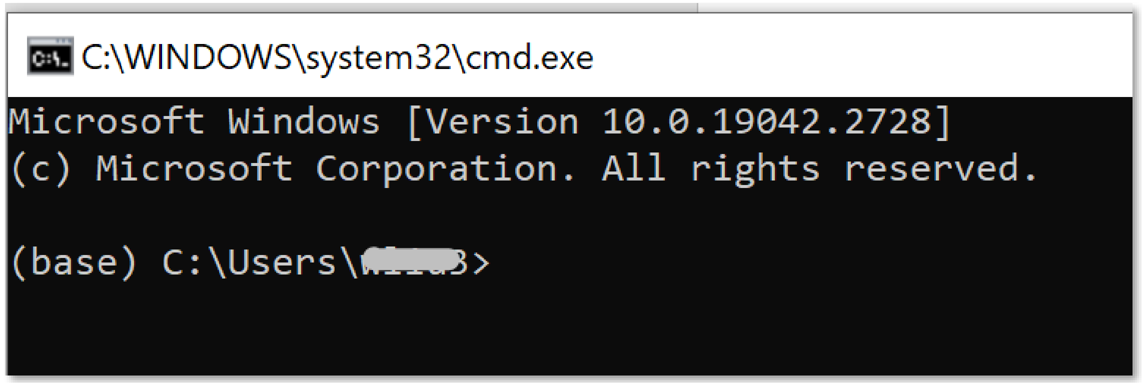
Launch Anaconda **CMD.exe Prompt** from within **Anaconda Navigator**:
(alternatively: Windows *Start* ➔ *Anaconda3* ➔ *Anaconda Prompt*)



DataSpell is an IDE for exploratory data analysis and prototyping machine learning models. It combines the interactivity of Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user-friendlv environment.

Cloud-hosted notebook service from Anaconda. Launch a preconfigured environment with hundreds of packages and store project files with persistent cloud storage.

Run a cmd.exe terminal with your current environment from Navigator activated

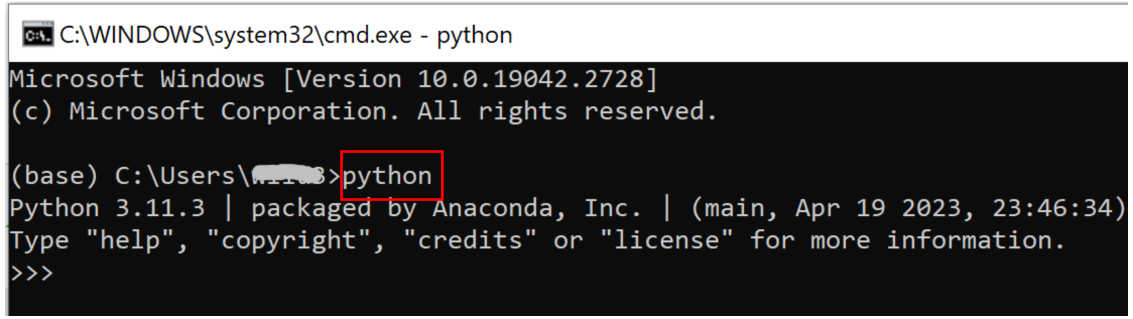


C:\WINDOWS\system32\cmd.exe

```
Microsoft Windows [Version 10.0.19042.2728]
(c) Microsoft Corporation. All rights reserved.

(base) C:\Users\[redacted]>
```

Here I launched the command-line interface of the standard Python interpreter. It's also called the interactive mode, which allows you to interactively write and run Python code statements, providing immediate feedback and results.

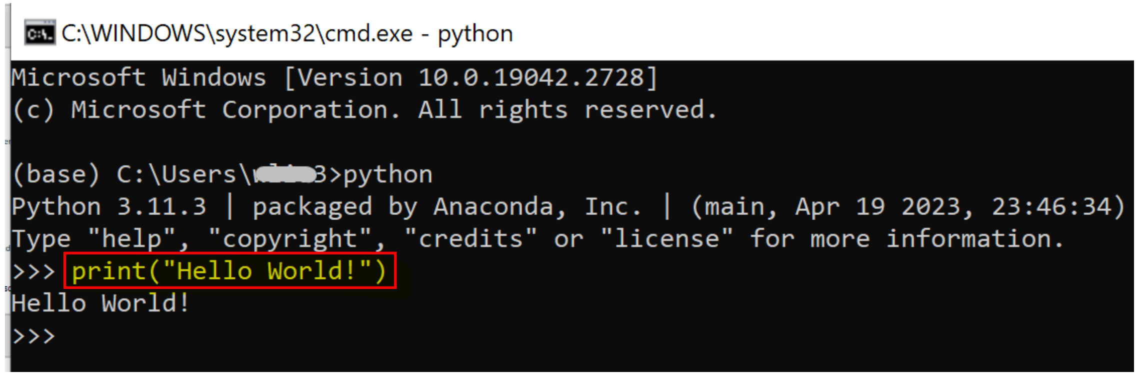
1. Input ***python*** on the CMD line and press Enter key



C:\WINDOWS\system32\cmd.exe - python

```
Microsoft Windows [Version 10.0.19042.2728]
(c) Microsoft Corporation. All rights reserved.

(base) C:\Users\[redacted]>python
Python 3.11.3 | packaged by Anaconda, Inc. | (main, Apr 19 2023, 23:46:34)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2. Input ***print("Hello World")*** and press Enter key



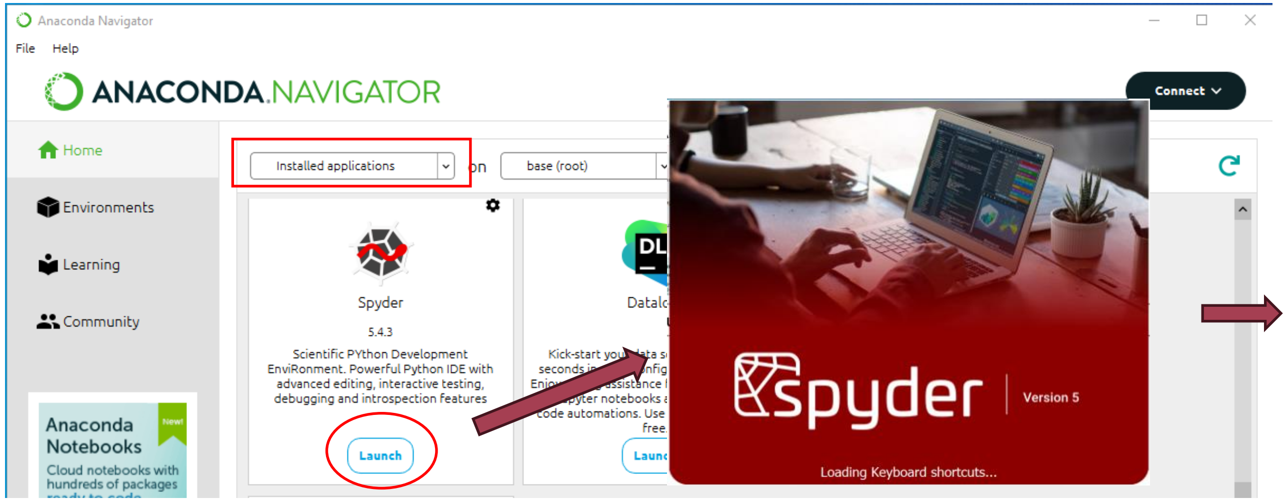
C:\WINDOWS\system32\cmd.exe - python

```
Microsoft Windows [Version 10.0.19042.2728]
(c) Microsoft Corporation. All rights reserved.

(base) C:\Users\[redacted]>python
Python 3.11.3 | packaged by Anaconda, Inc. | (main, Apr 19 2023, 23:46:34)
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

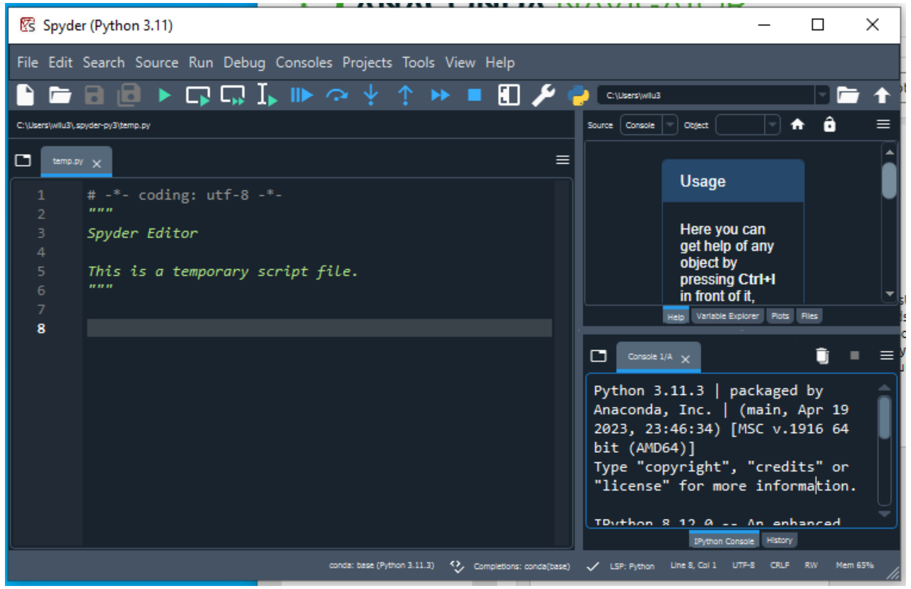
3. Setting up Your Development Environment

Test Anaconda Installation

Launch **Spyder** Integrated Development Environment (IDE) from within **Anaconda Navigator**:



Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features


Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
temp.py
Usage
Help Variable Explorer Plots Files
Console 1/A
IPython Console History

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

```
Python 3.11.3 | packaged by
Anaconda, Inc. | (main, Apr 19
2023, 23:46:34) [MSC v.1916 64
bit (AMD64)]
Type "copyright", "credits" or
"license" for more information.
```

Here you can get help of any object by pressing Ctrl+I in front of it,

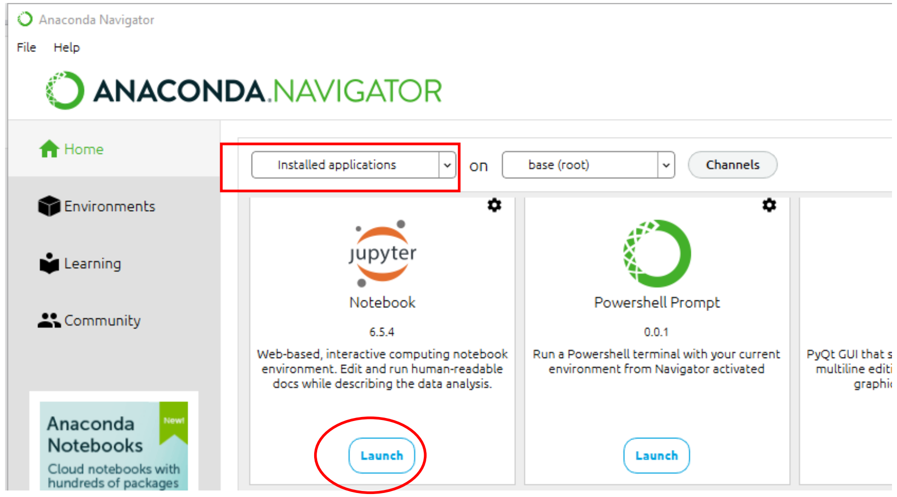
- Open-source IDE for scientific computing and data analysis in Python.
- User-friendly interface with the IPython console and data exploration tools.
- Supports popular scientific libraries, code auto-completion, and debugging.
- Cross-platform compatibility (Windows, macOS, Linux).
- Suitable for beginners and experienced users of scientific computing.

IPython is an interactive command-line shell for Python that provides enhancements over the standard Python interpreter.

3. Setting up Your Development Environment

Test Anaconda Installation

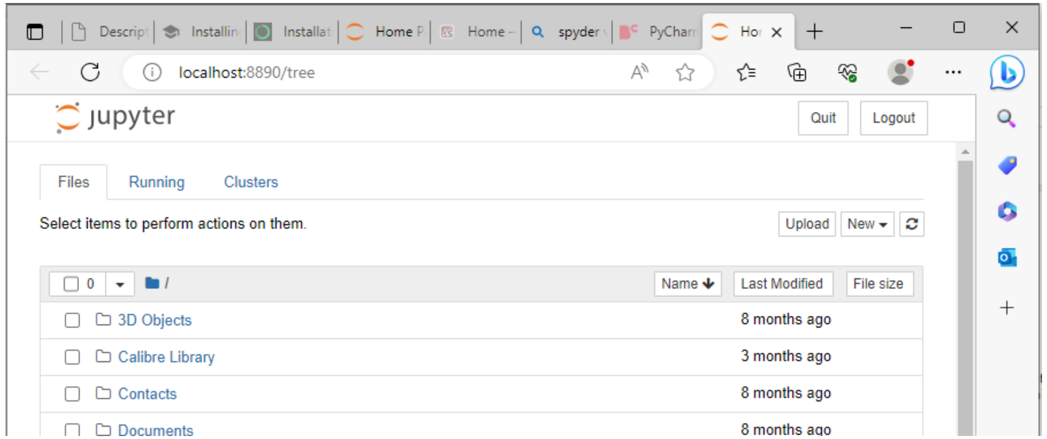
Launch **Jupyter Notebook** from within **Anaconda Navigator**:



Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Run a Powershell terminal with your current environment from Navigator activated

PyQt GUI that s
multiline editi
graphi



Select items to perform actions on them.

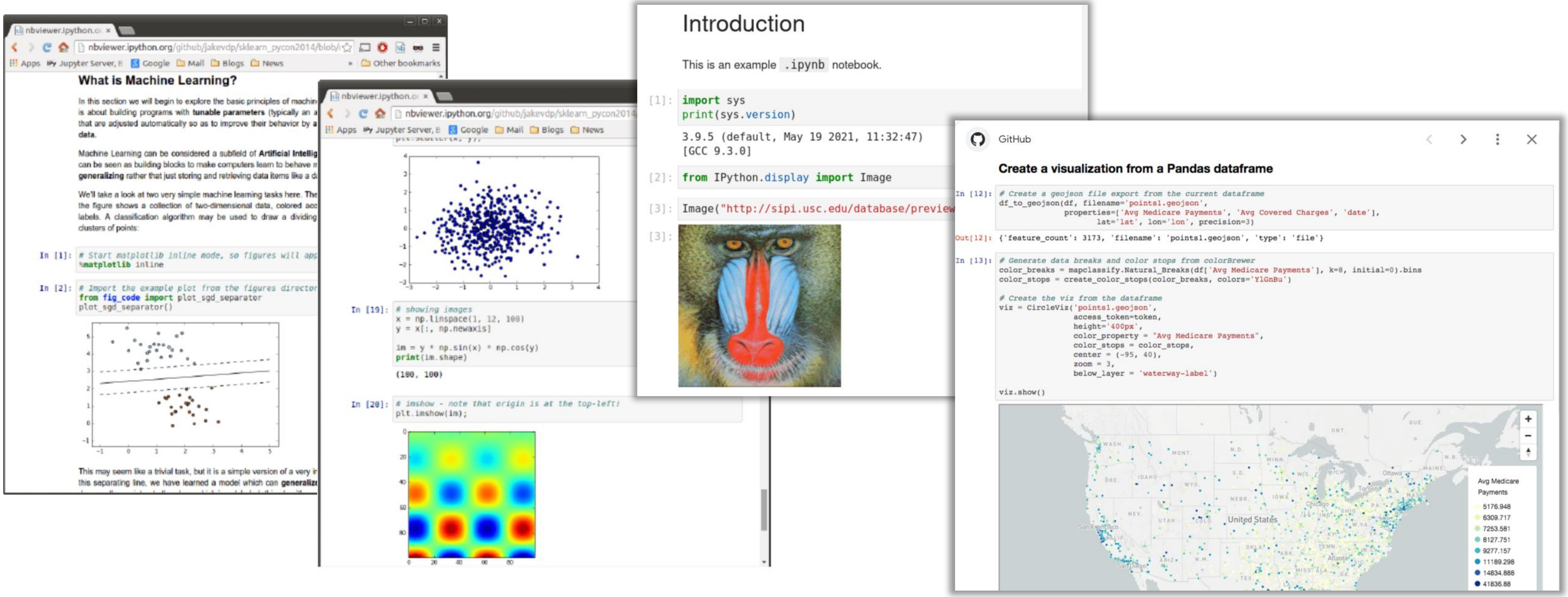
Name	Last Modified	File size
3D Objects	8 months ago	
Calibre Library	3 months ago	
Contacts	8 months ago	
Documents	8 months ago	

What is Jupyter Notebook?

- Web-based interface for creating interactive computational documents.
- Supports multiple programming languages, with Python being popular.
- Combines code cells and text cells using Markdown.
- Enables interactive data analysis, prototyping, and storytelling.
- Features syntax highlighting, code execution, and inline output display.
- Supports creation of interactive visualizations and plots.
- Facilitates collaboration and sharing of notebooks in various formats.
- Integrates with scientific libraries and data analysis tools.
- Has a large and active community with extensive documentation.

3. Setting up Your Development Environment

Jupyter Notebook usage examples



What is Machine Learning?

In this section we will begin to explore the basic principles of machin
is about building programs with **tunable parameters** (typically an a
that are adjusted automatically so as to improve their behavior by a
data.

Machine Learning can be considered a subfield of **Artificial Intellig**
can be seen as building blocks to make computers learn to behave m
generalizing rather that just storing and retrieving data items like a d

We'll take a look at two very simple machine learning tasks here. The
the figure shows a collection of two-dimensional data, colored acc
labels. A classification algorithm may be used to draw a dividing
clusters of points:

```
In [1]: # Start matplotlib inline mode, so figures will app
        %matplotlib inline

In [2]: # Import the example plot from the figures director
        from fig_code import plot_sgd_separator
        plot_sgd_separator()
```

This may seem like a trivial task, but it is a simple version of a very i
this separating line, we have learned a model which can generaliz

```
In [19]: # showing images
         x = np.linspace(1, 12, 100)
         y = x[:, np.newaxis]

         im = y * np.sin(x) * np.cos(y)
         print(im.shape)

         (100, 100)

In [20]: # imshow - note that origin is at the top-left!
         plt.imshow(im);
```

Introduction

This is an example `.ipynb` notebook.

```
[1]: import sys
     print(sys.version)

     3.9.5 (default, May 19 2021, 11:32:47)
     [GCC 9.3.0]

[2]: from IPython.display import Image

[3]: Image("http://sipi.usc.edu/database/preview

[3]:
```

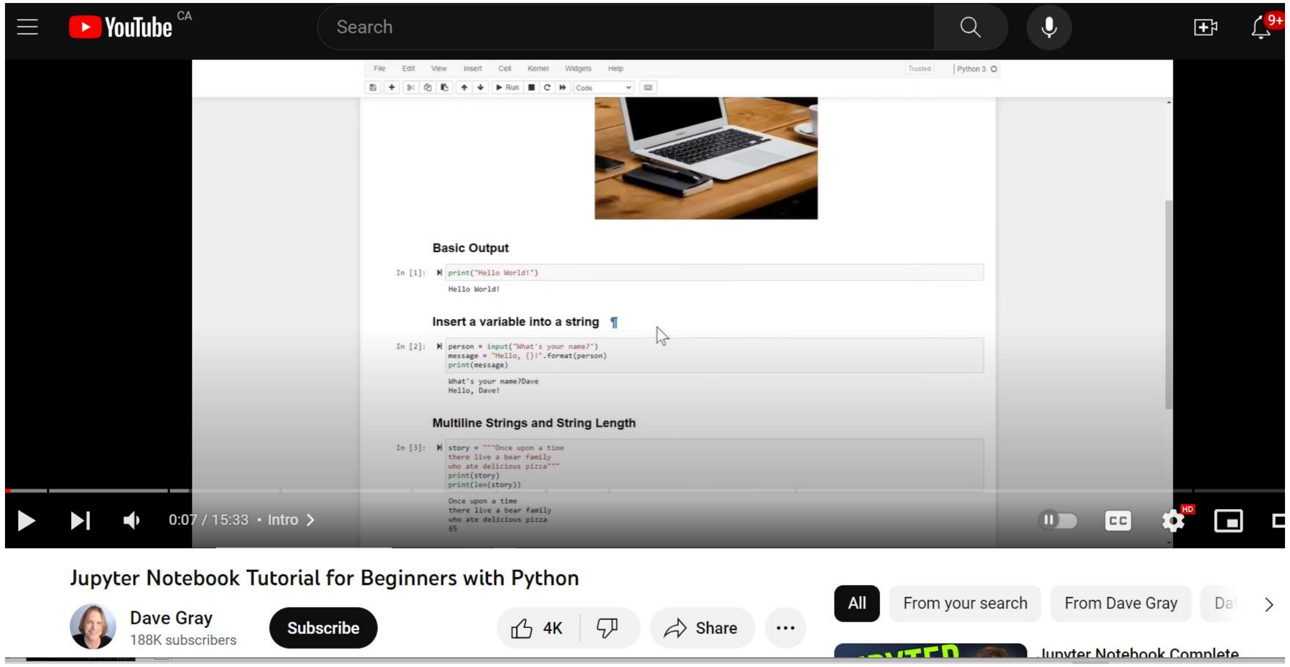
Create a visualization from a Pandas dataframe

```
In [12]: # Create a geojson file export from the current dataframe
         df_to_geojson(df, filename='points1.geojson',
                       properties=['Avg Medicare Payments', 'Avg Covered Charges', 'date'],
                           lat='lat', lon='lon', precision=3)

Out[12]: {'feature_count': 3173, 'filename': 'points1.geojson', 'type': 'file'}

In [13]: # Generate data breaks and color stops from colorBrewer
         color_breaks = mapclassify.Natural_Breaks(df['Avg Medicare Payments'], k=8, initial=0).bins
         color_stops = create_color_stops(color_breaks, colors='YlGnBu')

         # Create the viz from the dataframe
         viz = CircleViz('points1.geojson',
                         access_token=token,
                         height='400px',
                         color_property = "Avg Medicare Payments",
                         color_stops = color_stops,
                         center = (-95, 40),
                         zoom = 3,
                         below_layer = 'waterway-label')

         viz.show()
```

3. Setting up Your Development Environment

Jupyter Notebook How to use?

Watch the following introduction video to learn the basics of Jupter Notebook (15 mins), and then come back.
[Jupyter Notebook Tutorial for Beginners with Python by Dave Gray](https://youtu.be/2WL-XTl2QYI) (https://youtu.be/2WL-XTl2QYI)



Basic Output

```
In [1]:  print("Hello World!")

         Hello World!
```

Insert a variable into a string

```
In [2]:  person = input("What's your name?")
         message = "Hello, {}!".format(person)
         print(message)

         What's your name?Dave
         Hello, Dave!
```

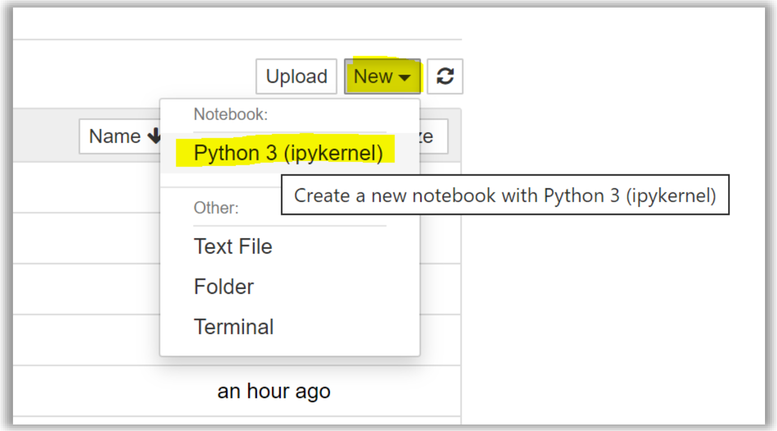
Multiline Strings and String Length

```
In [3]:  story = """Once upon a time
         there live a bear family
         who ate delicious pizza"""
         print(story)
         print(len(story))

         Once upon a time
         there live a bear family
         who ate delicious pizza
         65
```

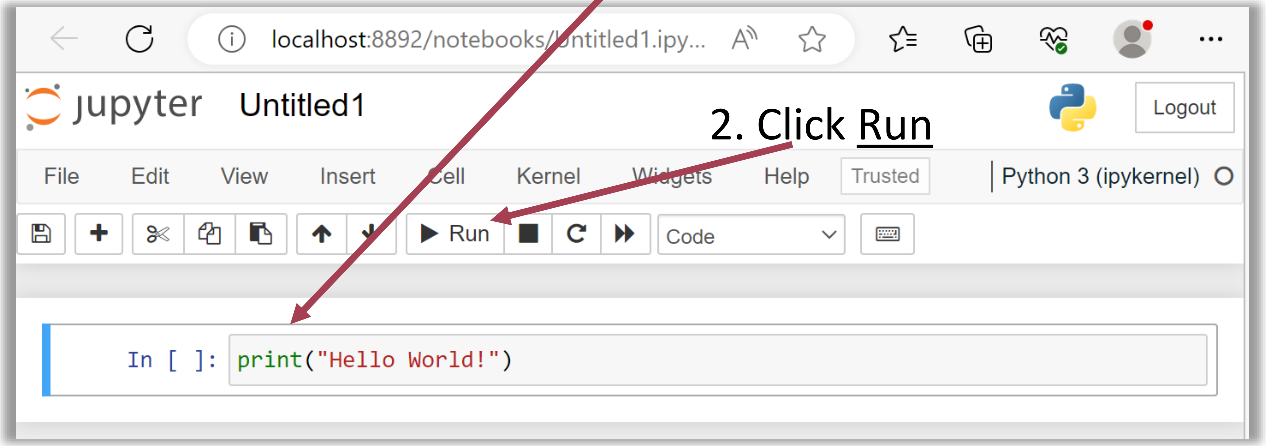
3. Setting up Your Development Environment

Create a new notebook with Python 3

New ➔ Python 3

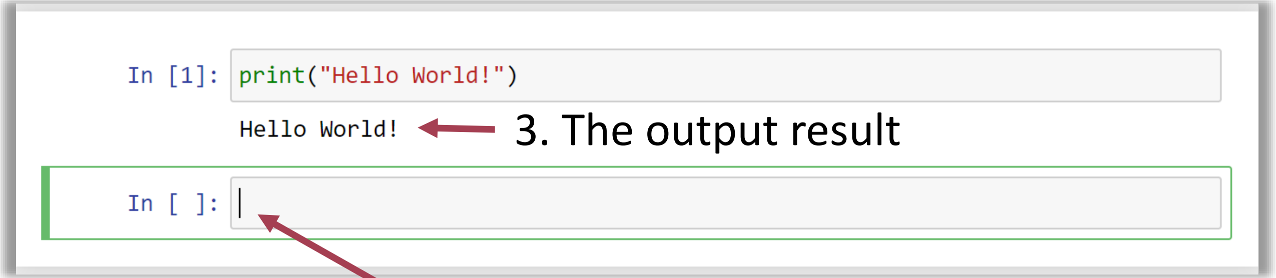


1. Input your Python code here



```
In [ ]: print("Hello World!")
```

2. Click Run



```
In [1]: print("Hello World!")
        Hello World!
In [ ]: |
```

3. The output result

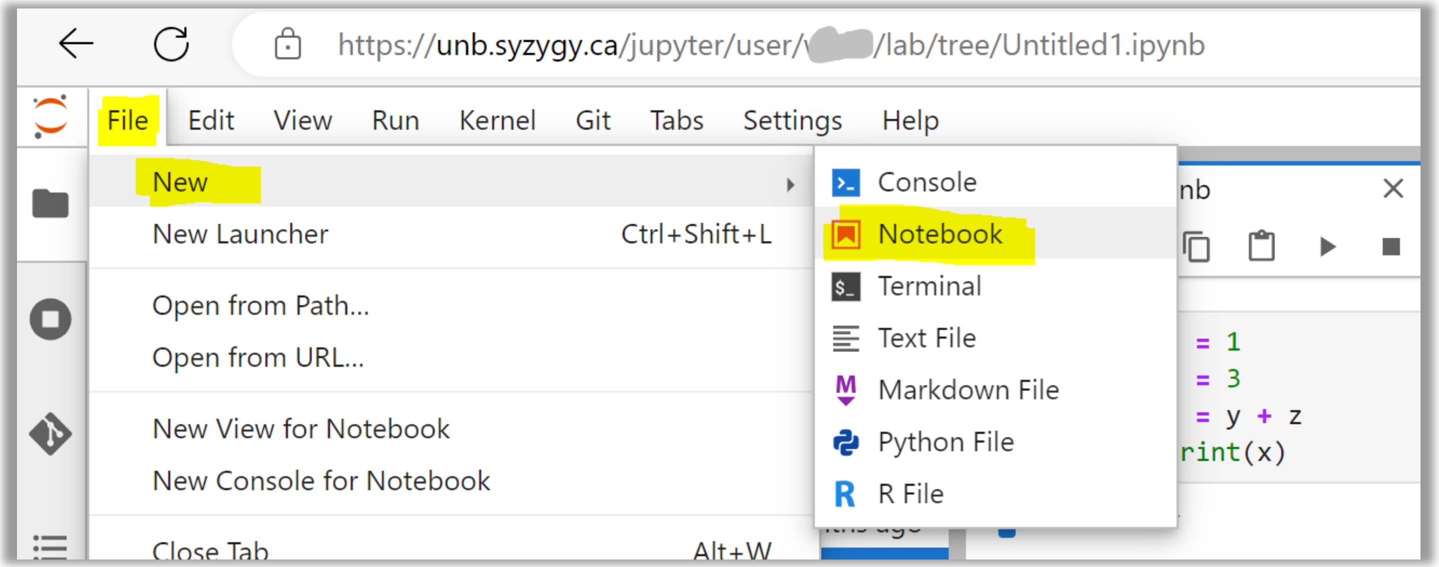
4. Input your new Python code

Please keep this browser window open while you are attending the workshop. Try the code examples in the following sections with this new notebook.

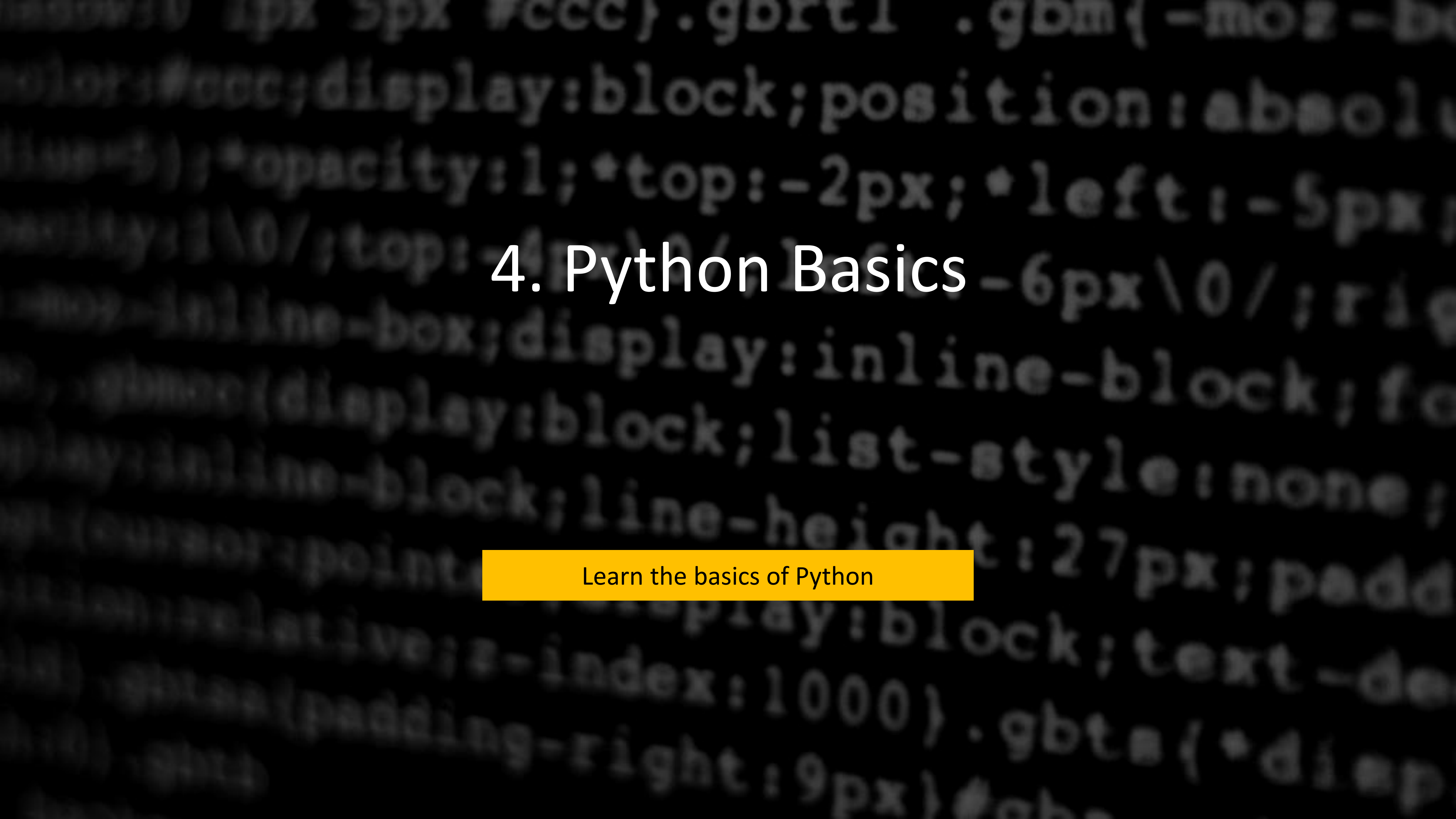
3. Setting up Your Development Environment

For any reason, if you could not use your local version of Jupyter Notebook:

- If you have a UNB ID, you can log in to the cloud-based Jupyter Notebook by Digital Research Alliance of Canada: https://unb.syzygy.ca/



- Otherwise, you can register an account at Anaconda, and use their cloud-based version. https://www.anaconda.com/code-in-the-cloud



4. Python Basics

Learn the basics of Python

4. Python Basics

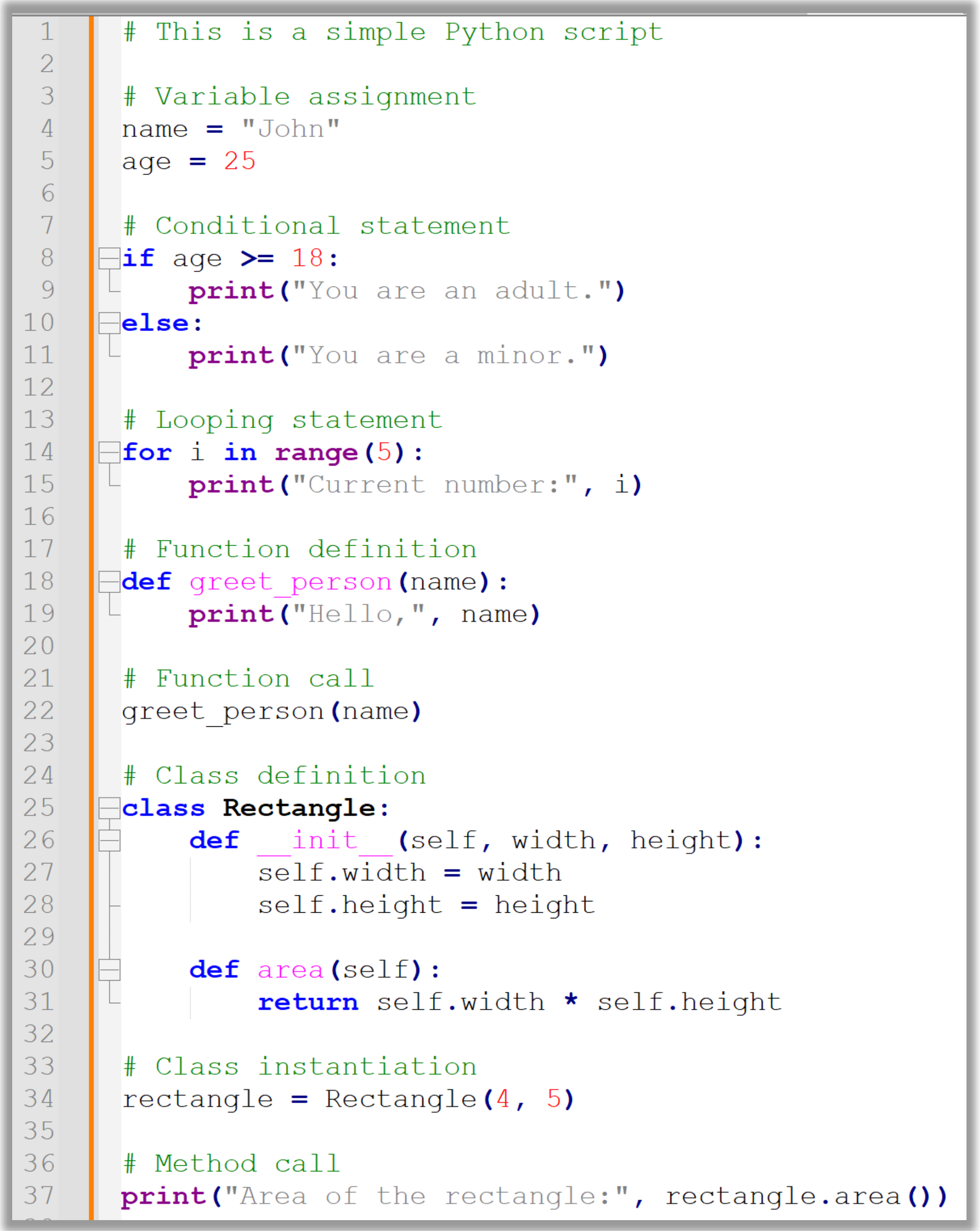
A sample Python script

This script demonstrates:

- Variable assignment and data types
- Conditional statement (if-else)
- Looping statement (for loop)
- Function definition and function call
- Class definition and instantiation
- Method calls

Try to read and understand this simple Python script.

- If you have prior programming experience, it will give you an initial impression of what Python language looks like.
- If you don’t have any experience at all, don’t worry. We will cover them all in the following sessions.



```python
# This is a simple Python script

# Variable assignment
name = "John"
age = 25

# Conditional statement
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")

# Looping statement
for i in range(5):
    print("Current number:", i)

# Function definition
def greet_person(name):
    print("Hello,", name)

# Function call
greet_person(name)

# Class definition
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Class instantiation
rectangle = Rectangle(4, 5)

# Method call
print("Area of the rectangle:", rectangle.area())
```

4. Python Basics

4.0 Comments

Comments can make code more understandable and easier to maintain.

1. Single-line comment:

```
# This is a single-line comment
```

2. Multi-line comment (using triple quotes):

```
"""
This is a
multi-line comment
"""
```

3. Multi-line comment (using multiple single-line comments):

```
# This is a
# multi-line comment
```

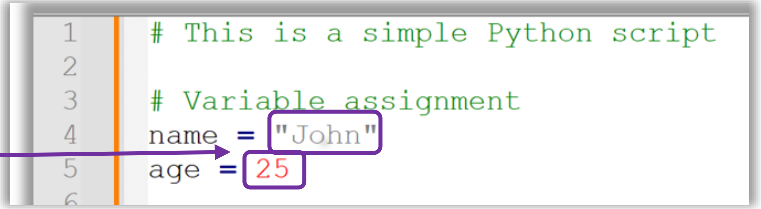
Example

```
# Calculate the sum of two numbers
num1 = 10
num2 = 20
sum = num1 + num2

def calculate_area(length, width):
    """
    Calculates the area of a rectangle.

    Parameters:
    length (float): The length of the rectangle.
    width (float): The width of the rectangle.

    Returns:
    float: The area of the rectangle.
    """
    area = length * width
    return area
```

4. Python Basics

4.1 Literals

In Python, literals are values that are directly represented in code.



```
# This is a simple Python script

# Variable assignment
name = "John"
age = 25
```

Basic type literals

1.Numeric Literals:

- Integer literals: Whole numbers (e.g., 5, -10, 0).
- Floating-point literals: Numbers with decimal points (e.g., 3.14, -0.5).
- Complex literals: Numbers with imaginary parts (e.g., 2+3j, -1j).

2.String Literals:

- Enclosed in single quotes ('...') or double quotes ("...").
- Examples: 'Hello', "Python", "123", 'Special characters: !@#$'.

3.Boolean Literals:

- Two possible values: True and False.

Note: Python is case-sensitive!
So, True != true != TRUE

4.None Literal:

- The special value None represents the absence of a value or a null value.

Literal Type	Example	Description
Numeric Literals	5, 3.14, 2+3j	Represent numerical values
String Literals	'Hello', "Python"	Represent sequences of characters
Boolean Literals	True, False	Represent logical values (True or False)
None Literal	None	Represents the absence of a value or null

4. Python Basics

4.1 Literals

In Python, literals are values that are directly represented in code.

<u>Collection literals</u>

1.List Literals:
- Enclosed in square brackets ([]).
- Examples: [1, 2, 3], ['apple', 'banana', 'cherry'], [True, False].

2.Tuple Literals:
- Enclosed in parentheses (()) or without any brackets.
- Examples: (1, 2, 3), 'apple', 'banana', 'cherry'.

3.Dictionary Literals:
- Enclosed in curly braces ({}) with key-value pairs.
- Examples: {'name': 'John', 'age': 25}, {'fruit': 'apple', 'color': 'red'}.

4.Set Literals:
- Enclosed in curly braces ({}) or created using the set() constructor.
- Examples: {1, 2, 3}, {'apple', 'banana', 'cherry'}, set([True, False]).

Literal Type	**Example**	**Description**
List Literals	[1, 2, 3]	Ordered collection of elements
Tuple Literals	(1, 2, 3) or 1, 2, 3	Ordered collection of immutable elements
Dictionary Literals	{'name': 'John', 'age': 25}	Collection of key-value pairs
Set Literals	{1, 2, 3}	Unordered collection of unique elements

Don't worry if you don't understand it for now.
This page is just for the completeness of literals

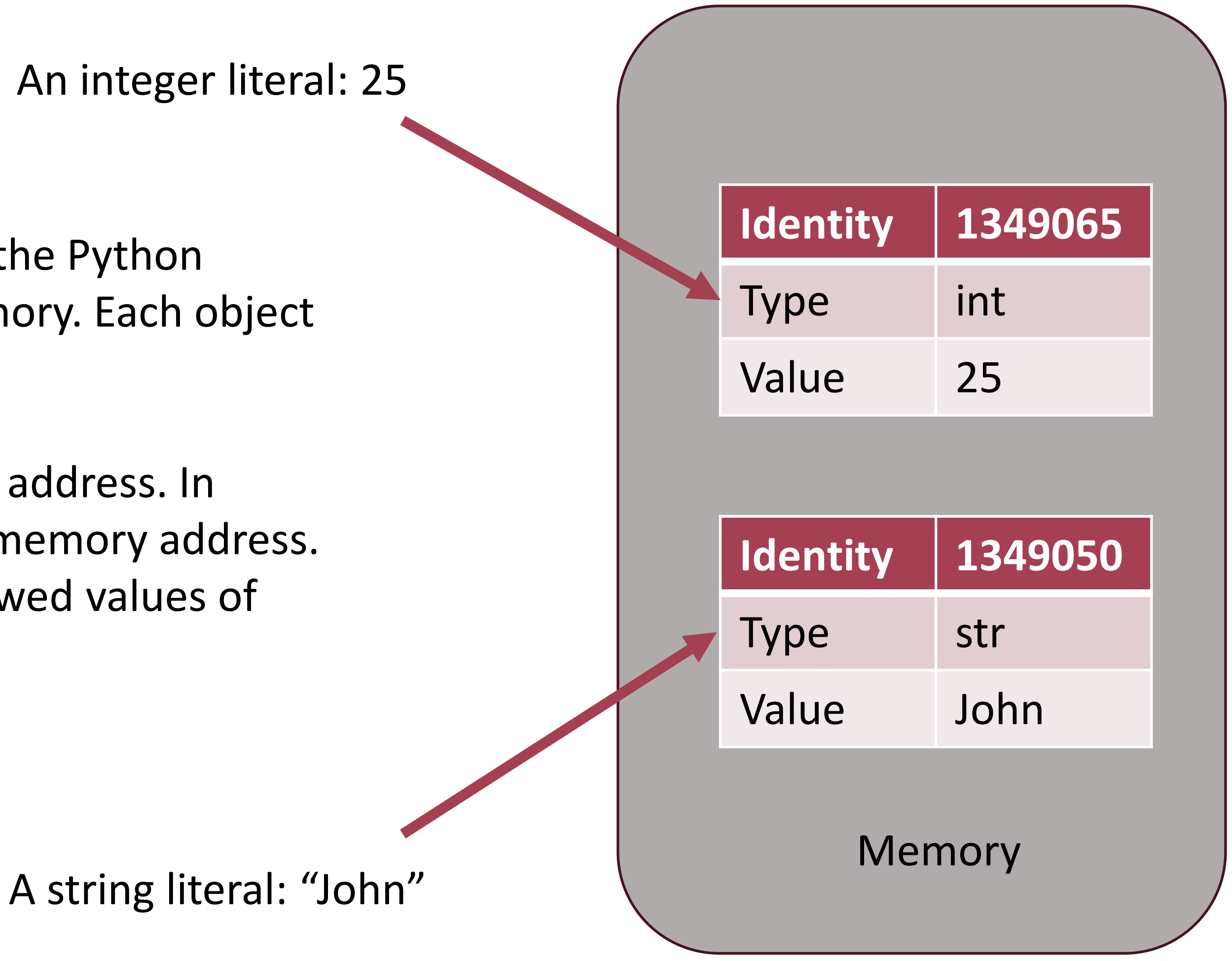
4. Python Basics

4.1 Literals

<u>What will happen to a literal declared in Python?</u>

When a literal is encountered in a Python program, the Python interpreter creates the corresponding object in memory. Each object has three properties:

- **Identity**: Each object occupies a specific memory address. In Python, an object's identity is determined by its memory address.
- **Type**: The type determines the behavior and allowed values of the object.
- **Value**: The literal's content



An integer literal: 25

Identity	**1349065**
Type	int
Value	25

Identity	**1349050**
Type	str
Value	John

Memory

A string literal: "John"

4. Python Basics

4.1 Literals

<u>What will happen to a literal declared in Python?</u>

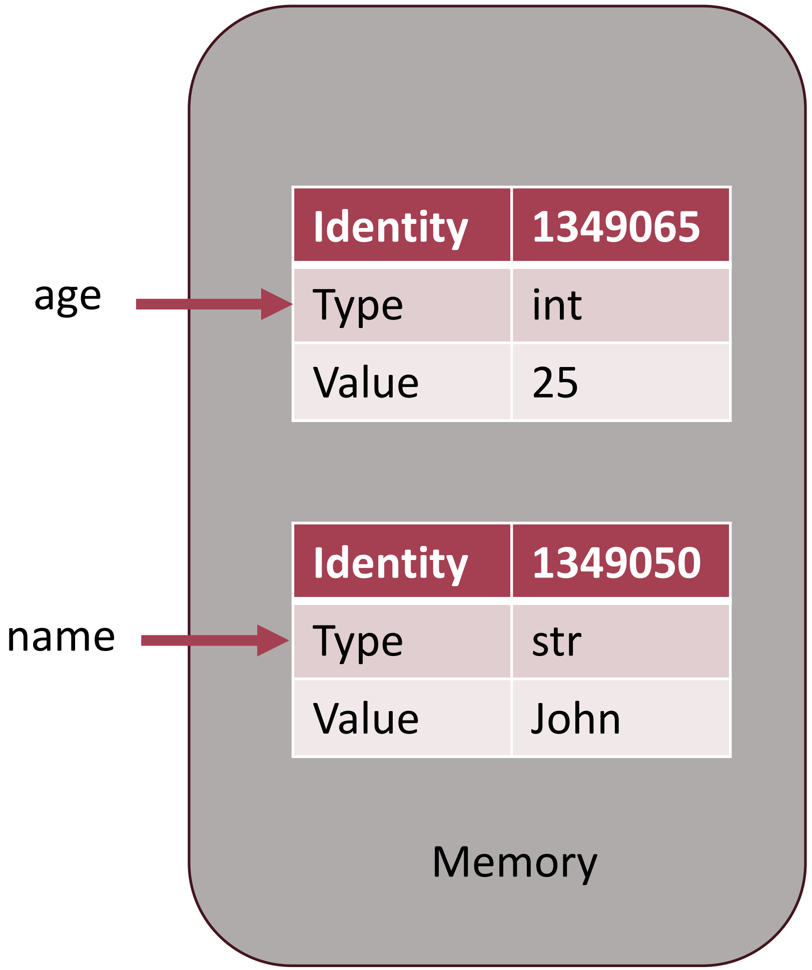
To access the created literals in memory, we need to reference them to continue using their values. We can use the assignment operator = to create a reference to the object.

```python
name = "John"
age = 25
```

`name` and `age` are the symbolic identifiers used for referencing objects and are usually called <u>**variable names**</u>.

Variable in Python is the conceptual container or entity that holds a reference to an object.

Although **variable** and **variable name** are often used interchangeably, it's helpful to be aware of this distinction.



age

Identity	1349065
Type	int
Value	25

name

Identity	1349050
Type	str
Value	John

Memory

4. Python Basics

4.2 Variables

In Python, valid variable names should follow:

1. **Valid Characters**:
   - Variable names can consist of letters (both uppercase and lowercase), digits, and underscores (_).
   - The first character of a variable name cannot be a digit. It must be a letter or an underscore.
2. **Case-Sensitivity**:
   - Python is case-sensitive, so variable names such as `myVar`, `myvar`, and `MYVAR` are all considered different.
3. **Reserved Keywords**:
   - Variable names cannot be the same as Python-reserved keywords. Reserved keywords are special words in the Python language with predefined meanings and cannot be used as variable names. Examples of reserved keywords include `if`, `for`, `while`, `def`, `class`, `import`, and others.
4. **Descriptive and Meaningful**:
   - It is good practice to choose variable names that are descriptive and convey the purpose or meaning of the variable. This improves code readability and maintainability.
5. **Avoid Starting with Underscore**:
   - Variable names starting with a single underscore (_) have a conventional meaning in Python. They are considered "weak internal use" and are typically used for special cases or as a hint to indicate that the variable is intended for internal use within a class or module.
6. **Style Conventions**:
   - Python has style conventions defined in PEP 8 (Python Enhancement Proposal) that recommend using lowercase letters for variable names and separating words with underscores (`snake_case`). This convention helps to improve code readability and consistency.

Valid variable name examples:

```
name
age
my_var
total_count
student1
```

Invalid or not recommended variable names:

```
23name
totalCount
class
nnnnn
```

?

4. Python Basics

4.2 Variables

Reserved keywords that can’t be used as variable names:

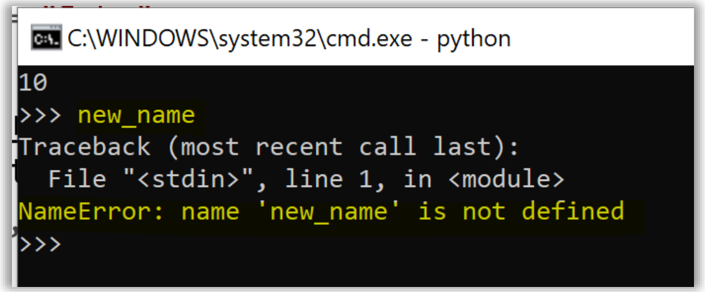
Reserved Keywords				
`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

4. Python Basics

4.2 Variables

- Python has no command for declaring a variable
  A variable is created the moment you first assign a value to it. (as we mentioned before, in the background, a value/object is referenced by a variable name)

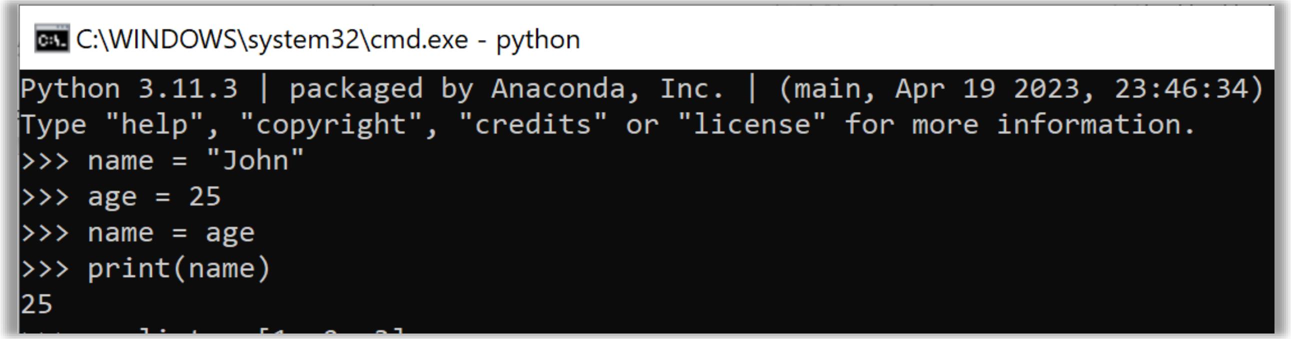
```
name = "John"
age = 25
```



```
10
>>> new_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'new_name' is not defined
>>>
```

- Python is a dynamically typed language, meaning that variables are not explicitly declared with a specific type.
  When you create a literal and assign it to a variable, Python infers the variable’s type based on the literal value. E.g., `age = 25`, 25 is an integer literal, so Python assigns the `int` type to the variable `age`.

- A variable’s type may even be changed later after its initialized
  Python infers the variable’s new type based on the new literal value, or the new variable assigned to it. Try the following statements with your Anaconda **CMD.exe Prompt**.



```
Python 3.11.3 | packaged by Anaconda, Inc. | (main, Apr 19 2023, 23:46:34)
Type "help", "copyright", "credits" or "license" for more information.
>>> name = "John"
>>> age = 25
>>> name = age
>>> print(name)
25
```

Notice that the variable name was changed to reference an integer variable after the assignment statement:
name = age

4. Python Basics

4.3 Operators

Python operators enable you to perform various operations, such as performing mathematical operations, assigning values, comparing values, evaluating logical conditions, manipulating bits, checking membership, testing identity, and applying unary operations.

Operator Category	**Operators**
Arithmetic Operators	`+, -, *, /, //, %, **`
Assignment Operators	`=, +=, -=, *=, /=, //=, %=, **=`
Comparison Operators	`==, !=, >, <, >=, <=`
Logical Operators	`and, or, not`
Bitwise Operators	`&, |, ^, ~, <<, >>`
Membership Operators	`in, not in`
Identity Operators	`is, is not`
Unary Operators	`+, -`

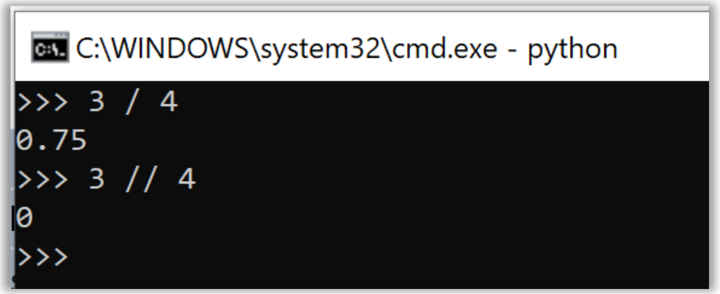
4. Python Basics

4.3 Operators

Arithmetic operators

Operator	Description	Example
+	Addition	2 + 3 equals 5
-	Subtraction	7 - 4 equals 3
*	Multiplication	5 * 6 equals 30
/	Division	10 / 2 equals 5.0
//	Floor Division	10 // 3 equals 3
%	Modulo (Remainder)	10 % 3 equals 1
**	Exponentiation	2 ** 4 equals 16

Note: in Python, the result of division / is always a float number, which is different from C/C++ programs. Floor division // in Python is like the division / in the latter.

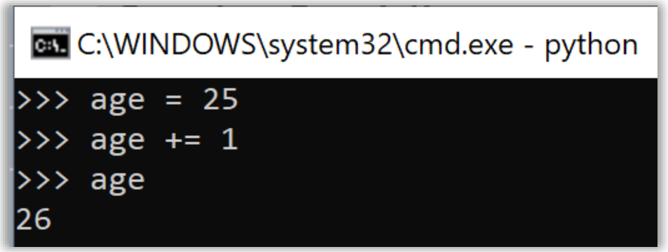


```
>>> 3 / 4
0.75
>>> 3 // 4
0
>>>
```

Assignment operators

Operator	Description	Example
=	Assigns a value to a variable	x = 5 assigns 5 to x
+=	Adds a value and assigns it	x += 3 is equivalent to x = x + 3
-=	Subtracts a value and assigns it	x -= 2 is equivalent to x = x - 2
*=	Multiplies by a value and assigns it	x *= 4 is equivalent to x = x * 4
/=	Divides by a value and assigns it	x /= 2 is equivalent to x = x / 2
//=	Performs floor division and assigns it	x //= 3 is equivalent to x = x // 3
%=	Performs modulo operation and assigns it	x %= 5 is equivalent to x = x % 5
**=	Performs exponentiation and assigns it	x **= 2 is equivalent to x = x ** 2

Note: in Python, there is no increment ++ and decrement operators --. You can use the assignment operator += and -= to achieve the same results.



```
>>> age = 25
>>> age += 1
>>> age
26
```

4. Python Basics

4.3 Operators

Comparison operators

Operator	Description	Example
==	Equal to	5 == 5 evaluates to True
!=	Not equal to	7 != 3 evaluates to True
>	Greater than	10 > 5 evaluates to True
<	Less than	2 < 6 evaluates to True
>=	Greater than or equal to	8 >= 8 evaluates to True
<=	Less than or equal to	4 <= 3 evaluates to False

Unary operators

Operator	Description	Example
+	Unary positive	+5 returns 5
-	Unary negative	-7 returns -7

Logical operators

Operator	Description	Example
and	Returns True if both are True	`True and False` evaluates to `False`
or	Returns True if at least one is True	`True or False` evaluates to `True`
not	Returns the opposite	`not True` evaluates to `False`

Bitwise operators

Operator	Description	Example (Binary)
&	Bitwise AND	0b1100 & 0b0111 returns 0b0100
|	Bitwise OR	0b1100 | 0b0111 returns 0b1111
^	Bitwise XOR (exclusive OR)	0b1100 ^ 0b0111 returns 0b1011
~	Bitwise NOT (complement)	~0b1100 returns -0b1101
<<	Bitwise left shift	0b1010 << 2 returns 0b101000
>>	Bitwise right shift	0b1010 >> 2 returns 0b10

4. Python Basics

4.3 Operators

Membership operators

Operator	Description	Example
`in`	Evaluates if a value is present in a sequence	`'a' in ['a', 'b', 'c']` returns `True`
`not in`	Evaluates if a value is not present in a sequence	`'d' not in ['a', 'b', 'c']` returns `True`

Identity operators

Operator	Description	Example

`is`	Evaluates if two objects are the same object	`x = 5`
		`y = x`
		`x is y` returns `True`

`is not`	Evaluates if two objects are not the same	`x = 5`
		`y = 10`
		`x is not y` returns `True`

4. Python Basics

4.3 Operator Precedence

Python operator precedence determines the order in which operators are evaluated in an expression. When multiple operators exist in an expression, Python follows a specific order of precedence to evaluate them. Here's a summary of the operator precedence in Python, from highest to lowest precedence:

Precedence	Operators
1	`()` (Parentheses)
2	`**` (Exponentiation)
3	`*, /, //, %` (Multiplication, Division, Floor Division, Modulo)
4	`+, -` (Addition, Subtraction)
5	`<<, >>` (Bitwise Shifts)
6	`&` (Bitwise AND)
7	`^` (Bitwise XOR)
8	`|`(Bitwise OR)
9	`==, !=, >, <, >=, <=, is, is not, in, not in` (Comparison Operators)
10	`not` (Logical NOT)
11	`and` (Logical AND)
12	`or` (Logical OR)

4. Python Basics

4.3 Operator Precedence

Ensure that your code is more readable, maintainable, and less prone to operator precedence-related errors, follow the best practice:

Best practice:

- **Use parentheses for clarity in complex expressions**.

```
# Complex expression without parentheses
result = 2 + 3 * 4 / (5 - 1) ** 2
# Preferred: Complex expression with parentheses for clarity
result = 2 + ((3 * 4) / ((5 - 1) ** 2))
```

- **Follow Python's operator precedence rules**.
- **Add whitespace for readability**.

```
result = 2+3*4   # Less readable
result = 2 + 3 * 4   # More readable with whitespace
```

- **Break down complex expressions when necessary**.

```
complex_result = (2 + 3 * 4) / (5 - 1)   # Complex expression
# Preferred: Breaking down the expression
intermediate_result = 3 * 4
complex_result = (2 + intermediate_result) / (5 - 1)
```

- **Comment on unusual or non-intuitive precedence**.

```
# Non-intuitive precedence: Comment to clarify the order of evaluation
result = (2 + 3) * 4  # Add 2 and 3 first, then multiply by 4
# Unusual precedence: Comment to explain the order of operations
result = (2 + 3) ** 4  # Add 2 and 3 first, then raise the sum to the power of 4
```

4. Python Basics

4.4 Control Structures

Python provides various control structures that enable you to control the flow and execution of your code.

Conditionals

The **if** statement

```
if condition:
    # Code to be executed if the condition is true
    statement1
    statement2
    # ...
else:
    # Code to be executed if the condition is false
    statement3
    statement4
    # ...
```

The **if elif** statement

```
if condition1:
    # Code to be executed if condition1 is true
    statement1
    statement2
    # ...
elif condition2:
    # Code to be executed if condition1 is false and
    # condition2 is true
    statement3
    statement4
    # ...
elif condition3:
    # Code to be executed if both condition1 and condition2
    # are false and condition3 is true
    statement5
    statement6
    # ...
else:
    # Code to be executed if none of the above conditions are
    # true
    statement7
    statement8
    # ...
```

4. Python Basics

4.4 Control Structures

Python provides various control structures that enable you to control the flow and execution of your code.

Conditionals

Python control structure formatting:

- **Indentation**: Python defines code blocks using indentation (usually 4 spaces) instead of curly braces ({ }). Consistent indentation with spaces or tabs is crucial for proper code structure and readability. (If you use tab, make sure to set one tab equal to 4 spaces in your editor settings)
- **Indentation Block Marker**: The colon (:) indicates the beginning of an indented code block. It is typically used after a statement that requires an indented block, such as in `if`, `else`, `elif`, `for`, `while`, and function definitions. The code block following the colon must be indented consistently to maintain the block structure.

A nested **if** statement

```
if condition1:
    # Outer if block
    statement1
    statement2
    # ...

    if condition2:
        # Nested if block
        statement3
        statement4
        # ...
    else:
        # Nested else block
        statement5
        statement6
        # ...

    # More statements after the nested if-else blocks
    statement7
    statement8
    # ...
else:
    # Outer else block
    statement9
    statement10
    # ...
```

4. Python Basics

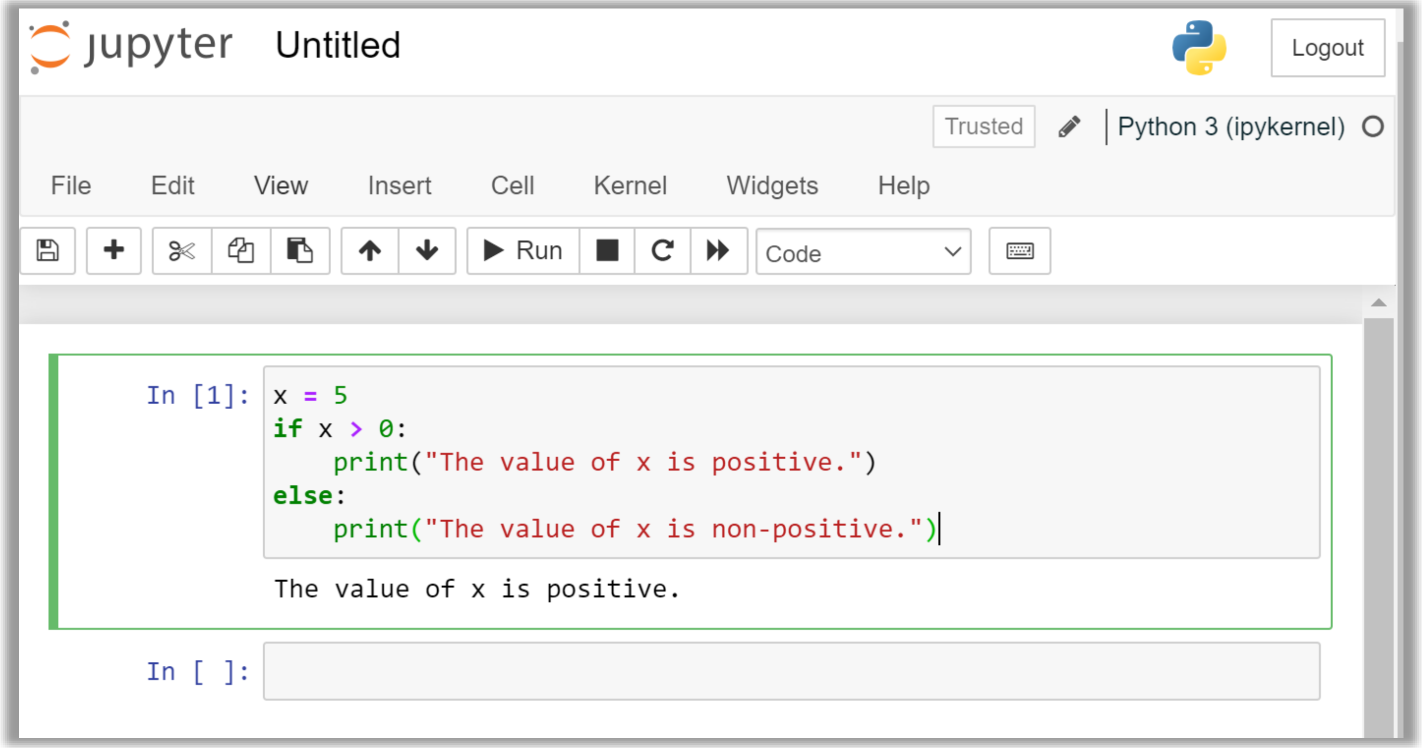
4.4 Control Structures

Conditionals

Example 1

```
x = 5

if x > 0:
    print("The value of x is positive.")
else:
    print("The value of x is non-positive.")
```

Learning by doing is an effective way to master a language. Try each example with your new notebook created before.



```
In [1]: x = 5
        if x > 0:
            print("The value of x is positive.")
        else:
            print("The value of x is non-positive.")

        The value of x is positive.

In [ ]:
```

4. Python Basics

4.4 Control Structures

Conditionals

Example 2

Automatic type conversion

```python
x = 5
y = 3.14

# Comparing an integer and a float
if x > y:
    print("x is greater than y")
elif x < y:
    print("x is less than y")
else:
    print("x is equal to y")
```

When comparing an integer with a float, Python will automatically perform type conversion and convert the integer to a float for comparison.

Example 3

Comparing two close float numbers

```python
x = 0.1 + 0.1 + 0.1  # 0.3
y = 0.3

tolerance = 1e-10  # Set your desired tolerance level

if abs(x - y) < tolerance:
    print("x is approximately equal to y")
else:
    print("x is not equal to y")
```

Avoid using simple equality comparisons (== or !=) due to potential precision issues when comparing close float numbers. Instead, use a tolerance-based approach by checking if the absolute difference falls within an acceptable range. This accounts for floating-point limitations and provides more reliable approximate equality comparisons.

4. Python Basics

4.4 Control Structures

Conditionals

Conditional expressions

How to simplify this block?

```
if x == 1:
    var = 20
else:
    var = 30
```

```
var = (20 if x == 1 else 30)
```

The conditional expression provides a concise way to express a simple conditional statement. It allows you to make a decision and return different values based on the outcome of the condition in a single line of code.

Conditional expression template

```
value_if_true if condition else value_if_false
```

In this template:

- `condition` represents the expression or condition that is evaluated.
- `value_if_true` is the value to be returned if the condition evaluates to True.
- `value_if_false` is the value to be returned if the condition evaluates to False.

4. Python Basics

4.4 Control Structures

Loops

while statement

```
while condition:
    # Code to be executed while the condition is true
    statement1
    statement2
    # ...
```

Example 1

```
count = 0

while count < 5:
    print("Count:", count)
    count += 1

print("Loop finished.")
```

Note: `print()` is a Python built-in function to display or output text or values to the console. We will introduce functions later.

for statement

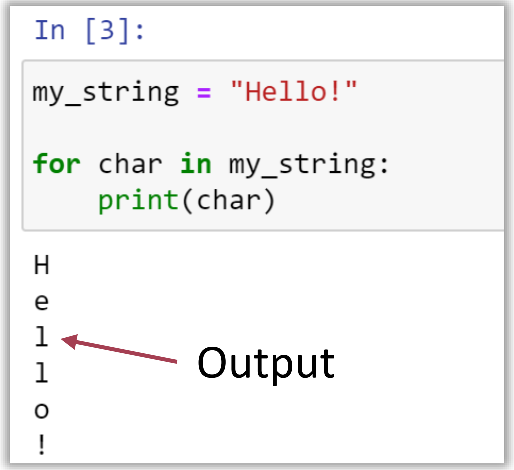
```
for item in iterable:
    # Code to be executed for each item in the iterable
    statement1
    statement2
    # ...
```

In this template:

- **item** represents a variable that takes on the value of each item in the iterable object during each iteration of the loop.
- **iterable** is an object that can be looped or iterated over. It's like a container or a sequence with multiple elements, such as a list of items, a string of characters, or even a range of numbers.

Example 2

```
my_string = "Hello!"

for char in my_string:
    print(char)
```



```
In [3]:
my_string = "Hello!"

for char in my_string:
    print(char)

H
e
l
l
o
!
```

Output

4. Python Basics

4.4 Control Structures

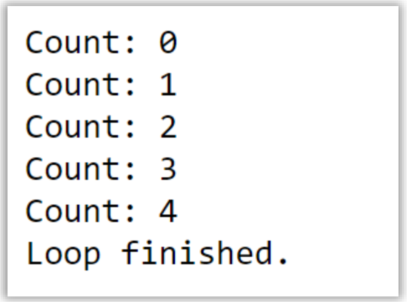
Loops

Terminate a loop: use the `break` keyword

When the `break` statement is encountered within a loop, the program flow immediately exits the loop, skipping any remaining iterations or code within the loop.

Example 1

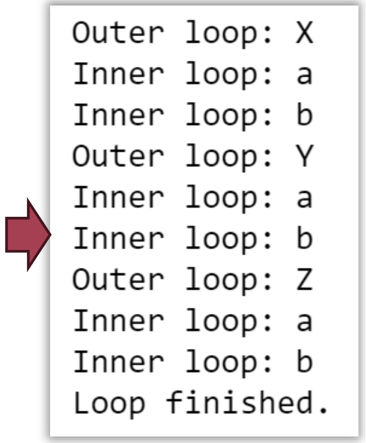
```python
count = 0

while True:
    print("Count:", count)
    count += 1

    if count >= 5:
        break

print("Loop finished.")
```



```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
Loop finished.
```

Example 2

```python
outer_string = "XYZ"
inner_string = "abc"

for outer_char in outer_string:
    print("Outer loop:", outer_char)

    for inner_char in inner_string:
        print("Inner loop:", inner_char)

        if inner_char == "b":
            break

print("Loop finished.")
```



```
Outer loop: X
Inner loop: a
Inner loop: b
Outer loop: Y
Inner loop: a
Inner loop: b
Outer loop: Z
Inner loop: a
Inner loop: b
Loop finished.
```

Note: The break statement only exits the innermost loop in nested loops. If you have multiple nested loops, the break statement will break out of the innermost loop and resume the execution from the next statement outside that loop.

4. Python Basics

4.4 Control Structures

<u>Loops</u>

<u>The **`while-else`** block to detect a loop break</u>

The `while-else` block allows you to handle scenarios where you want to perform certain actions if a **`while`** loop completes normally without encountering a `break` statement.

```
while condition:
    # Code to be executed inside the loop
    statement1
    statement2
    # ...

    if break_condition:
        # Code to be executed if break
        # condition is met
        break

else:
    # Code to be executed if the loop
    # completes without encountering a break
    statement3
    statement4
    # ...
```

Example

```
my_string = "Hello, World!"
target = “Z"

index = 0
while index < len(my_string):
    if my_string[index] == target:
        break
    index += 1
else:
    print("Target not found in the string.")
```

`my_string[index]` means taking the nth character of the string. Don’t worry about the syntax here and the `len()` function, which means taking the length of `my_string`.

```
Target not found in the string.
```

4. Python Basics

4.4 Control Structures

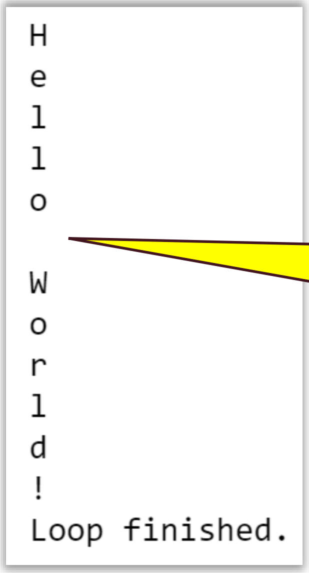
Loops

Use the **`continue`** keyword to abort the current iteration

The `continue` keyword allows you to skip certain iterations of a loop when a particular condition is met. It effectively aborts the current iteration and proceeds to the next iteration without executing the remaining code within the loop for that iteration.

Example

```python
my_string = "Hello, World!"

for char in my_string:
    if char == ",":
        continue
    print(char)

print("Loop finished.")
```



```
H
e
l
l
o
 
W
o
r
l
d
!
Loop finished.
```

Notice there is an empty row, and the “,” sign never got printed here

4. Python Basics

4.4 Control Structures

Loops

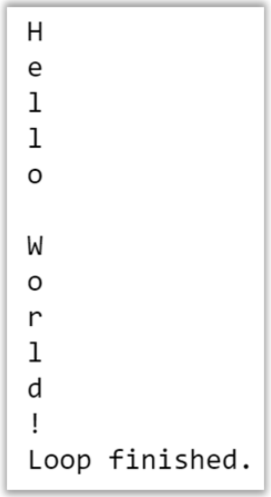
Use the **`pass`** statement as a placeholder for an unimplemented code block

Purpose: The `pass` statement is used as a placeholder when you need a statement syntactically but don't want to perform any specific action at that point.
Usage: It is commonly used in situations where you are defining a code block, such as in function or class definitions, conditional statements, or loops, but you want to skip that block without causing a syntax error.

```
if condition:
    pass  # Placeholder, no action needed for now
else:
    # Some code here
    statement1
    statement2
```

Example

```
my_string = "Hello, World!"

for char in my_string:
    if char == ",":
        pass  # No action needed for commas
    else:
        print(char)

print("Loop finished.")
```



```
H
e
l
l
o
 
W
o
r
l
d
!
Loop finished.
```

4. Python Basics

4.5 Data Types

A data type refers to the category of values that a particular object can have. It determines the kind of operations that can be performed on the object and the storage format for the data. Some built-in data types in Python include numeric types (`int, float, complex`), string (`str`), boolean (`bool`), `list`, `tuple`, `dict`, `set`, and more.

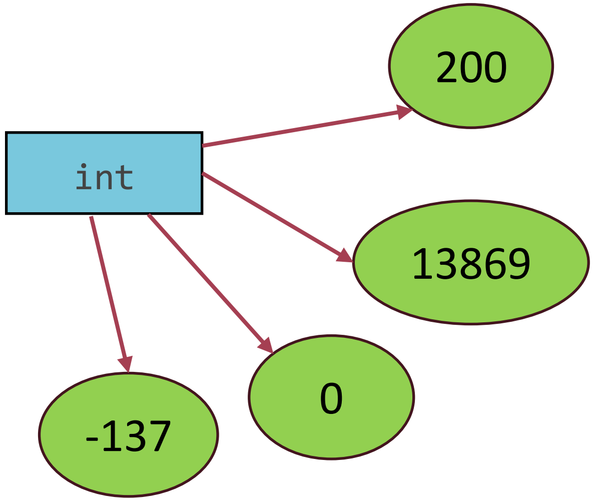
Data Type	**Description**	**Example**	**Mutability**
`NoneType`	Represents the absence of a value	`result = None`	Immutable
Numeric Types			
`int`	Represents whole numbers	`x = 5`	Immutable
`float`	Represents floating-point numbers	`pi = 3.14`	Immutable
`bool`	Represents binary values of either True or False	`is_valid = True`	Immutable
`complex`	Represents complex numbers with real and imaginary parts	`z = 2 + 3j`	Immutable
Sequential Types			
`str`	Represents a sequence of characters	`message = "Hello, World!"`	Immutable
`bytes`	Represents immutable sequences of bytes	`data = b'Hello'`	Immutable
`tuple`	Represents an ordered collection of elements	`coordinates = (10, 20)`	Immutable
`list`	Represents an ordered collection of elements	`numbers = [1, 2, 3, 4]`	Mutable
`bytearray`	Represents mutable sequences of bytes	`data = bytearray(b'Hello')`	Mutable
Set and Mapping Types			
`set`	Represents an unordered collection of unique elements	`unique_numbers = {1, 2, 3}`	Mutable
`dict`	Represents a collection of key-value pairs	`person = {'name': 'John', 'age': 25}`	Mutable
`frozenset`	Represents an immutable, unordered collection of unique elements	`frozen_numbers = frozenset({1, 2, 3})`	Immutable

4. Python Basics

4.5 Data Types

Objects and Data Types

- In Python, everything is an object, which refers to the concept that all entities in the language, including values, data structures, functions, and even classes, are objects.
- Every object in Python is an instance of a specific data type or class, and objects are created based on those data types.



- ☐ In this example, all these numbers are the instances of the `int` type (integer)
- ☐ The `int` type specifies the storage format and behaviors of the instances (numbers)
  - Storage Format: The `int` type specifies the specific format used to represent integer numbers in memory. Internally, integer numbers are stored using a fixed amount of memory, typically 32 bits or 64 bits, depending on the platform.
  - Behaviors: The `int` type defines the behaviors and operations that can be performed on integer numbers. For example, the int type supports mathematical operations like addition, subtraction, multiplication, and division and comparison operations like equality and inequality checks.

4. Python Basics

4.5 Data Types

NoneType

The `NoneType` is a special data type in Python that represents the absence of a value or the lack of a value. It is commonly associated with the singleton object `None`, which is a built-in constant. It is often used in conditional statements or function returns to handle cases where a value is missing or not applicable.

- Singleton Object: **None** is a unique instance of **NoneType** in the system.
- Comparisons: Use **is** operator to check if a variable refers to **None**.
- Usage: Commonly used to handle missing or non-applicable values.
- Immutable: The value of **None** cannot be changed once assigned.

Example

```
name = None
if name is None:
    print("No name provided")
```

In this example, we initialize the variable name with **None**, indicating that no specific value has been assigned yet. We then use the **is** operator to check if name is **None**.

4. Python Basics

4.5 Data Types

Numeric Types

All numeric types (`int`, `float`, `bool`, `complex`) are immutable, meaning that their value cannot be changed once related objects are created.

Numeric Types: `int`

<u>Range</u>: The `int` type can represent a wide range of integer values, limited only by the available memory in the system. On most platforms, the range of int values is approximately -2^31 to 2^31-1 for 32-bit systems and -2^63 to 2^63-1 for 64-bit systems.

Example: access the largest and smallest integer constant on your system

```
import sys

largest_int = sys.maxsize
smallest_int = -sys.maxsize - 1

# Output: 9223372036854775807 (or platform-dependent)
print(largest_int)
# Output: -9223372036854775808 (or platform-dependent)
print(smallest_int)
```

- If a memory can hold an integer value up to 2^31-1, it would require a memory capacity of approximately 2 gigabytes (GB) on a 32-bit system;
- To hold an integer up to 2^63-1 on a 64-bit system, the memory capacity must be around **1,048,576 TB** !!!!!

4. Python Basics

4.5 Data Types

Numeric Types: `float`

Finite Range: The range of finite float numbers is limited by the specific implementation and the floating-point format used. In Python, the range is typically from approximately 2.2 x 10^-308 to 1.8 x 10^308 for positive and negative values.
Not-a-Number (NaN): The floating-point format includes a special NaN (Not-a-Number) value. NaN represents the result of mathematically undefined or indeterminate operations, such as dividing zero by zero or taking the square root of a negative number.
Precision limitations: floating-point arithmetic is subject to precision limitations due to the nature of representing real numbers in a finite format. This can lead to rounding errors and small discrepancies when calculating float numbers.

```
import sys

largest_float = sys.float_info.max
smallest_float = -sys.float_info.max
# Output: 1.7976931348623157e+308 (or platform-dependent)
print(largest_float)
# Output: -1.7976931348623157e+308 (or platform-dependent)
print(smallest_float)
```

Example: access the largest and smallest float number constant on your system

4. Python Basics

4.5 Data Types

Numeric Types: `bool`

In Python, the `bool` type is considered a numeric type because it shares certain characteristics and behaviors with other numeric types. It can participate in numeric operations, is internally represented as integers (`True` as `1`, `False` as `0`), and can be converted to other numeric types.

For each other data type, a specific value is defined as bool `False`, as listed below`.` Except these, all other values are defined as `True`.

Objects	**Example**	**Description**
False	`False`	The boolean value False.
None	`None`	The special object representing absence of a value.
Numeric Zero	`0, 0.0, 0j`	Numeric values equal to zero.
Empty Sequences	`'', [], (), set(), {}`	Empty sequences like strings, lists, tuples, sets, and dictionaries.
Empty Objects	Custom objects with special "empty" state	Custom objects that define their own truthiness behavior. They evaluate as False when their length or internal state is empty.

4. Python Basics

4.5 Data Types

Numeric Types: `bool`

Other objects can also be converted to `bool` type using the built-in function `bool()`.

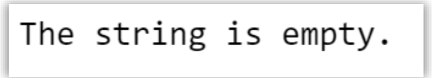
By using a logical operator `not`, the operand that is defined as False will be automatically interpreted as a truth value.

Example 1

```
string = ""

if bool(string) == False:
    print("The string is empty.")
else:
    print("The string is not empty.")
```

Example 2

```
string = ""

if not string:
    print("The string is empty.")
else:
    print("The string is not empty.")
```



```
The string is empty.
```

4. Python Basics

4.5 Data Types

Numeric Types:

Built-in functions in Python that can be used for converting between numeric data types:

Function	Description	Example Usage	Result
`int(x)`	Converts x to an integer	`int(3.14)`	`3`
`float(x)`	Converts x to a float	`float(5)`	`5.0`
`complex(real, imag)`	Creates a complex number with the given real and imaginary parts	`complex(2, 3)`	`(2+3j)`
`bin(x)`	Converts an integer x to a binary string representation	`bin(10)`	`'0b1010'`
`oct(x)`	Converts an integer x to an octal string representation	`oct(20)`	`'0o24'`
`hex(x)`	Converts an integer x to a hexadecimal string representation	`hex(15)`	`'0xf'`
`round(x, n)`	Rounds x to n decimal places	`round(3.14159, 2)`	`3.14`
`abs(x)`	Returns the absolute value of x	`abs(-10)`	`10`
`divmod(a, b)`	Returns the quotient and remainder of a divided by b as a tuple	`divmod(10, 3)`	`(3, 1)`
`bool(x)`	Converts x to a boolean value	`bool(0)`	`False`

4. Python Basics

4.5 Data Types

Sequential Types

In Python, a sequential type is a data type that represents an ordered sequence of elements. Sequential types allow accessing and manipulating individual elements within the sequence based on their positions or indices.

Sequential Types

Data Type	**Description**	**Example**	**Mutability**
`str`	Represents a sequence of characters	`message = "Hello, World!"`	Immutable
`tuple`	Represents an ordered collection of elements (could be in different types)	`coordinates = (10, 20)`	Immutable
`list`	Represents an ordered collection of elements (could be in different types)	`numbers = [10, 23, 30, 4]`	Mutable
`range`	Represents a sequence of numbers within a specified range	`range1 = range(0, 10)`	Immutable

- `str`, `tuple`, and `list` are general-purpose sequential types used for different purposes. `str` is used for working with text, `tuple` for fixed collections, and `list` for dynamic collections that can be modified.
- The `range` type in Python is a special sequential data type that serves a specific purpose. It generates numbers on-the-fly as they are needed, rather than storing all the numbers explicitly in memory. This makes range objects memory-efficient, particularly for large ranges. Used primarily for iterating over a specific range of numbers.

4. Python Basics

4.5 Data Types

Sequential Types: Creating new objects

```
# Creating a string
greeting = "Hello"
sentence1 = 'Python is a versatile programming language.'
Sentence2 = 'Python is a "versatile" programming language.'

# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)
mixed_tuple = ('apple', 3.14, True)

# Creating a list
my_list1 = [1, 2, 3, 4, 5]
my_list2 = [True, False, True]
mixed_list = [True, 'apple', 3.14]

# Creating a range
my_range = range(1, 6)
custom_range = range(0, 10, 2)
```

The range() function accepts one, two, or three arguments, and returns an object of the range type, representing a sequence of numbers that can be iterated over.

- `range(stop)`: Generates numbers starting from `0` up to `stop` (exclusive) with a default step size of `1`.
- `range(start, stop)`: Generates numbers starting from `start` up to `stop` (exclusive) with a default step size of `1`.
- `range(start, stop, step)`: Generates numbers starting from `start` up to `stop` (exclusive) with a specified step size.

```
# Generating numbers from 0 to 4 (exclusive)
for num in range(5):
    print(num)  # Output: 0, 1, 2, 3, 4

# Generating numbers from 1 to 6 (exclusive)
for num in range(1, 6):
    print(num)  # Output: 1, 2, 3, 4, 5

# Generating even numbers from 2 to 10 (exclusive)
for num in range(2, 10, 2):
    print(num)  # Output: 2, 4, 6, 8
```

4. Python Basics

4.5 Data Types

Sequential Types: Indexing

```
my_string = "Python"
```

Index from front	0	1	2	3	4	5
Elements	P	y	t	h	o	n
Index from back	-6	-5	-4	-3	-2	-1

- **Indexing is zero-based**: In Python, indexing starts from 0, so the first element in a sequence has an index of 0, the second element has an index of 1, and so on.

- **Positive indexing**: You can access elements sequentially using positive indices. For example, `my_list[0]` retrieves the first element of the list, `my_string[2]` retrieves the third character of the string, and `my_tuple[1]` retrieves the second element of the tuple.

- **Negative indexing**: Python also supports negative indexing, which allows you to access elements from the end of the sequence. For example, `my_list[-1]` retrieves the last element of the list, `my_string[-2]` retrieves the second-to-last character of the string, and `my_tuple[-3]` retrieves the third-to-last element of the tuple.

- **Out-of-range indexing**: If you try to access an index that is outside the valid range of indices for the sequence, Python raises an *IndexError*. For example, attempting to access `my_list[6]` when my_list has only 6 elements will result in an *IndexError*.

4. Python Basics

4.5 Data Types

Sequential Types: Operations on objects

1. Indexing: Accessing individual elements by their position using square brackets `[]`.

```
my_string = 'Hello'
my_tuple = (1, 2, 3)
my_list = [True, False, True]
print(my_string[0]) ⟶ Output: 'H'
print(my_tuple[2]) ⟶ Output: 3
print(my_list[1]) ⟶ Output: False
```

2. Slicing: Extracting a subset of elements using a range of indices.

```
my_string = 'Hello World'
my_tuple = (1, 2, 3, 4, 5)
my_list = [True, False, True, False, True]
print(my_string[1:5]) ⟶ Output: 'ello'
print(my_tuple[2:4]) ⟶ Output: (3, 4)
print(my_list[:3]) ⟶ Output: [True, False, True]
```

3. Membership: Checking if an element is present in the sequence using the **`in`** operator.

```
my_string = 'Hello World'
my_tuple = (1, 2, 3)
my_list = [True, False, True]
print('o' in my_string) ⟶ Output: True
print(2 in my_tuple) ⟶ Output: True
print(False in my_list) ⟶ Output: True
```

The slicing syntax follows a common pattern:
`s[start:end:step]`

- `start` (optional): The index where the slice begins. If omitted, the slice starts from the beginning (index 0).
- `end` (optional, exclusive): The index where the slice ends. The slice does not include this index. If omitted, the slice goes up to the end of the sequence.
- `step` (optional): The step or increment between elements in the slice. If omitted, the default step is 1. A positive step moves forward through the sequence, while a negative step moves backward.

```
my_string = 'Hello World'
print(my_string[1:5])   # Output: 'ello'
print(my_string[:5])    # Output: 'Hello'
print(my_string[6:])    # Output: 'World'
print(my_string[::2])   # Output: 'HloWrd'
print(my_string[::-1])  # Output: 'dlroW olleH'

my_list = [1, 2, 3, 4, 5]
print(my_list[1:4])     # Output: [2, 3, 4]
print(my_list[:3])      # Output: [1, 2, 3]
print(my_list[2:])      # Output: [3, 4, 5]
print(my_list[::2])     # Output: [1, 3, 5]
print(my_list[::-1])    # Output: [5, 4, 3, 2, 1]
```

4. Python Basics

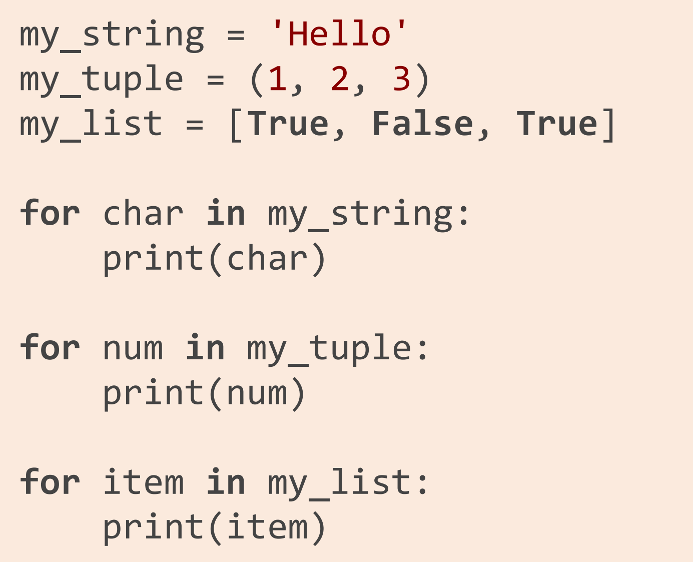
4.5 Data Types

Sequential Types: Operations on objects

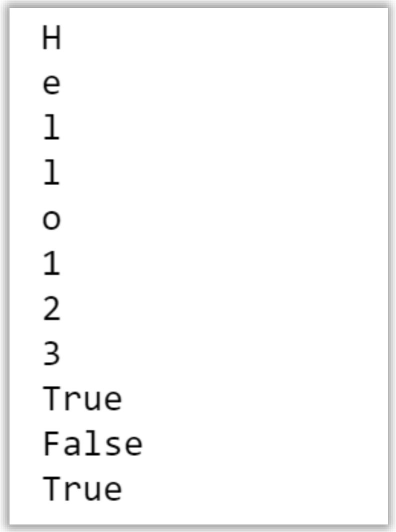
4. Equality: Comparing if two sequences are equal using the == operator.

```
str1 = 'Hello'
str2 = 'World'
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
list1 = [True, False]
list2 = [1, 2, 3]
print(str1 == str2) ⟶ Output: False
print(tuple1 == tuple2) ⟶ Output: False
print(list1 == list2) ⟶ Output: False
```

5. Iteration: Looping over each element in the sequence using a `for` loop.



```
my_string = 'Hello'
my_tuple = (1, 2, 3)
my_list = [True, False, True]

for char in my_string:
    print(char)

for num in my_tuple:
    print(num)

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



```
H
e
l
l
o
1
2
3
True
False
True
```

4. Python Basics

4.5 Data Types

Sequential Types: Operations on objects

6. Returns the maximum value from a sequence using the **`max()`** function.

```
my_string = 'Hello World'
max_char = max(my_string)
print(max_char) ⟶ Output: 'r'
my_tuple = (5, 2, 8, 1, 9, 3)
max_value = max(my_tuple)
print(max_value) ⟶ Output: 9
```

7. Returns the minimum value from a sequence using the **`min()`** function.

```
my_list = [5, 2, 8, 1, 9, 3]
min_value = min(my_list)
print(min_value) ⟶ Output: 1
my_tuple = (5, 2, 8, 1, 9, 3)
min_value = min(my_tuple)
print(min_value) ⟶ Output: 1
```

8. Returns the number of occurrences of a specified element in a sequence using the **`count()`** function.

```
my_string = 'Hello World'
count = my_string.count('l')
print(count) ⟶ Output: 3
my_list = [1, 2, 3, 4, 1, 5, 1]
count = my_list.count(1)
print(count) ⟶ Output: 3
```

4. Python Basics

4.5 Data Types

Sequential Types: Operations on objects

9. Length: Getting the number of elements in the sequence using the **`len()`** function.

```
my_string = 'Hello World'
my_tuple = (1, 2, 3, 4, 5)
my_list = [True, False, True]
print(len(my_string)) ⟶ Output: 11
print(len(my_tuple)) ⟶ Output: 5
print(len(my_list)) ⟶ Output: 3
```

10. Returns the index of the first occurrence of a specified element in a sequence using the **`index()`** function

```
my_string = 'Hello World'
index = my_string.index('W')
print(index) ⟶ Output: 6
my_list = [5, 2, 8, 1, 9, 3]
index = my_list.index(8)
print(index) ⟶ Output: 2
```

11. Repetition: Repeating the elements using the ***** operator. (*Not applicable to the **range** object*)

```
my_string = 'abc'
my_tuple = (1, 2)
my_list = [True, False]
print(my_string * 3) ⟶ Output: 'abcabcabc'
print(my_tuple * 2) ⟶ Output: (1, 2, 1, 2)
print(my_list * 4) ⟶ Output: [True, False, True, False, True, False, True, False]
```

4. Python Basics

4.5 Data Types

Sequential Types: Operations on objects

12. Concatenation: Joining or combining multiple sequences using the **+** operator. (*Not applicable to the **range** object*)

```
str1 = 'Hello'
str2 = 'World'
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
list1 = [True, False]
list2 = [1, 2, 3]
print(str1 + ' ' + str2) ⟶ Output: 'Hello World'
print(tuple1 + tuple2) ⟶ Output: (1, 2, 3, 4, 5, 6)
print(list1 + list2) ⟶ Output: [True, False, 1, 2, 3]
```

13. Mutation: Modifying elements or structure of the sequence. (*Only applicable to the mutable object*)

```
my_list = [1, 2, 3, 4, 5]
my_list[0] = 'New'
my_list.append(6)
print(my_list) ⟶ Output: ['New', 2, 3, 4, 5, 6]
```

Note: A tuple is an immutable object, meaning you cannot directly assign a new value to the tuple or modify other elements. However, if it contains a list object as its element, that list object could still be updated.

```
my_tuple = ([1, 2, 3], 4, 5)
print(my_tuple)  # Output: ([1, 2, 3], 4, 5)

my_tuple[0].append(4)
print(my_tuple)  # Output: ([1, 2, 3, 4], 4, 5)
```

4. Python Basics

4.5 Data Types

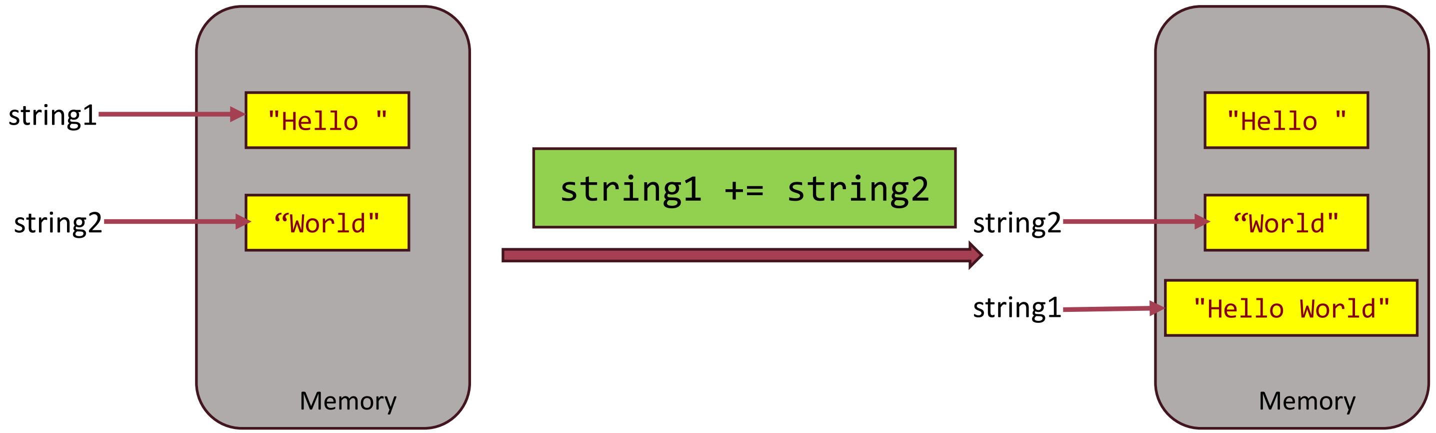
Sequential Types: Operations on objects

More on mutations

Considering the right example, it seems like the immutable `string1` is updated by the assignment operator, which is not the case. Actually, a new string object is created with the value of `string1 + string2`, which is then referenced by `string1`. So, after that operation, the three objects `"Hello "`, `"World"`, and `"Hello World"` exist in the memory, whereas there is no reference to `"Hello "` anymore.

```python
string1 = "Hello "
string2 = "World"
string1 += string2
print(string1)
```

```
Hello World
```



4. Python Basics

4.5 Data Types

Sequential Types: the `str` type

The `str` type in Python represents a sequence of characters and is used to store and manipulate textual data. Here are key points summarizing the `str` type:

- **Textual Data**: `str` objects hold strings of characters, enclosed in single quotes ('') or double quotes (""). They can represent words, sentences, paragraphs, or any other text-based information.
- **Immutable**: `str` objects are immutable, meaning their contents cannot be changed after creation. Any operation that modifies a string actually creates a new string object.
- **Type Conversion**: You can convert other data types to `str` using the `str()` function, which creates a string representation of the given object.
- **String Interpolation**: Python offers multiple ways to format strings and allows you to create dynamic strings without the need for excessive concatenation or complex formatting operations.
- **String Methods**: `str` objects have many built-in methods that allow you to perform operations like case conversion, string formatting, searching, replacing, splitting, and more.

4. Python Basics

4.5 Data Types

Sequential Types: the `str` type

String Interpolation: Python offers multiple ways to format strings and allows you to create dynamic strings without the need for excessive concatenation or complex formatting operations.

1. Concatenation: You can concatenate strings and variables using the + operator.

```
name = "Alice"
age = 25
message = "My name is " + name + " and I am " + str(age) + " years old."
```

2. `%-formatting`: This method uses the % operator to format strings. You provide a format specifier and pass the values in a tuple.

```
name = "Alice"
age = 25
message = "My name is %s and I am %d years old." % (name, age)
```

3. `str.format()`: This method uses the `str.format()` function to format strings. You can specify placeholders in the string and provide the values in the format() function.

```
name = "Alice"
age = 25
message = "My name is {} and I am {} years old.".format(name, age)
```

4. Python Basics

4.5 Data Types

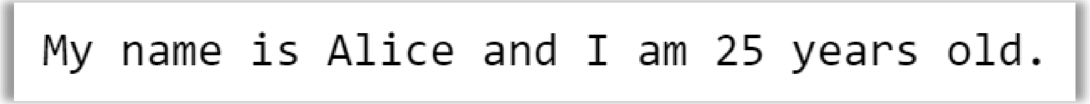
Sequential Types: the `str` type

String Interpolation: Python offers multiple ways to format strings and allows you to create dynamic strings without the need for excessive concatenation or complex formatting operations.

4. `f-strings` (formatted string literals): Introduced in Python 3.6, f-strings provide a concise and readable way to interpolate variables and expressions directly into strings. You can enclose expressions within curly braces {} and prefix the string with the letter f.

```python
name = "Alice"
age = 25
message = f"My name is {name} and I am {age} years old."
```

All the above outputs are the same:



```
My name is Alice and I am 25 years old.
```

4. Python Basics

4.5 Data Types

Sequential Types: the `str` type

String Methods: commonly used methods are listed below. More methods can be found in the online documentation: https://docs.python.org/3/library/stdtypes.html#string-methods

Method	**Description**	**Example**
`str.lower()`	Returns a lowercase version of the string.	`"Hello".lower()` returns `"hello".`
`str.upper()`	Returns an uppercase version of the string.	`"Hello".upper()` returns `"HELLO".`
`str.strip()`	Removes leading and trailing whitespace.	`" Hello ".strip()` returns `"Hello".`
`str.replace(old, new)`	Replaces occurrences of a substring with another.	`"Hello".replace("l", "L")` returns `"HeLLo".`
`str.split(separator)`	Splits the string into a list of substrings.	`"Hello World".split(" ")` returns `["Hello", "World"].`
`str.join(iterable)`	Joins the elements of an iterable into a string.	`" ".join(["Hello", "World"])` returns `"Hello World".`
`str.isdigit()`	Checks if all characters are digits.	`"123".isdigit()` returns `True.`
`str.isalpha()`	Checks if all characters are alphabetic.	`"Hello".isalpha()` returns `True.`
`str.islower()`	Checks if all characters are lowercase.	`"hello".islower()` returns `True.`
`str.isupper()`	Checks if all characters are uppercase.	`"HELLO".isupper()` returns `True.`
`str.startswith(prefix)`	Checks if the string starts with the given prefix.	`"Hello".startswith("He")` returns `True.`
`str.endswith(suffix)`	Checks if the string ends with the given suffix.	`"Hello".endswith("lo")` returns `True.`
`str.count(substring)`	Counts the number of occurrences of a substring.	`"Hello World".count("o")` returns `2.`

4. Python Basics

4.5 Data Types

Sequential Types: the `list` type

The `list` type in Python is an ordered collection of items that can hold various data types. Here are key points summarizing the `list` type:

- **Ordered Collection**: Lists maintain the order of elements, meaning the position of each item is preserved.

- **Heterogeneous Data**: Lists can hold items of different data types, such as integers, floats, strings, or even other lists.

- **Dynamic Size**: Lists can grow or shrink dynamically as items are added or removed. They have no fixed size limitations.

- **Mutable**: Lists are mutable, which means you can modify their elements by assigning new values, adding or removing items, or reordering the existing items. Lists provide methods like append(), extend(), insert(), remove(), pop(), and clear() to manipulate their contents.

4. Python Basics

4.5 Data Types

Sequential Types: the `list` type

Operators on the `list` type for in-place modification.
The following operators demonstrate how to modify list elements, replace sublists with new elements, and delete elements or sublists using the `del` statement. Note that the examples modify the list in-place, resulting in a modified list after each operation.

Operation	**Description**	**Example**

`list[index] = value`	Changes the value at the specified index in the list.	`my_list = [1, 2, 3]`
		`my_list[1] = 4  # Result: [1, 4, 3]`

`list[start:end] = iterable`	Replaces a sublist with the elements from the iterable.	`my_list = [1, 2, 3, 4]`
		`my_list[1:3] = ['a', 'b’]  # Result: [1, 'a', 'b', 4]`

`list[start:end:step] = iterable`	Replaces a sublist with the elements from the iterable using a step.	`my_list = [1, 2, 3, 4, 5]`
		`my_list[::2] = ['a', 'b', 'c’]  # Result: ['a', 2, 'b', 4, 'c']`

`del list[index]`	Deletes the item at the specified index in the list.	`my_list = [1, 2, 3]`
		`del my_list[1]  # Result: [1, 3]`

`del list[start:end]`	Deletes a sublist from the list within the specified range.	`my_list = [1, 2, 3, 4, 5]`
		`del my_list[1:4]  # Result: [1, 5]`

`del list[start:end:step]`	Deletes a sublist from the list using a step value.	`my_list = [1, 2, 3, 4, 5]`
		`del my_list[::2]  # Result: [2, 4]`

4. Python Basics

4.5 Data Types

Sequential Types: the `list` type

List Methods: commonly used methods are listed below. More methods can be found in the online documentation: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Method	Description	Example

`list.append(item)`	Appends an item to the end of the list.	`my_list = [1, 2, 3]`
		`my_list.append(4)  # Result: [1, 2, 3, 4]`

`list.extend(iterable)`	Extends the list by appending elements from the iterable.	`my_list = [1, 2, 3]`
		`my_list.extend([4, 5])  # Result: [1, 2, 3, 4, 5]`

`list.insert(index, item)`	Inserts an item at the specified index in the list.	`my_list = [1, 2, 3]`
		`my_list.insert(1, 'a’)  # Result: [1, 'a', 2, 3]`

`list.remove(item)`	Removes the first occurrence of the item from the list.	`my_list = [1, 2, 3, 2]`
		`my_list.remove(2) # Result: [1, 3, 2]`

`list.pop(index=-1)`	Removes and returns the item at the specified index, or the last item if no index is provided.	`my_list = [1, 2, 3]`
		`item = my_list.pop(1) # Result: my_list = [1, 3], item = 2`

`list.clear()`	Removes all items from the list.	`my_list = [1, 2, 3]`
		`my_list.clear()  # Result: []`

`list.sort(key=None, reverse=False)`	Sorts the items in the list in ascending order.	`my_list = [3, 1, 2]`
		`my_list.sort()  # Result: [1, 2, 3]`

`list.reverse()`	Reverses the order of items in the list.	`my_list = [1, 2, 3]`
		`my_list.reverse()  # Result: [3, 2, 1]`

4. Python Basics

4.5 Data Types

Sequential Types: the `tuple` type

The `tuple` type in Python is an ordered collection of items that can hold various data types. Here are key points summarizing the `tuple` type:

- **Ordered Collection**: Lists maintain the order of elements, meaning the position of each item is preserved.

- **Heterogeneous Data**: Lists can hold items of different data types, such as integers, floats, strings, or even other tuples.

- **Immutable**: meaning its elements cannot be modified after creation. Although tuples are immutable, they can contain mutable objects like lists.

4. Python Basics

4.5 Data Types

Sequential Types: the `tuple` type

The `tuple` type in Python is an ordered collection of items that can hold various data types. Here are key points summarizing the `tuple` type:

- **Ordered Collection**: Lists maintain the order of elements, meaning the position of each item is preserved.

- **Heterogeneous Data**: Lists can hold items of different data types, such as integers, floats, strings, or even other tuples.

- **Immutable**: meaning its elements cannot be modified after creation. Although tuples are immutable, they can contain mutable objects like lists.

Common Methods	Description	Example

`tuple.count(value)`	Returns the number of occurrences of a specified value in the tuple.	`my_tuple = (1, 2, 3, 2, 4, 2)`
		`count = my_tuple.count(2)`
		`# Result: count = 3`

`tuple.index(value, start=0, end=len(tuple))`	Returns the index of the first occurrence of a specified value in the tuple within the given range.	`my_tuple = (1, 2, 3, 2, 4, 2)`
		`index = my_tuple.index(4)`
		`# Result: index = 4`

4. Python Basics

4.5 Data Types

Sequential Types: Packing and Unpacking

Sequential data types in Python, such as lists, tuples, and strings, have the unique feature of packing and unpacking.

- **Packing** is the process of combining multiple values into a single sequential data object. For example, you can pack multiple values into a tuple by simply enclosing them within parentheses ( ) (optional for tuple) or a list by using square brackets [ ]. Packing allows you to group related values together into a single object.

- **Unpacking**, on the other hand, is the process of extracting values from a sequential data object into individual variables. It allows you to assign the elements of a tuple, list, or string to separate variables in a single statement. Unpacking is particularly useful when you want to conveniently access and work with the individual elements of a sequence.

- To unpack a sequential data object, you can assign its elements to variables using the assignment operator = and separate the variables with commas. The number of variables on the left side of the assignment must match the number of elements in the unpacked sequence.

4. Python Basics

4.5 Data Types

Sequential Types: Packing and Unpacking

Packing and unpacking examples for the `str`, `list`, and `tuple` types.

```
# List Packing and Unpacking
my_list = [1, 2, 3]  # List Packing
a, b, c = my_list  # List Unpacking
print(a, b, c)  # Output: 1 2 3

# Tuple Packing and Unpacking
my_tuple = 1, 2, 3  # Tuple Packing
a, b, c = my_tuple  # Tuple Unpacking
print(a, b, c)  # Output: 1 2 3

# String Packing and Unpacking
my_string = "hello"  # String Packing
a, b, c, d, e = my_string  # String Unpacking
print(a, b, c, d, e)  # Output: h e l l o
```

4. Python Basics

4.5 Data Types

Sequential Types: Packing and Unpacking

In unpacking, you can use the asterisk (*) to collect multiple elements into a single variable. The variable that collects the remaining elements always receives a list type reference.

```python
my_list = [1, 2, 3, 4, 5]
a, *b, c = my_list

print(a)  # Output: 1
print(b)  # Output: [2, 3, 4]
print(c)  # Output: 5

my_tuple = (1, 2, 3, 4, 5)
a, *b, c = my_tuple

print(a)  # Output: 1
print(b)  # Output: [2, 3, 4]
print(c)  # Output: 5

my_string = "hello"
a, *b, c = my_string

print(a)  # Output: 'h'
print(b)  # Output: ['e', 'l', 'l']
print(c)  # Output: 'o'
```

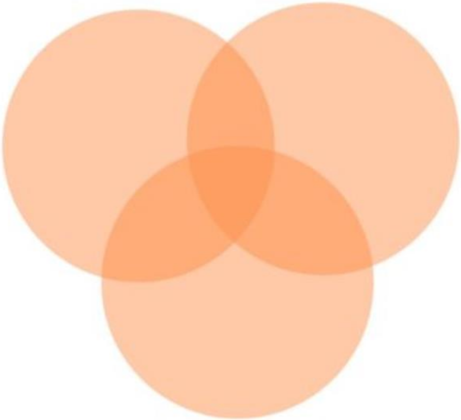
4. Python Basics

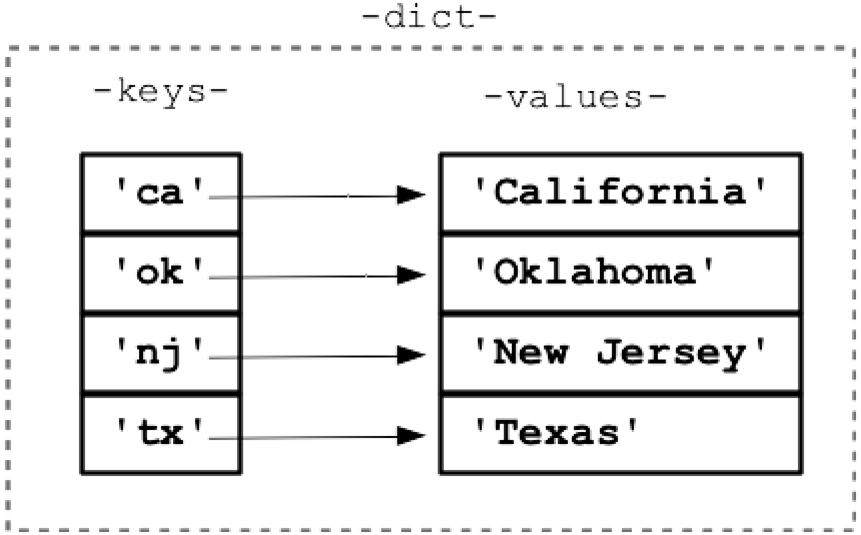
4.5 Data Types

Mapping and Set

Mapping data types (dictionaries) in Python represent key-value pairs, while set data types represent unordered collections of unique elements. Both are mutable, and their size and elements can be modified.

Set and Mapping Types			
`set`	Represents an unordered collection of unique elements	`unique_numbers = {1, 2, 3}`	Mutable
`dict`	Represents a collection of key-value pairs	`person = {'name': 'John', 'age': 25}`	Mutable





4. Python Basics

4.5 Data Types

Mapping and Set: the `set` type

- Set data type in Python represents a mutable unordered collection of unique elements.
- Sets are created by enclosing comma-separated values within curly braces (`{}`), or by using the `set()` constructor.
- Sets provide operations like union, intersection, difference, and membership testing.
- Sets are useful for tasks that involve finding unique values, testing membership, or performing set operations.

```
# Creating a Set using Curly Braces
my_set = {1, 2, 3, 4, 5}

# Creating a Set using the set() Constructor
my_set = set([1, 2, 3, 4, 5])

# Creating an Empty Set
my_set = set()

# Creating a Set from a String
my_set = set("hello")

# Creating a Set from a List
my_list = [1, 2, 3, 4, 5]
my_set = set(my_list)
```

4. Python Basics

4.5 Data Types

Mapping and Set: the `set` type

Some common set operators

Operator	**Description**	**Example**

`set1 | set2`	Union: Returns a set containing all elements from both sets.	`set1 = {1, 2, 3}`
		`set2 = {3, 4, 5}`
		`union_set = set1 & set2  # Result: {1, 2, 3, 4, 5}`

`set1 & set2`	Intersection: Returns a set containing common elements between two sets.	`set1 = {1, 2, 3}`
		`set2 = {3, 4, 5}`
		`intersection_set = set1 & set2  # Result: {3}`

`set1 - set2`	Difference: Returns a set containing elements present in the first set but not in the second set.	`set1 = {1, 2, 3}`
		`set2 = {3, 4, 5}`
		`difference_set = set1 - set2  # Result: {1, 2}`

`set1 ^ set2`	Symmetric Difference: Returns a set containing elements that are in either of the sets, but not both.	`set1 = {1, 2, 3}`
		`set2 = {3, 4, 5}`
		`symmetric_difference_set = set1 ^ set2  # Result: {1, 2, 4, 5}`

`set1 <= set2`	Subset: Returns True if all elements of the first set are present in the second set.	`set1 = {1, 2}`
		`set2 = {1, 2, 3}`
		`is_subset = set1 <= set2  # Result: True`

`set1 >= set2`	Superset: Returns True if all elements of the second set are present in the first set.	`set1 = {1, 2, 3}`
		`set2 = {1, 2}`
		`is_superset = set1 >= set2  # Result: True`

4. Python Basics

4.5 Data Types

Mapping and Set: the `set` type

Some common set methods. Refer to the documentation for details:
https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Method	**Description**
`set.add(element)`	Adds an element to the set.
`set.remove(element)`	Removes an element from the set.
`set.discard(element)`	Removes an element from the set if it exists, otherwise does nothing.
`set.pop()`	Removes and returns an arbitrary element from the set.
`set.clear()`	Removes all elements from the set, making it empty.
`set.copy()`	Returns a shallow copy of the set.
`set.update(iterable)`	Updates the set by adding elements from an iterable.
`set.intersection(*others)`	Returns a new set with elements common to the set and all other sets.
`set.difference(*others)`	Returns a new set with elements in the set but not in any of the other sets.
`set.symmetric_difference(other)`	Returns a new set with elements present in either the set or the other set, but not both.
`set.union(*others)`	Returns a new set with elements from the set and all other sets.

4. Python Basics

4.5 Data Types

Mapping and Set: the `dict` type

- `dict` data type in Python represents a mutable unordered collection of key-value pairs.
- Each key-value pair in a dictionary is separated by a colon (`:`) and enclosed in curly braces (`{}`).
- Keys within a dictionary must be unique, and they are typically immutable types such as strings or numbers.
- Values within a dictionary can be of any type and can be accessed and modified using their corresponding keys.
- Dictionaries provide fast lookups based on keys, making them efficient for data retrieval by key-value association.

```
# Creating an empty dictionary
my_dict = {}

# Creating a dictionary with key-value pairs
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Using the dict() constructor
my_dict = dict()

# Creating a dictionary from a list of tuples
my_dict = dict([('name', 'John'), ('age', 30), ('city', 'New York')])

# Creating a dictionary using keyword arguments
my_dict = dict(name='John', age=30, city='New York')
```

4. Python Basics

4.5 Data Types

Mapping and Set: the `dict` type

Iteration

1. Iterating over keys using `'for key in dict'`

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
for key in my_dict:
    print(key)
```

2. Iterating over values using `'for value in dict.values()'`

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
for value in my_dict.values():
    print(value)
```

3. Iterating over key-value pairs using `'for key, value in dict.items()'`

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
for key, value in my_dict.items():
    print(key, value)
```

4. Python Basics

4.5 Data Types

Mapping and Set: the `dict` type

Operators

Operator | **Description** | **Example**

`[]` | Accessing values by key: my_dict[key]

```
my_dict = {'name': 'John’}
value = my_dict['name’]  # Result: 'John'
```

`in` | Checking if a key exists: key in my_dict

```
my_dict = {'name': 'John'}
if 'name' in my_dict:
    print('Key exists')
```

`not in` | Checking if a key does not exist: key not in my_dict

```
my_dict = {'name': 'John’}
if 'age' not in my_dict:
    print('Key does not exist')
```

`==` | Equality comparison: dict1 == dict2

```
dict1 = {'name': 'John’}
dict2 = {'name': 'John’}
if dict1 == dict2:
    print('Dictionaries are equal')
```

`!=` | Inequality comparison: dict1 != dict2

```
dict1 = {'name': 'John’}
dict2 = {'name': 'Jane’}
if dict1 != dict2:
    print('Dictionaries are not equal')
```

`len()` | Getting the number of key-value pairs: len(my_dict)

```
my_dict = {'name': 'John', 'age': 30}
length = len(my_dict)  # Result: 2
```

`del` | Deleting a key-value pair: del my_dict[key]

```
my_dict = {'name': 'John'}
del my_dict['name'] # Result: dictionary is empty
```

4. Python Basics

4.5 Data Types

Mapping and Set: the `dict` type

Some common `dict` methods. Refer to the documentation for details:
https://docs.python.org/3/library/stdtypes.html#dict

Method	**Description**
`dict.get(key, default)`	Returns the value for a given key. If the key is not found, returns default.
`dict.keys()`	Returns a view object containing all the keys in the dictionary.
`dict.values()`	Returns a view object containing all the values in the dictionary.
`dict.items()`	Returns a view object containing all the key-value pairs in the dictionary.
`dict.update(other_dict)`	Updates the dictionary with the key-value pairs from another dictionary.
`dict.pop(key, default)`	Removes and returns the value for a given key. If the key is not found, returns default.
`dict.popitem()`	Removes and returns an arbitrary key-value pair from the dictionary.
`dict.clear()`	Removes all key-value pairs from the dictionary, making it empty.
`dict.copy()`	Returns a shallow copy of the dictionary.

4. Python Basics

4.5 Data Types

Sequence Comprehension

Python comprehensions create new sequences from existing ones. There are three types of comprehensions in Python:

1. **List Comprehensions**: List Comprehensions provide a way to create new lists from existing iterables (e.g., a list, tuple, or string) and applying an expression to each element. The result is a new list derived from the original elements. List comprehensions follow the syntax: `[expression for item in iterable if condition]`.

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers if n % 2 == 0] # a new list: [4, 16]
```

2. **Set Comprehensions**: Set comprehensions are similar to list comprehensions but produce sets as output.. Set comprehensions follow the syntax: `{expression for item in iterable if condition}`.

```
numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1]
unique_squares = {n**2 for n in numbers} # a new set: {1, 4, 9, 16, 25}
```

3. **Dictionary Comprehensions:** Dictionary comprehensions allow you to create new dictionaries by iterating over an iterable and constructing key-value pairs based on an expression. Dictionary comprehensions follow the syntax: `{key_expression: value_expression for item in iterable if condition}.`

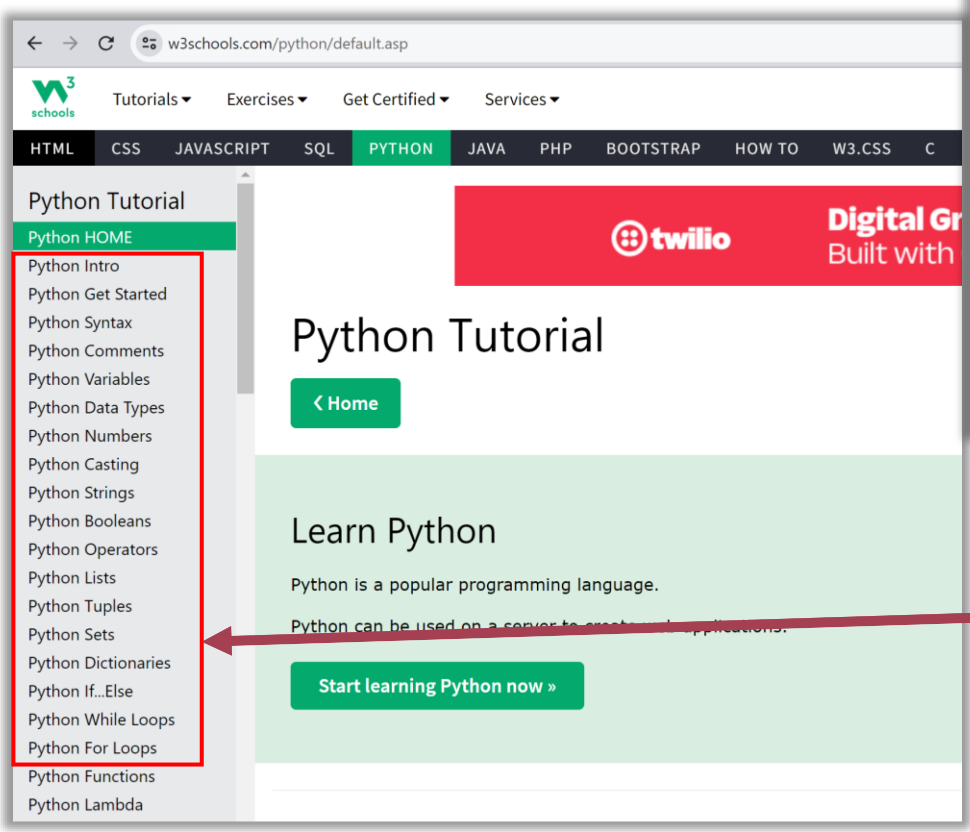
```
fruits = ['apple', 'banana', 'orange']
fruit_lengths = {fruit: len(fruit) for fruit in fruits} # a new dict: {'apple': 5, 'banana': 6, 'orange': 6}
```

4. Python Basics

Coding exercises

W3schools offers interactive quizzes and coding exercises to help you learn Python. Please pause the video and open the following page. Try all you have learned so far.

https://www.w3schools.com/python/default.asp



Python Tutorial

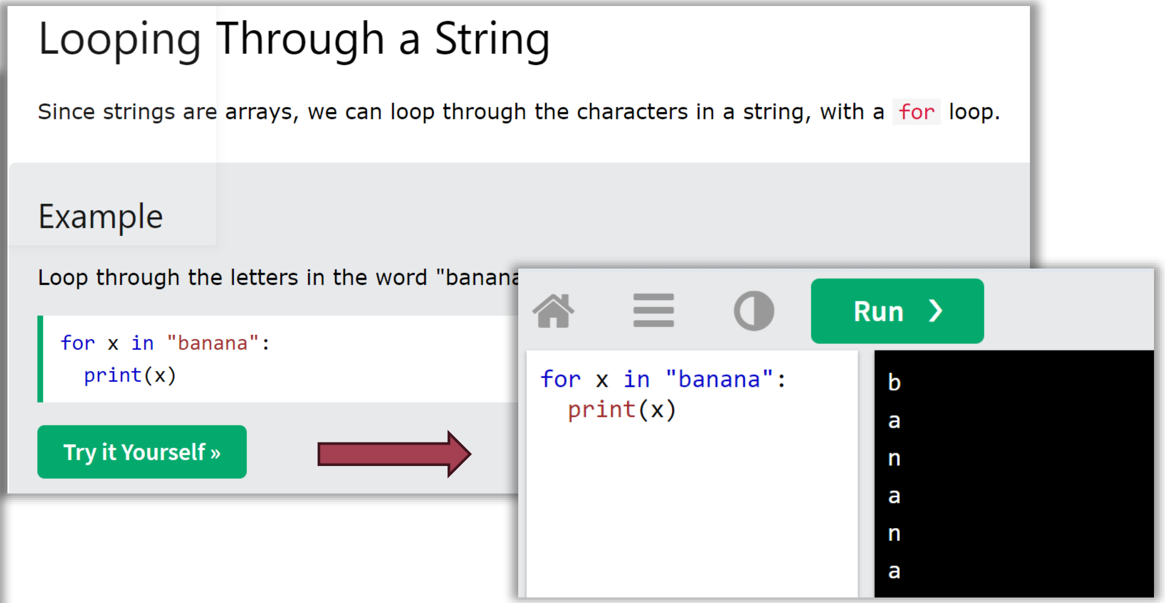
- Python HOME
- Python Intro
- Python Get Started
- Python Syntax
- Python Comments
- Python Variables
- Python Data Types
- Python Numbers
- Python Casting
- Python Strings
- Python Booleans
- Python Operators
- Python Lists
- Python Tuples
- Python Sets
- Python Dictionaries
- Python If...Else
- Python While Loops
- Python For Loops
- Python Functions
- Python Lambda

Python Tutorial

Learn Python

Python is a popular programming language.

Python can be used on a server to create web applications.



Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

Loop through the letters in the word "banana

```
for x in "banana":
  print(x)
```

```
for x in "banana":
  print(x)
```

```
b
a
n
a
n
a
```

Try all these sections

4. Python Basics

4.6 Functions

Functions in Python are reusable blocks of code that perform specific tasks. They help organize code, improve readability, and promote code reuse.

Defining a Function

```
def function_name(parameter1, parameter2, ...):
    """
    Docstring: Description of the function.
    Optional documentation providing more details about the function.
    """

    # Function code block
    # Perform operations and calculations here

    # Optional: Return a value
    return result
```

- **function_name** is the name you choose for your function.
- **parameter1, parameter2, ...** are optional input parameters that your function may accept. You can define as many parameters as needed, separating them with commas.
- **A docstring in triple quotes `"""`… `"""`** provides information on a function’s purpose, inputs, return value, and more. (optional)
- The **function code block** is indented beneath the function definition. This is where you write the actual code to execute when the function is called.
- If your function is designed to return a value, you can use the **return statement** followed by the value you want to return. This is optional, and not all functions need to have a return statement.

4. Python Basics

4.6 Functions

Defining a local (nested) Function within another function

```
def outer_function():
    # Define the local function
    def inner_function():
        # Function logic goes here
        pass

    # Call the local function
    inner_function()

# Call the outer function
outer_function()
```

Remember that the local function is only accessible within the scope of the outer function. It cannot be directly accessed or called from outside the outer function.

4. Python Basics

4.6 Functions

Function Example

```
def square(number):
    """
    Calculates the square of a given number.

    Parameters:
    number (int or float): The number to be squared.

    Returns:
    int or float: The square of the input number.
    """
    result = number ** 2
    return result

# Calling the square function
result = square(5)
print(result)
```

In Python, you need to define a function before you can call it. This is because Python is an interpreted language, which means it executes code line by line. When you define a function, you are creating a block of code that can be executed later when you call it.

If you try to call a function before it is defined, you will encounter a NameError because Python hasn't encountered the function definition yet.

4. Python Basics

4.6 Functions

Function Parameters

Functions can have different types of parameters that allow you to pass information into the function.

1. **Positional Parameters:** Parameters defined by their position in the function call. The values passed as arguments match the parameters based on their position.

```
def function_name(param1, param2, ...):
    # Function code
```

2. **Default Parameters**: Default parameters have predefined values assigned to them. The default value is used if no argument is provided for a default parameter.

```
def function_name(param1=default_value1, param2=default_value2, ...):
    # Function code
```

```
def greet(name, greeting="Hello"):
    """
    Greets a person with a specified greeting (default: Hello).

    Parameters:
    name (str): The name of the person.
    greeting (str, optional): The greeting to use (default: Hello).
    """
    print(greeting + ", " + name + "!")

# Calling the greet function with default greeting
greet("Alice")  # Output: Hello, Alice!

# Calling the greet function with a custom greeting
greet("Bob", "Hi")  # Output: Hi, Bob!
```

4. Python Basics

4.6 Functions

Function Parameters

3. **Variable-Length Positional Parameters** (args**):** Variable-length positional parameters, commonly known as *args, allow you to pass an arbitrary number of positional arguments to a function. The arguments are collected into a tuple within the function.

```
def function_name(*args):
    # Function code
```

```
def calculate_sum(*numbers):
    """
    Calculates the sum of an arbitrary number of numbers.

    Parameters:
    *numbers (float): Variable-length positional parameters representing numbers.

    Returns:
    float: The sum of the input numbers.
    """
    total = sum(numbers)
    return total

# Calling the calculate_sum function with different numbers
result1 = calculate_sum(2, 4, 6)  # Output: 12
result2 = calculate_sum(1, 3, 5, 7, 9)  # Output: 25

print(result1)
print(result2)
```

4. Python Basics

4.6 Functions

Function Parameters

4. **Keyword Parameters**: Keyword parameters, also known as named parameters, allow you to pass arguments using their parameter names. This provides clarity and flexibility in function calls.

```
def function_name(param1=value1, param2=value2, ...):
    # Function code
```

```
def create_person(name, age, occupation=None, city=None):
    """
    Creates a person with a given name, age, occupation, and city.

    Parameters:
    name (str): The name of the person.
    age (int): The age of the person.
    occupation (str, optional): The occupation of the person. Default is None.
    city (str, optional): The city of the person. Default is None.

    Returns:
    dict: A dictionary containing the person's information.
    """
    person = {
        'name': name,
        'age': age,
        'occupation': occupation,
        'city': city
    }
    return person

# Creating a person with required parameters only
person1 = create_person('Alice', 25)
print(person1)
# Output: {'name': 'Alice', 'age': 25, 'occupation': None, 'city': None}

# Creating a person with additional information
person2 = create_person('Bob', 30, occupation='Engineer', city='New York')
print(person2)
# Output: {'name': 'Bob', 'age': 30, 'occupation': 'Engineer', 'city': 'New York'}
```

4. Python Basics

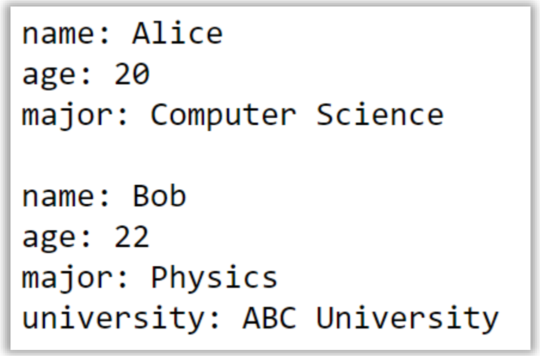
4.6 Functions

Function Parameters

5. **Variable-Length Keyword Parameters (kwargs):** Variable-length keyword parameters, commonly known as **kwargs, allow you to pass an arbitrary number of keyword arguments to a function. The arguments are collected into a dictionary object within the function.

```
def function_name(**kwargs):
    # Function code
```

```
def print_student_details(**kwargs):
    """
    Prints the details of a student.

    Parameters:
    **kwargs (dict): Variable-length keyword parameters representing student details.
    """
    for key, value in kwargs.items():
        # Print each key-value pair
        print(key + ':', value)

# Calling the function with different student details
print_student_details(name='Alice', age=20, major='Computer Science’)
print("")
print_student_details(name='Bob', age=22, major='Physics', university='ABC University')
```



```
name: Alice
age: 20
major: Computer Science

name: Bob
age: 22
major: Physics
university: ABC University
```

4. Python Basics

4.6 Functions

Function Returns

Return Multiple Values

In Python, you can return multiple values from a function by using a tuple, a list, or multiple variables separated by commas

```
# Returning as Tuple
def get_values_tuple():
    value1 = 10
    value2 = "Hello"
    return (value1, value2)

result_tuple = get_values_tuple()
print(result_tuple)  # Output: (10, "Hello")

# Returning as List
def get_values_list():
    value1 = 10
    value2 = "Hello"
    return [value1, value2]

result_list = get_values_list()
print(result_list)  # Output: [10, "Hello"]

# Returning as Multiple Variables
def get_values_multiple():
    value1 = 10
    value2 = "Hello"
    return value1, value2

result1, result2 = get_values_multiple()
print(result1)  # Output: 10
print(result2)  # Output: "Hello"
```

The values are implicitly packed into a tuple

Tuple unpacking

4. Python Basics

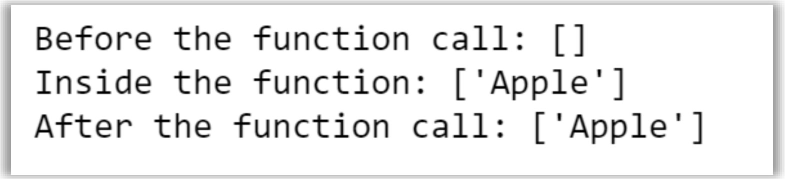
4.6 Functions

“Calling by Reference” Side Effects

In Python, when you pass an argument to a function, it is passed by reference. If the argument is mutable, any modifications made to it within the function persist outside the function scope, affecting the original object.

Be aware of these effects of altering mutable objects inside a function.

```python
def add_item_to_list(item, my_list):
    """
    Adds an item to the provided list.

    Parameters:
    item: The item to be added.
    my_list: The list to which the item is added.
    """
    my_list.append(item)
    print("Inside the function:", my_list)

# Creating an empty list
my_list = []

print("Before the function call:", my_list)

# Calling the function and modifying the list
add_item_to_list("Apple", my_list)

print("After the function call:", my_list)
```



```
Before the function call: []
Inside the function: ['Apple']
After the function call: ['Apple']
```

4. Python Basics

4.6 Functions

Namespace and Scope

In Python, a namespace is a system that determines the scope and accessibility of names (variables, functions, classes, etc.) within a program.

- Each function in Python has its own namespace, which is a local scope where the function's variables and other names are stored.
- Any names defined within the function are inaccessible outside of the function.
- The function can access names from its own namespace and names from any outer scopes (such as global scope or enclosing functions) where it was defined. If a name is not found in the local namespace, Python will search the enclosing scopes and then the global scope until the name is found or an error occurs.

```python
# Global variable
global_var = "Global"

def outer_function():
    # Outer function's local variable
    outer_var = "Outer"

    def inner_function():
        # Inner function's local variable
        inner_var = "Inner"

        # Accessing variables from different scopes
        print("Inner:", inner_var)
        print("Outer:", outer_var)
        print("Global:", global_var)

    # Calling the inner function
    inner_function()

# Calling the outer function
outer_function()

# Attempting to access variables from outer scopes
# Uncommenting any of the lines below will result in
# a NameError.

# print("Outer (from global):", outer_var)
# print("Inner (from global):", inner_var)
```

4. Python Basics

4.6 Functions

Built-in Functions

Python provides a rich set of built-in functions that are readily available for use without requiring explicit import statements. Here is a summary of some commonly used Python built-in functions:

Function	**Description**
`print()`	Outputs text or values to the console.
`len()`	Returns the length of an object, such as a string, list, or tuple.
`type()`	Returns the type of an object.
`int(), float(), str(), bool()`	Converts values to integer, float, string, or boolean types, respectively.
`input()`	Reads input from the user via the console.
`range()`	Generates a sequence of numbers within a specified range.
`abs()`	Returns the absolute value of a number.
`max(), min()`	Returns the maximum or minimum value from a sequence of values.
`sum()`	Calculates the sum of values in a sequence.
`round()`	Rounds a number to a specified precision.
`sorted()`	Returns a new sorted list from an iterable.
`enumerate()`	Returns an iterator of tuples with indices and corresponding values from an iterable.
`zip()`	Combines multiple iterables into a single iterator of tuples.
`any(), all()`	Returns True if any or all elements in an iterable are true, respectively.
`map()`	Applies a function to each element of an iterable and returns an iterator with the results.
`filter()`	Filters elements from an iterable based on a specified condition and returns an iterator

For complete reference: https://docs.python.org/3.11/library/functions.html

4. Python Basics

4.6 Functions

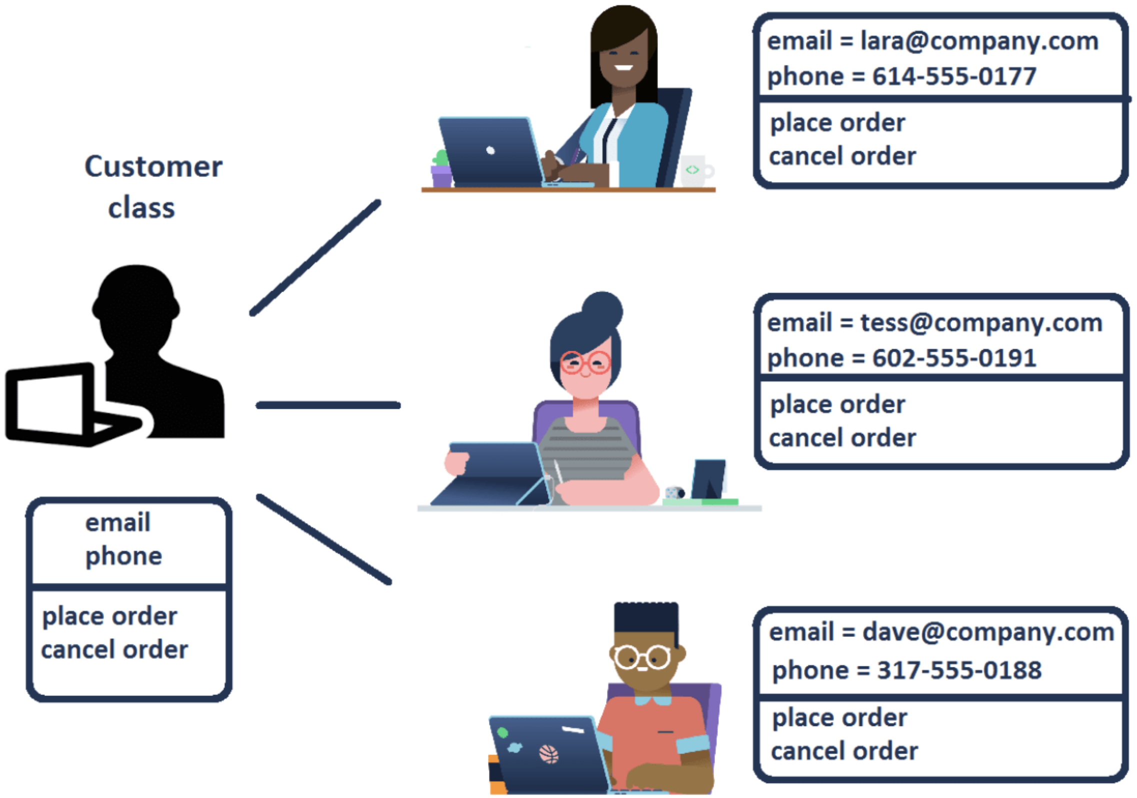
Do some coding exercises here:

https://www.w3schools.com/python/python_functions.asp
https://www.w3schools.com/python/python_scope.asp

4. Python Basics

4.7 Classes, Objects, and Methods

- **Class**: A blueprint/template that defines the structure and behavior of objects. It encapsulates related data and functionality into a single entity.
- **Object**: A specific instance created from a class, representing an individual entity with its own data and behaviors.
- **Method**: A function defined within a class that operates on objects, defining their behavior and allowing them to perform actions.



Customer class

email
phone

place order
cancel order

email = lara@company.com
phone = 614-555-0177

place order
cancel order

email = tess@company.com
phone = 602-555-0191

place order
cancel order

email = dave@company.com
phone = 317-555-0188

place order
cancel order

4. Python Basics

4.7 Classes, Objects, and Methods

Let's use the `str` class (or type) as an example to explain the concepts

- Class: The `str` class in Python is a built-in class that represents strings. It defines the structure and behavior of string objects.
- Object: An object of the `str` class is an instance of the class representing a specific string value. For example, "`Hello, World!`" is an object of the `str` class, representing a string with that specific content.
- Method: The `str` class provides various methods that can be applied to string objects. For instance, the upper() method is a method defined within the `str` class. We can use it on string objects to convert the string to the uppercase.

```
my_string = "Hello, World!"

# The `str` class represents the structure and behavior of string objects.
# `my_string` is an object of the `str` class, representing the specific string
"Hello, World!".

uppercase_string = my_string.upper()
# `upper()` is a method defined within the `str` class.
# It is called on the `my_string` object to convert the string to uppercase.

print(uppercase_string)  # Output: "HELLO, WORLD!"
```

Python can access methods from objects using dot notation:

- `object_name.object_method()`

4. Python Basics

4.8 Packages and Modules

In Python, packages and modules are organizational structures that help manage and organize code.

A module is a single file containing Python definitions, functions, classes, or variables. It is a way to organize code into reusable components.

- You can import a module into other Python scripts or modules using the `import` statement. After that, you can use dot notation to access code in a module, such as `module_name.function_name()` or `module_name.variable_name`.
- For example, if you have a module called `my_module.py` with a function named `my_function`, you can import and use it as follows:

```
import my_module

my_module.my_function()
```

my_module.py

4. Python Basics

4.8 Packages and Modules

In Python, there are primarily three types of modules:

1. **Built-in Modules**: These modules are part of the Python Standard Library and come pre-installed with Python. They offer various functionalities, including file handling, math operations, network communication, and data manipulation. Examples of built-in modules include `math`, `os`, `datetime`, and `random`.

2. **Third-Party Modules:** Not part of the Python Standard Library, these modules are created by outside developers. They can be installed separately using package managers like pip. Third-party modules offer additional functionality beyond what is available in the built-in modules and can be used to extend the capabilities of Python. Popular third-party modules include `NumPy`, `pandas`, `requests`, and `matplotlib`.

3. **User-Defined Modules**: These modules are created by users to encapsulate reusable code. User-defined modules are typically Python scripts with .py extensions that contain functions, classes, or variables that can be imported and used in other Python programs. By organizing related code into modules, you can achieve better code organization and reusability.

4. Python Basics

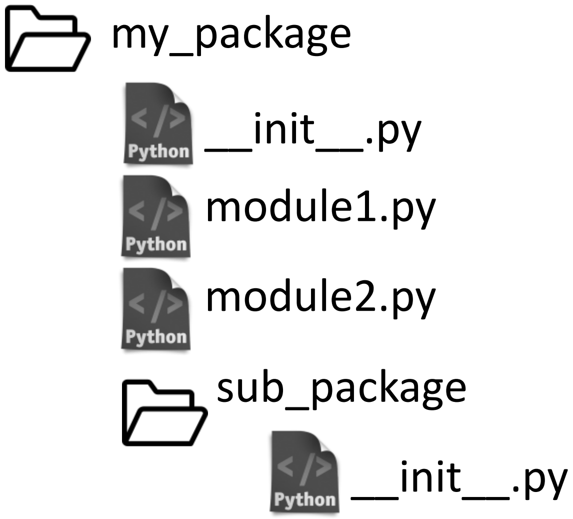
4.8 Packages and Modules

In Python, packages and modules are organizational structures that help manage and organize code.

A package is a way to organize related modules into a directory hierarchy. It can contain multiple modules and even nested sub-packages. Packages allow for a more structured code organization, especially for larger projects.

- A package is simply a directory that contains an `__init__.py` file, which marks the directory as a Python package. This file can be empty or can include the initialization code for the package. The package directory can also contain other modules or sub-packages.
- Packages are imported similarly to modules, using the import statement. When importing a package, you can access its modules and sub-packages using dot notation.
- For example, if you have a package called `my_package` with modules `module1` and `module2` inside it, you can import and use them as follows:

```
import my_package.module1
import my_package.module2

my_package.module1.my_function()
my_package.module2.my_function()
```



- my_package
  - __init__.py
  - module1.py
  - module2.py
  - sub_package
    - __init__.py

4. Python Basics

4.8 Packages and Modules

Possible import statements

```
# Importing the Entire Module
import math  # Imports the entire math module

# Importing Specific Items
from math import sqrt, pi  # Imports only sqrt() and pi from math

# Importing with an Alias
import pandas as pd  # Imports pandas module with an alias pd

# Importing All Items from a Module (Discouraged)
from math import *  # Imports all items from math (not recommended)

# Importing a Module from a Package
from my_package import my_module  # Imports my_module from my_package
```

4. Python Basics

4.8 Packages and Modules

Best Practices

```
# Use Explicit Imports
import math  # Good: Importing the entire module

from math import sqrt  # Good: Importing specific items

# Avoid using import *
# from math import *  # Avoid: Imports all items from math (not recommended)

# Import with an Alias
import pandas as pd  # Good: Importing with an alias

# Import Each Module on a Separate Line
import os
import sys

import numpy as np

import my_module

# Avoid importing multiple modules on a single line
# import os, sys  # Avoid: Importing multiple modules on a single line
```

4. Python Basics

4.8 Packages and Modules

Best Practices (continued)

```
# Group Imports Appropriately
# Standard library imports
# Third-party library imports
# Local module imports

# Standard library imports
import os
import sys
import datetime

# Third-party library imports
import numpy as np
import pandas as pd
import requests

# Local module imports
import my_module
from my_package import my_other_module
```

4. Python Basics

4.8 Packages and Modules

Coding Exercises
https://www.w3schools.com/python/python_modules.asp

5. Creating Your First Python Program

Apply what you have learned so far

5. Creating Your First Python Program

5.1 Getting familiar with the Spyder IDE

Jupyter Notebook excels in interactive data exploration and analysis, but only has limited functionality to debug your code if you work in a larger project.

Spyder offers a complete IDE with a code editor, variable explorer, debugger, profiler, and file explorer. It offers a comprehensive development environment focusing on code editing, project management, and debugging capabilities. This makes it well-suited for larger projects and more traditional software development workflows.

- Spyder's variable explorer provides an interactive way to inspect and explore variables in your code. It allows you to view, modify, and track the values of variables during execution, making it easier to debug and understand your code's behavior.
- Spyder offers a feature-rich code editor with advanced code editing capabilities. It includes features like syntax highlighting, code folding, code navigation, and integrated code documentation. Spyder also provides autocompletion, which can help speed up coding by suggesting possible completions as you type.

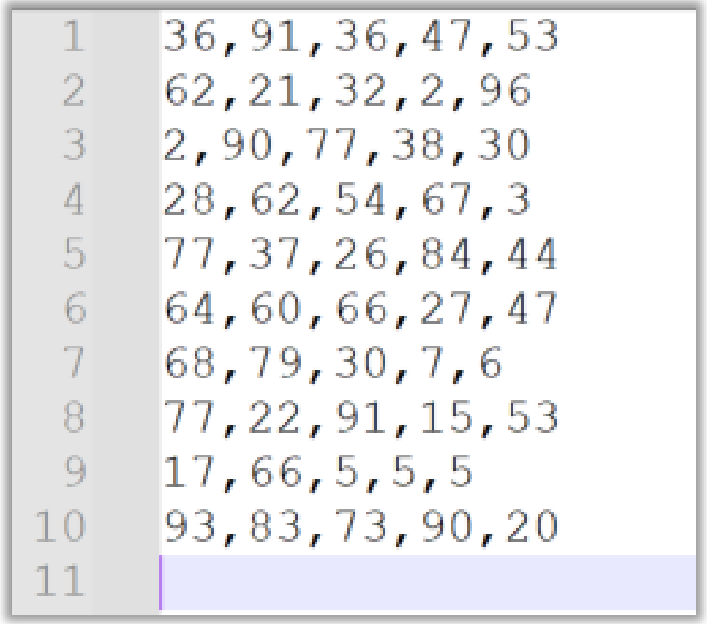
Watch the following video to learn how to use the Spyder IDE:
Introduction to the Spyder IDE for Python by Kyle Bradbury (15 mins)
Covering the IPython shell, text editor, variable editor, and code execution and debugging tools.

5. Creating Your First Python Program

5.2 Solve a Real Problem

Task: Find the maximum number from a CSV file

Requirement:
We have a CSV (Comma-Separated Values) file (data.csv) which contains multiple rows of integer numbers. Write a Python script to find the maximum integer from the file and then output it to the screen.



```
36,91,36,47,53
62,21,32,2,96
2,90,77,38,30
28,62,54,67,3
77,37,26,84,44
64,60,66,27,47
68,79,30,7,6
77,22,91,15,53
17,66,5,5,5
93,83,73,90,20

```

5. Creating Your First Python Program

5.2 Solve a Real Problem

General problem-solving process:

1. Separate your complex problem into multiple manageable components.
2. For each component, set up the requirements, research possible solutions, develop, and test the code.
3. Combine your components together and test them again with the original requirements.

For this specific task, we need to separate it into the following components:

1. Read the CSV file to access its content
2. For each row, find the maximum integer number, and store it somewhere.
3. After we complete iterating all the rows, we should get a collection of maximum numbers for each row.
4. Find the maximum number from the above collection and print it out on the screen.

5. Creating Your First Python Program

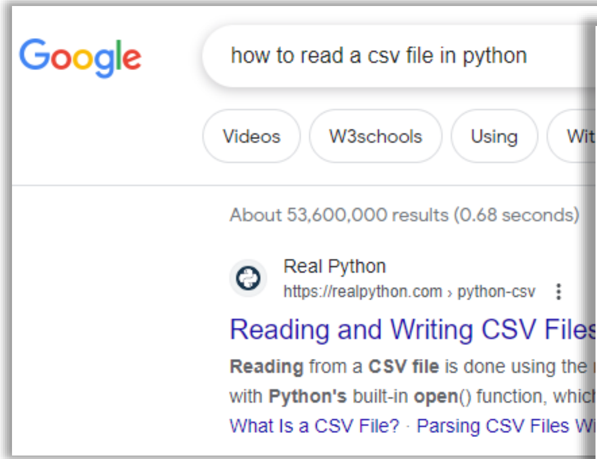
5.2 Solve a Real Problem

1. Read the CSV file to access its content

The functionalities of CSV file manipulation are part of the Python Standard library. You may have never used it before. However, you may seek help from

A. Python Standard library official documentation: https://docs.python.org/3/library/csv.html

B. Google



how to read a csv file in python

About 53,600,000 results (0.68 seconds)

Real Python
https://realpython.com › python-csv

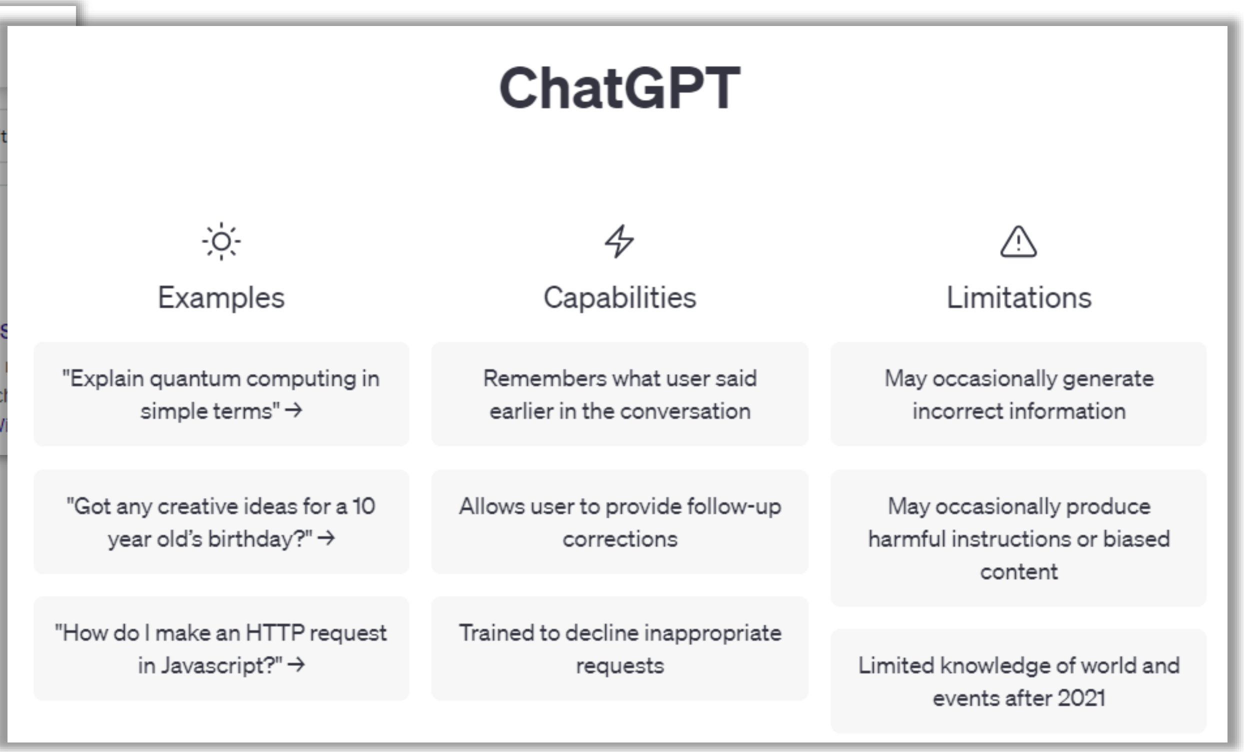
Reading and Writing CSV Files

Reading from a **CSV file** is done using the
with **Python's** built-in **open**() function, which

What Is a CSV File? · Parsing CSV Files Wi

C. ChatGPT: https://chat.openai.com/

Your best friend to solve problems quicker!



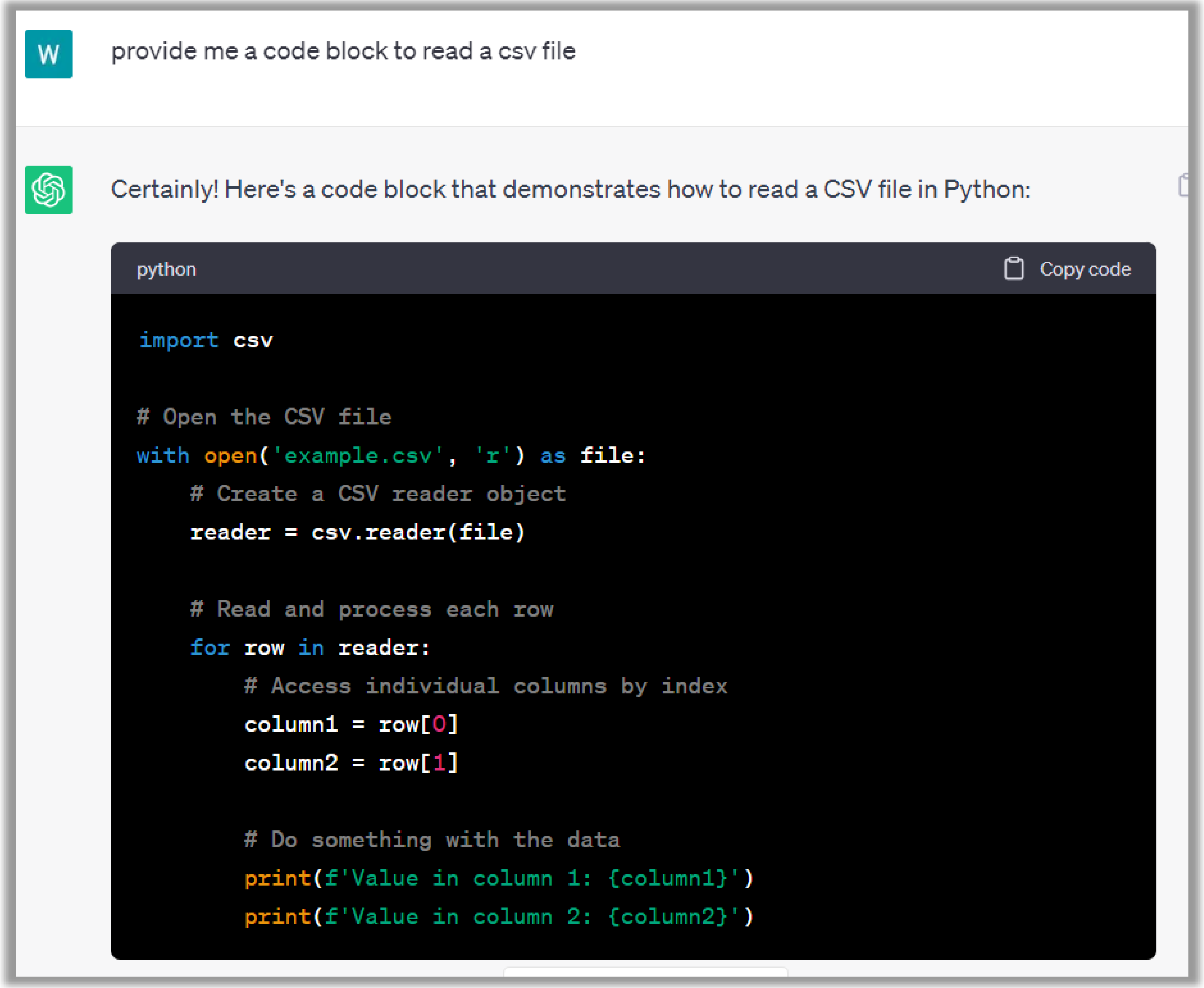
Examples	Capabilities	Limitations
"Explain quantum computing in simple terms" →	Remembers what user said earlier in the conversation	May occasionally generate incorrect information
"Got any creative ideas for a 10 year old's birthday?" →	Allows user to provide follow-up corrections	May occasionally produce harmful instructions or biased content
"How do I make an HTTP request in Javascript?" →	Trained to decline inappropriate requests	Limited knowledge of world and events after 2021

5. Creating Your First Python Program

5.2 Solve a Real Problem

1. Read the CSV file to access its content

1) Raise a question to ChatGPT and check its answer. Carefully read the explanations following the code. If you don't understand some statements, ask ChatGPT for clarification (and/or read Python documentation) until you fully understand them.
2) Copy and paste the code to a newly created Python file in the Spyder IDE. Save the file to a local folder.
3) Put the supplied "data.csv" file in the same folder.
4) Change the corresponding CSV file name in the Python file to "data.csv" and save it.



provide me a code block to read a csv file

Certainly! Here's a code block that demonstrates how to read a CSV file in Python:

```python
import csv

# Open the CSV file
with open('example.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Read and process each row
    for row in reader:
        # Access individual columns by index
        column1 = row[0]
        column2 = row[1]

        # Do something with the data
        print(f'Value in column 1: {column1}')
        print(f'Value in column 2: {column2}')
```

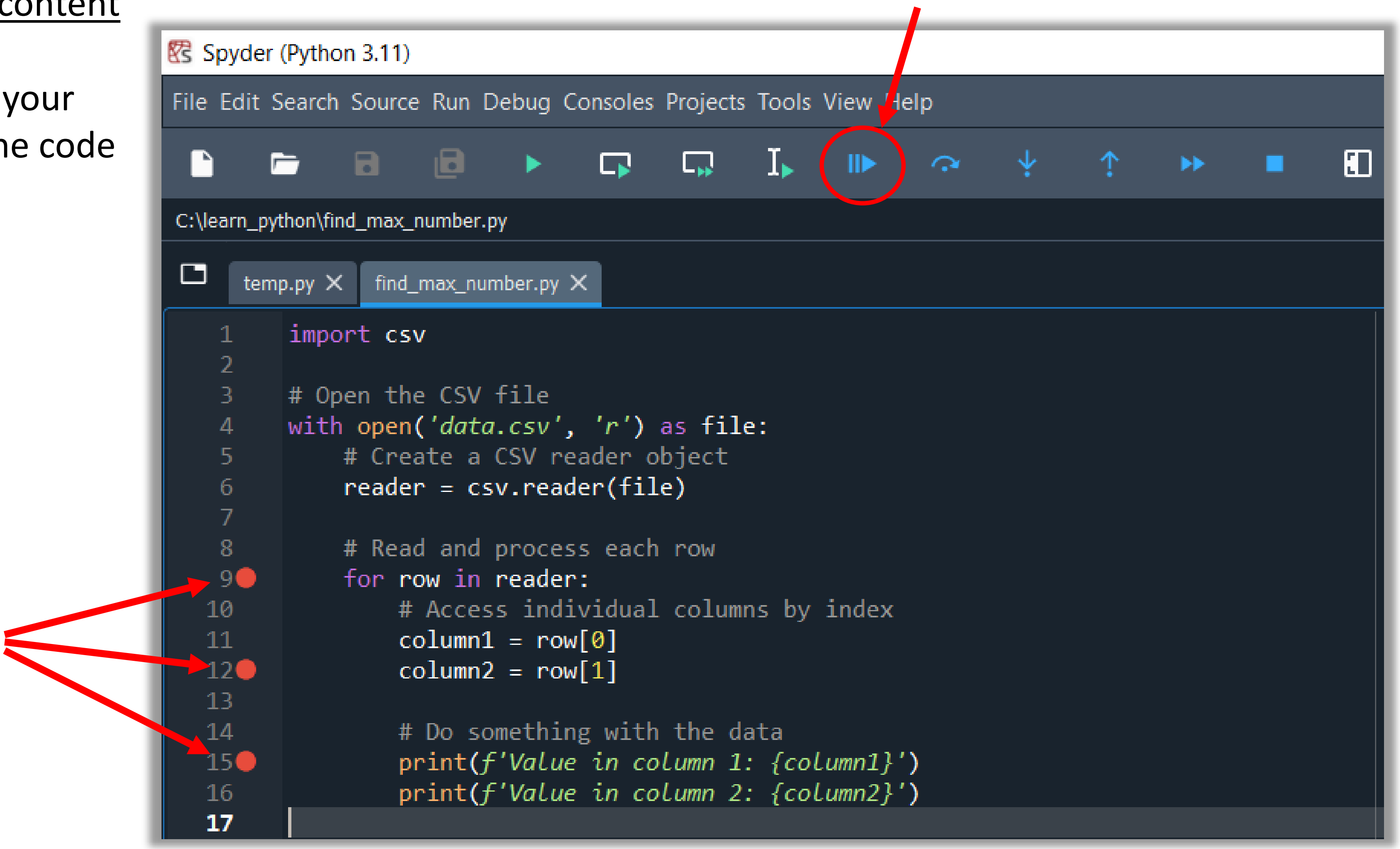
5. Creating Your First Python Program

5.2 Solve a Real Problem

1. <u>Read the CSV file to access its content</u>

5) Set a couple of breakpoints of your interest and start debugging the code by pressing the debug button

Debug button


Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\learn_python\find_max_number.py
temp.py
find_max_number.py

```
import csv

# Open the CSV file
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Read and process each row
    for row in reader:
        # Access individual columns by index
        column1 = row[0]
        column2 = row[1]

        # Do something with the data
        print(f'Value in column 1: {column1}')
        print(f'Value in column 2: {column2}')
```

breakpoints

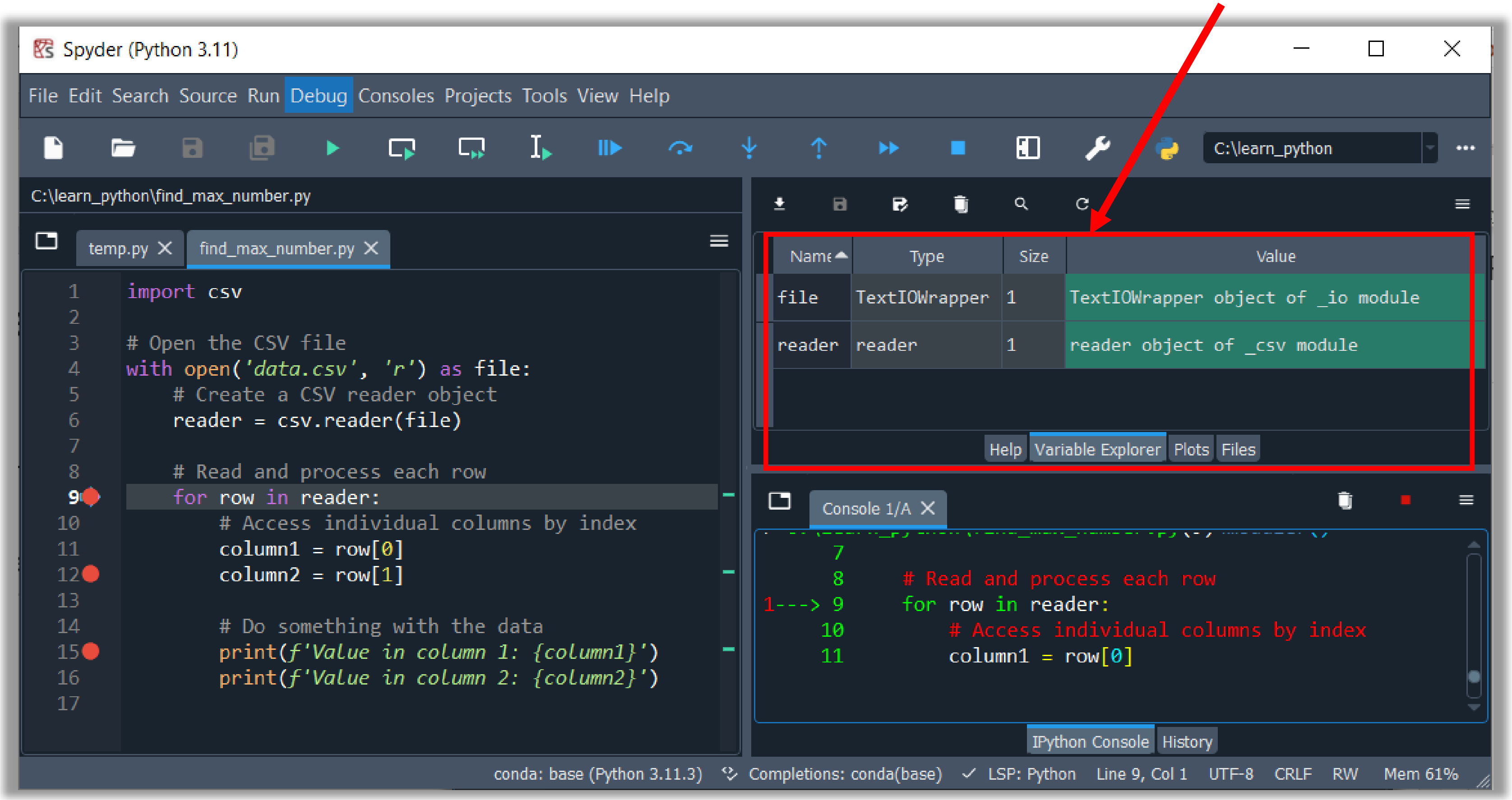
5. Creating Your First Python Program

5.2 Solve a Real Problem

1. <u>Read the CSV file to access its content</u>

6) Check the *Variable Explorer* for the active variables.
   Examine their Name, Type, Size, and Value

Variable Explorer



Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\learn_python
C:\learn_python\find_max_number.py
temp.py
find_max_number.py
Help Variable Explorer Plots Files
Console 1/A
IPython Console History
conda: base (Python 3.11.3) Completions: conda(base) LSP: Python Line 9, Col 1 UTF-8 CRLF RW Mem 61%

```python
import csv

# Open the CSV file
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Read and process each row
    for row in reader:
        # Access individual columns by index
        column1 = row[0]
        column2 = row[1]

        # Do something with the data
        print(f'Value in column 1: {column1}')
        print(f'Value in column 2: {column2}')
```

Name	Type	Size	Value
file	TextIOWrapper	1	TextIOWrapper object of _io module
reader	reader	1	reader object of _csv module

```
      7
      8     # Read and process each row
1---> 9     for row in reader:
     10         # Access individual columns by index
     11         column1 = row[0]
```

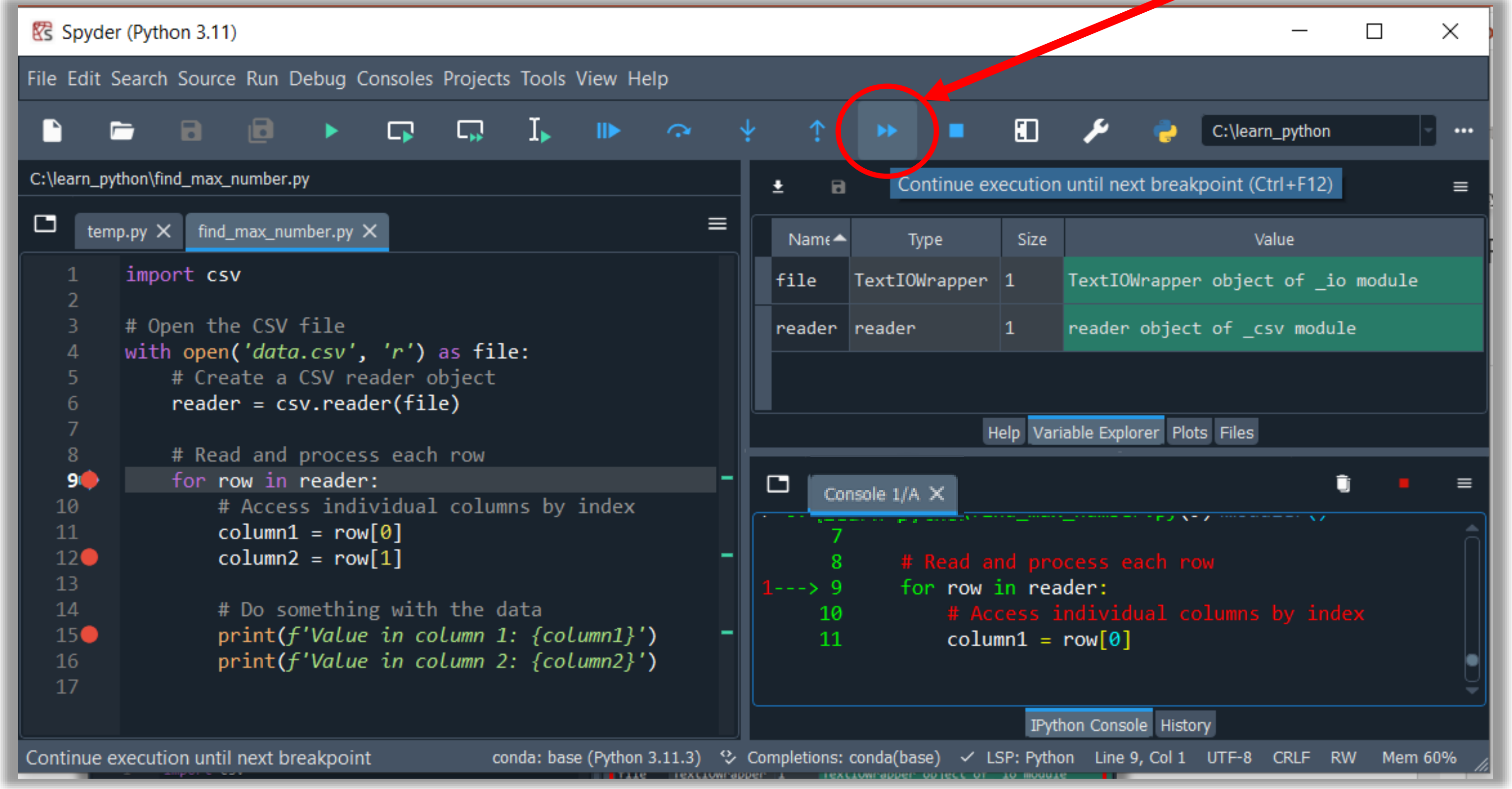
5. Creating Your First Python Program

5.2 Solve a Real Problem

1. <u>Read the CSV file to access its content</u>

7) Go to the next breakpoint until you can see the row variable is active and appears in the *Variable Explorer* pane

Click to go to the next breakpoint


Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\learn_python
Continue execution until next breakpoint (Ctrl+F12)
C:\learn_python\find_max_number.py
temp.py
find_max_number.py
Help
Variable Explorer
Plots
Files
Console 1/A
IPython Console
History
Continue execution until next breakpoint
conda: base (Python 3.11.3)
Completions: conda(base)
LSP: Python
Line 9, Col 1
UTF-8
CRLF
RW
Mem 60%

```python
import csv

# Open the CSV file
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Read and process each row
    for row in reader:
        # Access individual columns by index
        column1 = row[0]
        column2 = row[1]

        # Do something with the data
        print(f'Value in column 1: {column1}')
        print(f'Value in column 2: {column2}')
```

Name	Type	Size	Value
file	TextIOWrapper	1	TextIOWrapper object of _io module
reader	reader	1	reader object of _csv module

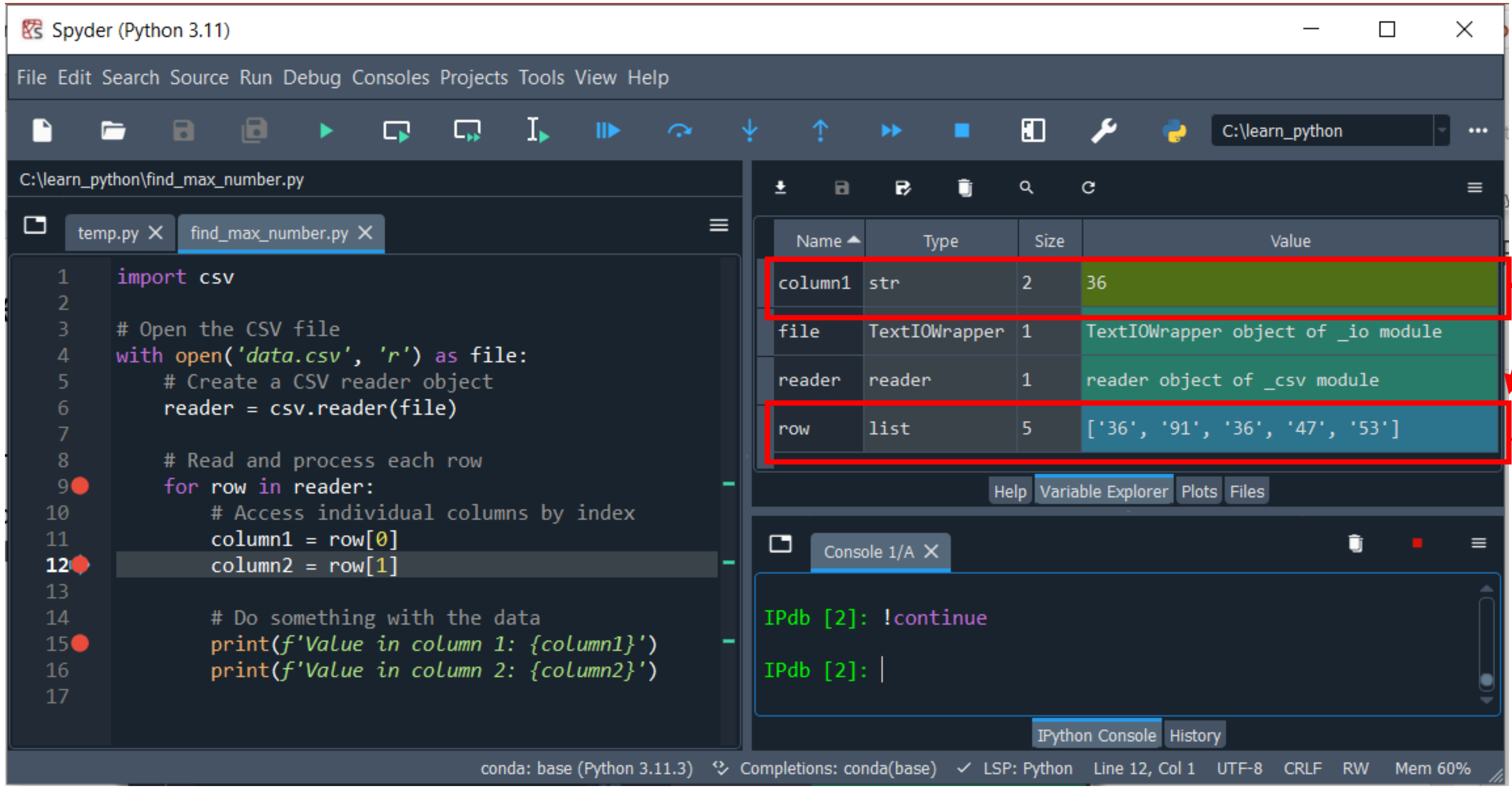
```
      7
      8      # Read and process each row
1---> 9      for row in reader:
     10          # Access individual columns by index
     11          column1 = row[0]
```

5. Creating Your First Python Program

5.2 Solve a Real Problem

1. Read the CSV file to access its content

8) Go to the next breakpoint until you can see the row variable is active and appears in the *Variable Explorer* pane



```
import csv

# Open the CSV file
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Read and process each row
    for row in reader:
        # Access individual columns by index
        column1 = row[0]
        column2 = row[1]

        # Do something with the data
        print(f'Value in column 1: {column1}')
        print(f'Value in column 2: {column2}')
```

Name	Type	Size	Value
column1	str	2	36
file	TextIOWrapper	1	TextIOWrapper object of _io module
reader	reader	1	reader object of _csv module
row	list	5	['36', '91', '36', '47', '53']

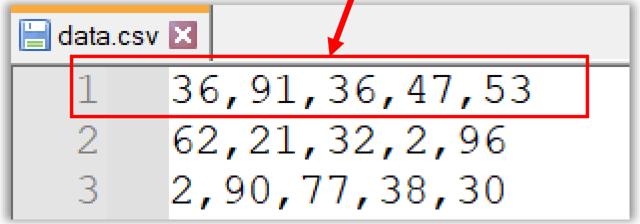
```
IPdb [2]: !continue

IPdb [2]: 
```

Do you find something interesting here?

Each row is read as a list that contains a series of str type elements

The list element is a str type, although it represents an integer!

Compared with the input data.csv file, the first row of numbers read into Python is correct!



```
36,91,36,47,53
62,21,32,2,96
2,90,77,38,30
```

We are done for the first step!

5. Creating Your First Python Program

5.2 Solve a Real Problem

Try applying what you have learned from this session for the rest of the steps. Modify the code and debug it, until you get the maximum number.

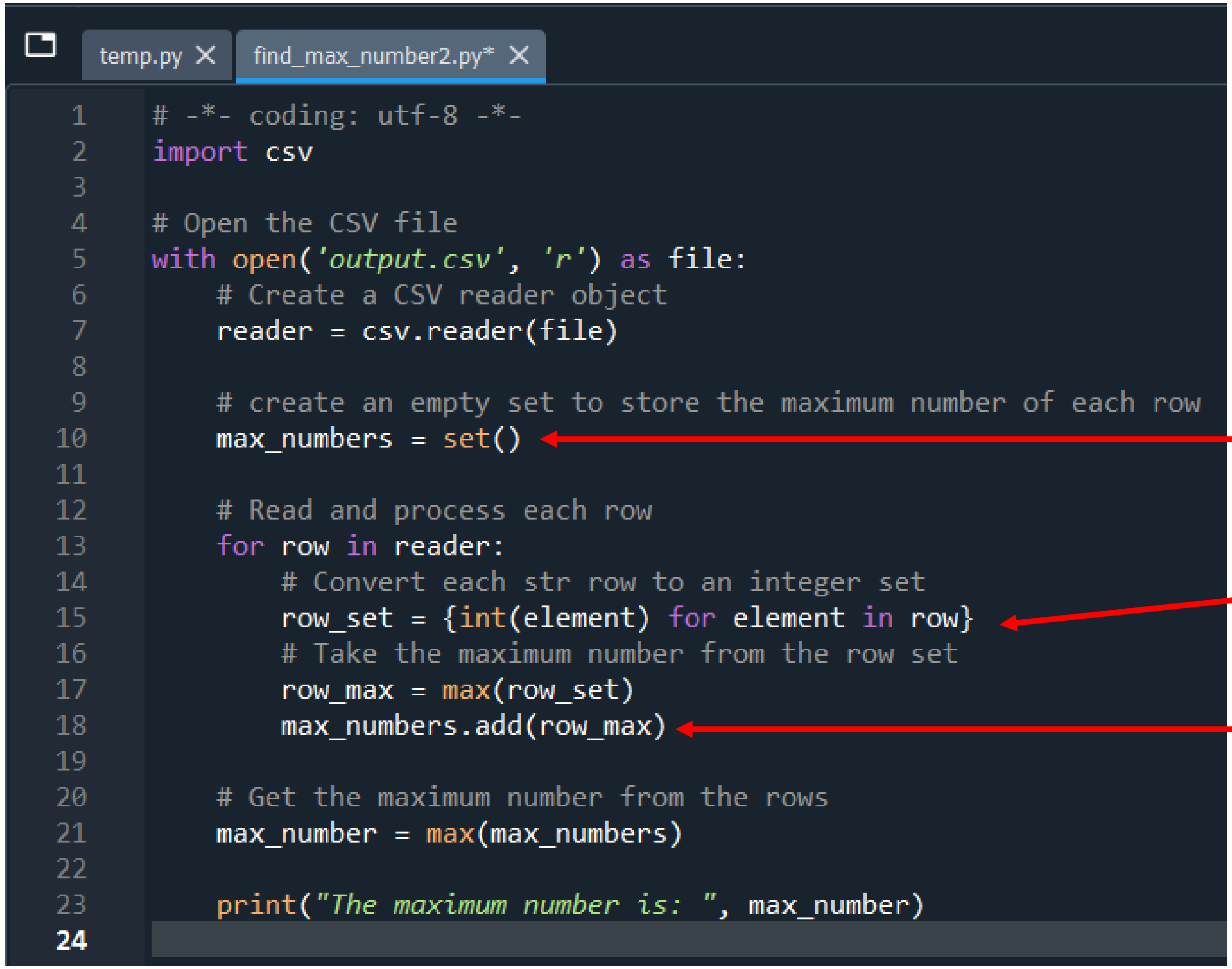
STOP

I will show you my solution on the next slide. But don’t peek at it. Try your solution first.

5. Creating Your First Python Program

5.2 Solve a Real Problem

Try applying what you have learned from this session for the rest of the steps. Modify the code and debug it, until you get the maximum number. There are many ways to solve the problem. I just show you my way to solve it:



```python
# -*- coding: utf-8 -*-
import csv

# Open the CSV file
with open('output.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # create an empty set to store the maximum number of each row
    max_numbers = set()

    # Read and process each row
    for row in reader:
        # Convert each str row to an integer set
        row_set = {int(element) for element in row}
        # Take the maximum number from the row set
        row_max = max(row_set)
        max_numbers.add(row_max)

    # Get the maximum number from the rows
    max_number = max(max_numbers)

    print("The maximum number is: ", max_number)

```

Why did I put this statement here instead of inside of the for loop?

What's this? Check "set comprehension"!

What did I do here?

I used built-in functions to make my life easier!

6. Assignment

Parse an XML file

Research the XML (eXtensible Markup Language) file structure via Google or ChatGPT. Write a Python script to analyze the provided XML file (books.xml), find the books that were published the earliest, and finally print out their titles and publish year. Make sure your logic can handle multiple books published in the same year.
Note, the book's title and author are random strings in the provided XML file, such as "SysXGxgMgX," and not the same as the following examples.

```xml
<?xml version='1.0' encoding='utf-8'?>
<catalog>
  <book id="1">
    <title>Python Crash Course</title>
    <author>Eric Matthes</author>
    <year>2015</year>
  </book>
  <book id="2">
    <title>Fluent Python</title>
    <author>Luciano Ramalho</author>
    <year>2015</year>
  </book>
</catalog>
```

A sample structure of the provided "books.xml" file

THANK YOU

Happy Coding!