

CS316 INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2 Python for Data Science

LECTURE 1 *Introduction to Python Programming*

Prof. Anis Koubaa

SEP 2024

PYTHON

WHY CHOOSE PYTHON

01

INTERPRETED LANGUAGE

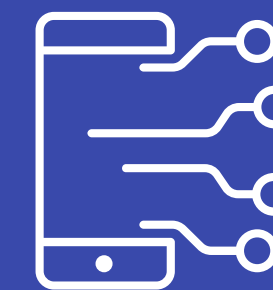
Python is a **interpreted, object-oriented, high-level** programming **scripting** language with **dynamic semantics**
Developed in the late 1980s by Guido van Rossum



02

PYTHON IS SIMPLE

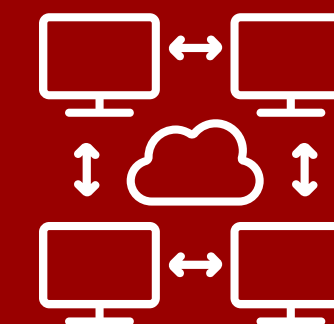
Python is simple and easy to learn syntax
Supports a large variety of standard libraries and packages
Increased productivity for developers – no segmentation fault

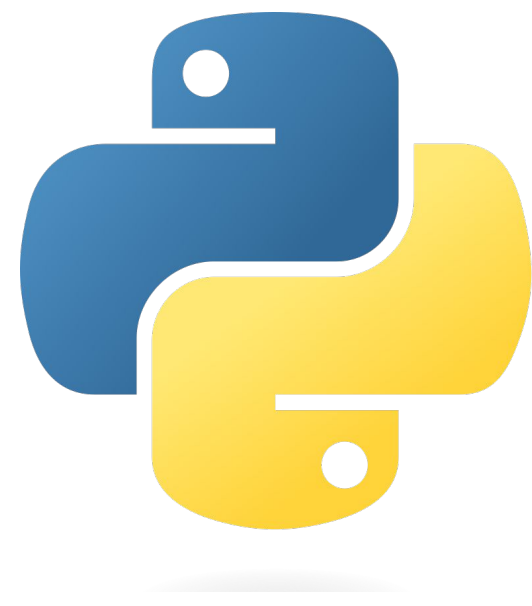


03

MULTI-PURPOSE LANGUAGE

Number 1 language for data science
Web development (back-end side – and recently front-end)
System programming (embedded)





Python Popularity

Prof. Anis Koubaa



SPECIALIZATION

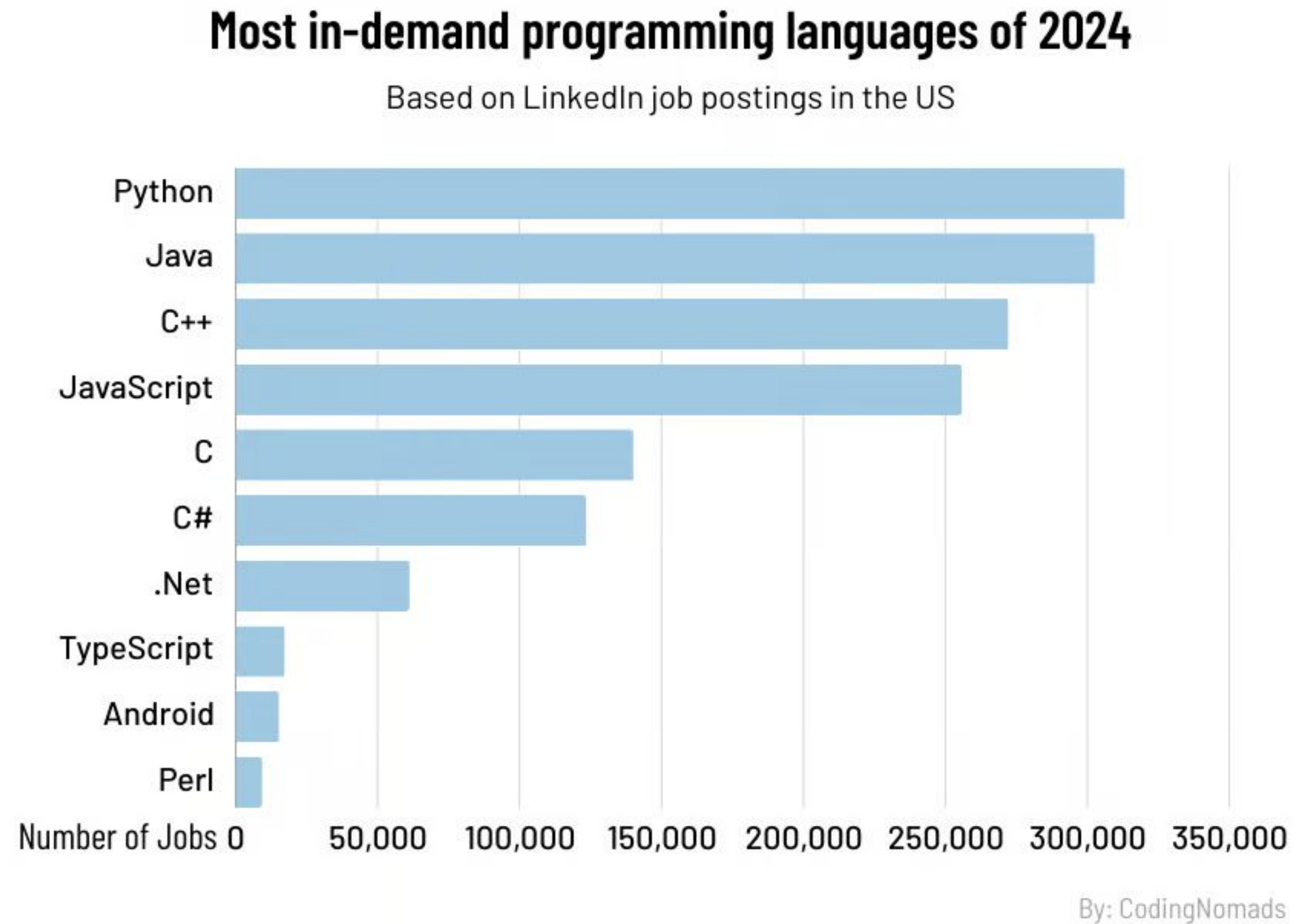
CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

PYTHON POPULARITY

IS PYTHON THE MOST POPULAR?

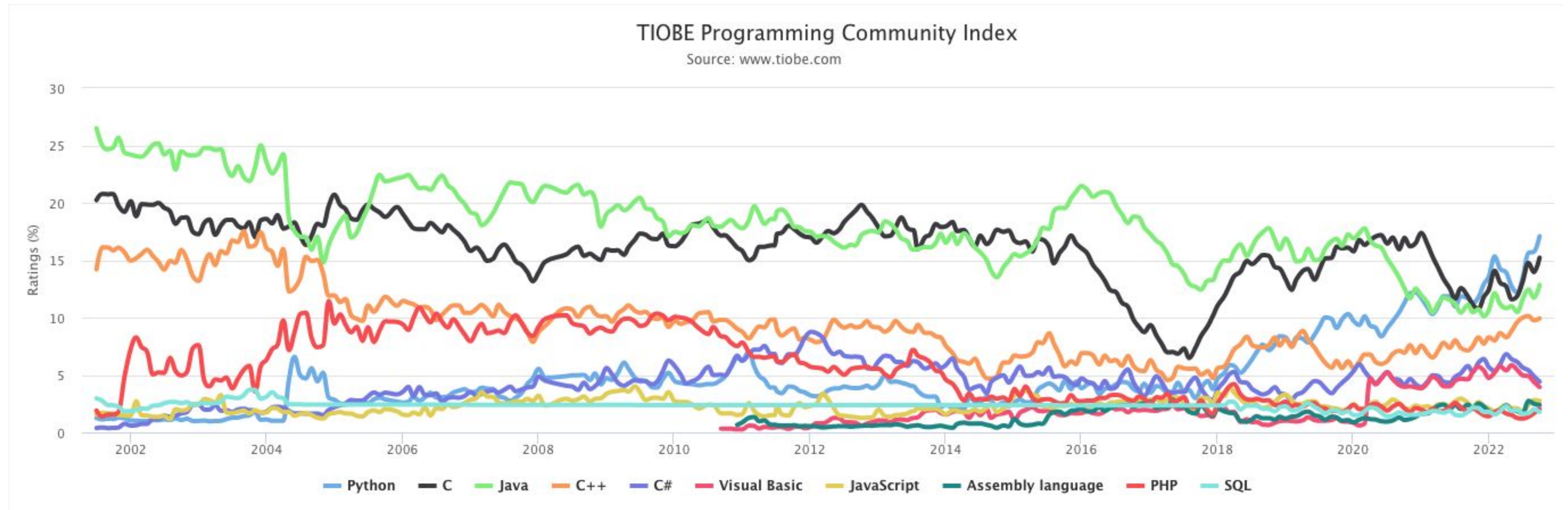


PYTHON POPULARITY

IS PYTHON THE MOST POPULAR?

Oct 2022	Oct 2021	Change	Programming Language		Ratings	Change
1	1			Python	17.08%	+5.81%
2	2			C	15.21%	+4.05%
3	3			Java	12.84%	+2.38%
4	4			C++	9.92%	+2.42%
5	5			C#	4.42%	-0.84%
6	6			Visual Basic	3.95%	-1.29%
7	7			JavaScript	2.74%	+0.55%
8	10	▲		Assembly language	2.39%	+0.33%
9	9			PHP	2.04%	-0.06%
10	8	▼		SQL	1.78%	-0.39%
11	12	▲		Go	1.27%	-0.01%
12	14	▲		R	1.22%	+0.03%
13	29	▲		Objective-C	1.21%	+0.76%
14	13	▼		MATLAB	1.18%	-0.02%
15	17	▲		Swift	1.05%	-0.06%
16	16			Ruby	0.88%	-0.24%
17	11	▼		Classic Visual Basic	0.87%	-0.96%
18	20	▲		Delphi/Object Pascal	0.85%	-0.09%
19	18	▼		Fortran	0.79%	-0.29%
20	26	▲		Rust	0.70%	+0.17%

PROGRAMMING LANGUAGES TRENDS



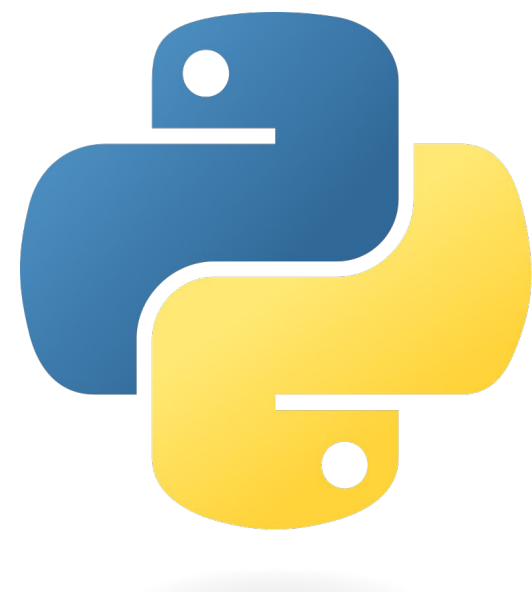
PROGRAMMING LANGUAGES

HISTORICAL RANKS

Programming Language	2022	2017	2012	2007	2002	1997	1992	1987
Python	1	5	8	7	12	28	-	-
C	2	2	1	2	2	1	1	1
Java	3	1	2	1	1	16	-	-
C++	4	3	3	3	3	2	2	6
C#	5	4	5	8	13	-	-	-
Visual Basic	6	15	-	-	-	-	-	-
JavaScript	7	8	10	9	8	23	-	-
Assembly language	8	10	-	-	-	-	-	-
SQL	9	-	-	-	7	-	-	-
PHP	10	7	6	5	6	-	-	-
Prolog	25	32	32	26	16	20	13	3
Lisp	31	31	13	16	14	9	5	2
Pascal	269	117	14	21	99	11	3	5
(Visual) Basic	-	-	7	4	4	3	6	4

BIG TECH APPS USING PYTHON





Install Python

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Python Installation Options

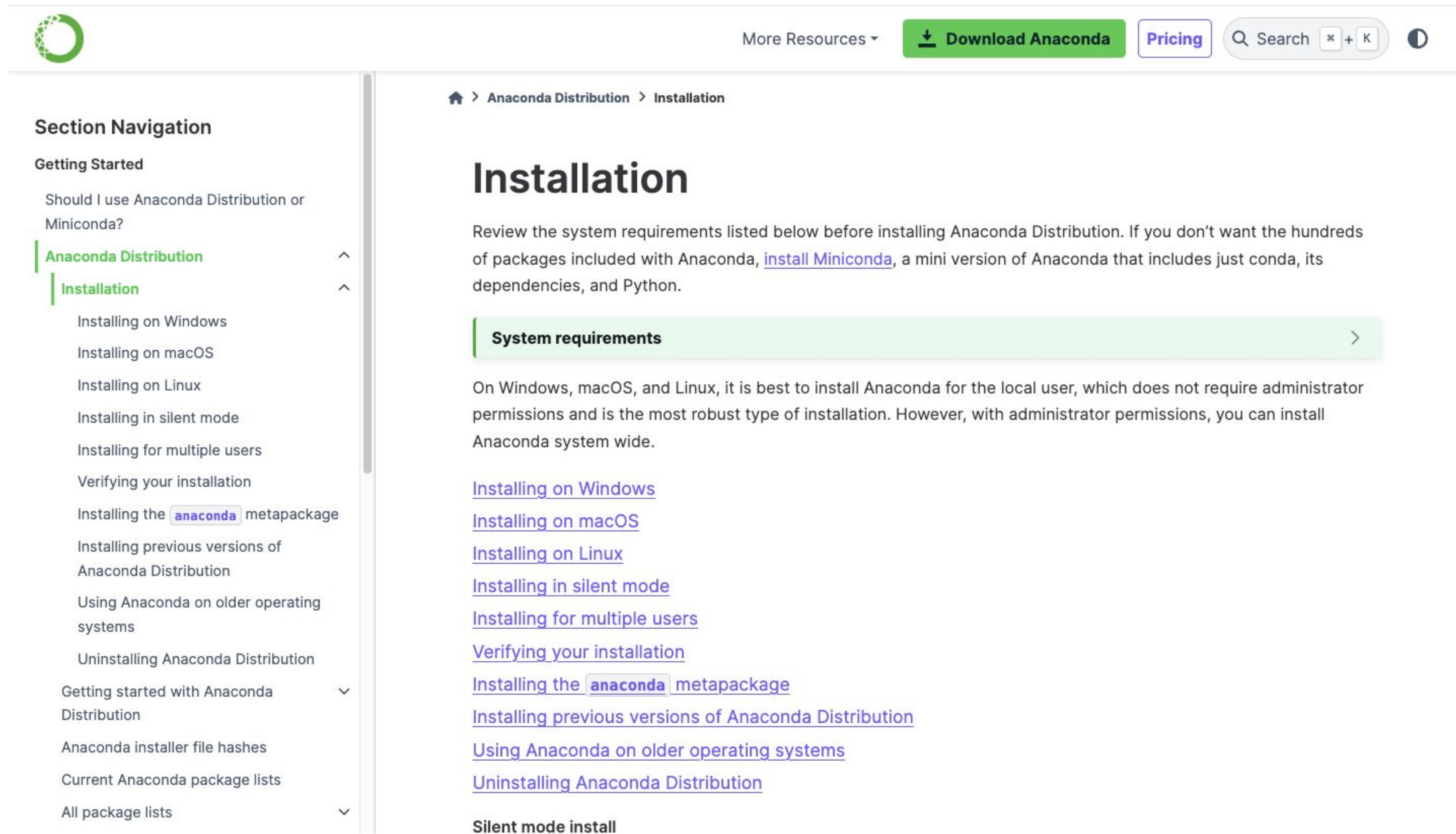
- **Official Python Distribution**

- **Availability:** Windows, Mac, Linux
- **Source:** Download directly from python.org
- **Instructions:** Follow installation steps for your OS, ensuring the addition of Python to the system PATH.

- **Anaconda Distribution (Recommended for Data Science)**

- **Includes:** Python, Conda (environment manager), and key pre-installed libraries (e.g., NumPy, Pandas, SciPy).
- **Compatibility:** Available for Windows, Mac, and Linux
- **Download:** Obtain from Anaconda.com
- **Advantages:** Simplifies package management and deployment with isolated environments.

Python Installation Options



The screenshot shows the Anaconda Distribution Installation page. The top navigation bar includes the Anaconda logo, a 'More Resources' dropdown, a green 'Download Anaconda' button, a 'Pricing' button, a search bar, and a user profile icon. The left sidebar, titled 'Section Navigation', lists various topics under 'Getting Started', with 'Anaconda Distribution' and 'Installation' highlighted. The main content area, titled 'Installation', provides an overview of the system requirements and a list of links for installing on different operating systems and in different modes. A green box labeled 'System requirements' is prominently displayed.

Section Navigation

- Getting Started
 - Should I use Anaconda Distribution or Miniconda?
 - Anaconda Distribution**
 - Installation**
 - Installing on Windows
 - Installing on macOS
 - Installing on Linux
 - Installing in silent mode
 - Installing for multiple users
 - Verifying your installation
 - Installing the `anaconda` metapackage
 - Installing previous versions of Anaconda Distribution
 - Using Anaconda on older operating systems
 - Uninstalling Anaconda Distribution
 - Getting started with Anaconda Distribution
 - Anaconda installer file hashes
 - Current Anaconda package lists
 - All package lists

Installation

Review the system requirements listed below before installing Anaconda Distribution. If you don't want the hundreds of packages included with Anaconda, [install Miniconda](#), a mini version of Anaconda that includes just conda, its dependencies, and Python.

System requirements

On Windows, macOS, and Linux, it is best to install Anaconda for the local user, which does not require administrator permissions and is the most robust type of installation. However, with administrator permissions, you can install Anaconda system wide.

[Installing on Windows](#)

[Installing on macOS](#)

[Installing on Linux](#)

[Installing in silent mode](#)

[Installing for multiple users](#)

[Verifying your installation](#)

[Installing the `anaconda` metapackage](#)

[Installing previous versions of Anaconda Distribution](#)

[Using Anaconda on older operating systems](#)

[Uninstalling Anaconda Distribution](#)

Silent mode install

<https://docs.anaconda.com/anaconda/install/>

Installing Python from the Terminal

1. Check Existing Python Installation

- **Command:** `python --version` Or `python3 --version`
- **Purpose:** Verify if Python is already installed and check the current version.

Installing Python from the Terminal

1. Installing Python on Different OS

- **Linux:**

- **Update Package List:** `sudo apt-get update`

- **Install Python:**

- **Python 3:** `sudo apt-get install python3`

- **Pip (Package Manager):** `sudo apt-get install python3-pip`

Installing Python from the Terminal

1. Installing Python on Different OS

- **macOS:**

- **Using Homebrew:**

- **Install Homebrew (if not installed):**

- ```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- **Install Python:**

- ```
brew install python
```

Installing Python from the Terminal

1. Installing Python on Different OS

- **Windows:**

- **Using Windows Subsystem for Linux (WSL):**

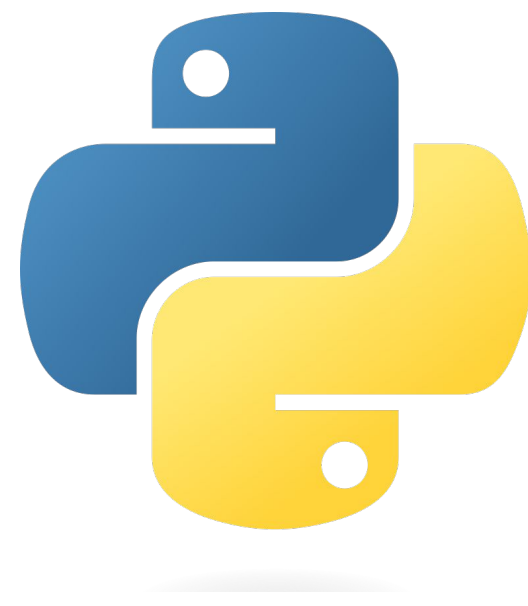
- **Enable WSL:**

- `wsl --install`

- **Install Python within WSL:**

- **Update:** `sudo apt-get update`

- **Install:** `sudo apt-get install python3`



Python IDE

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Development Environments

1. IDLE

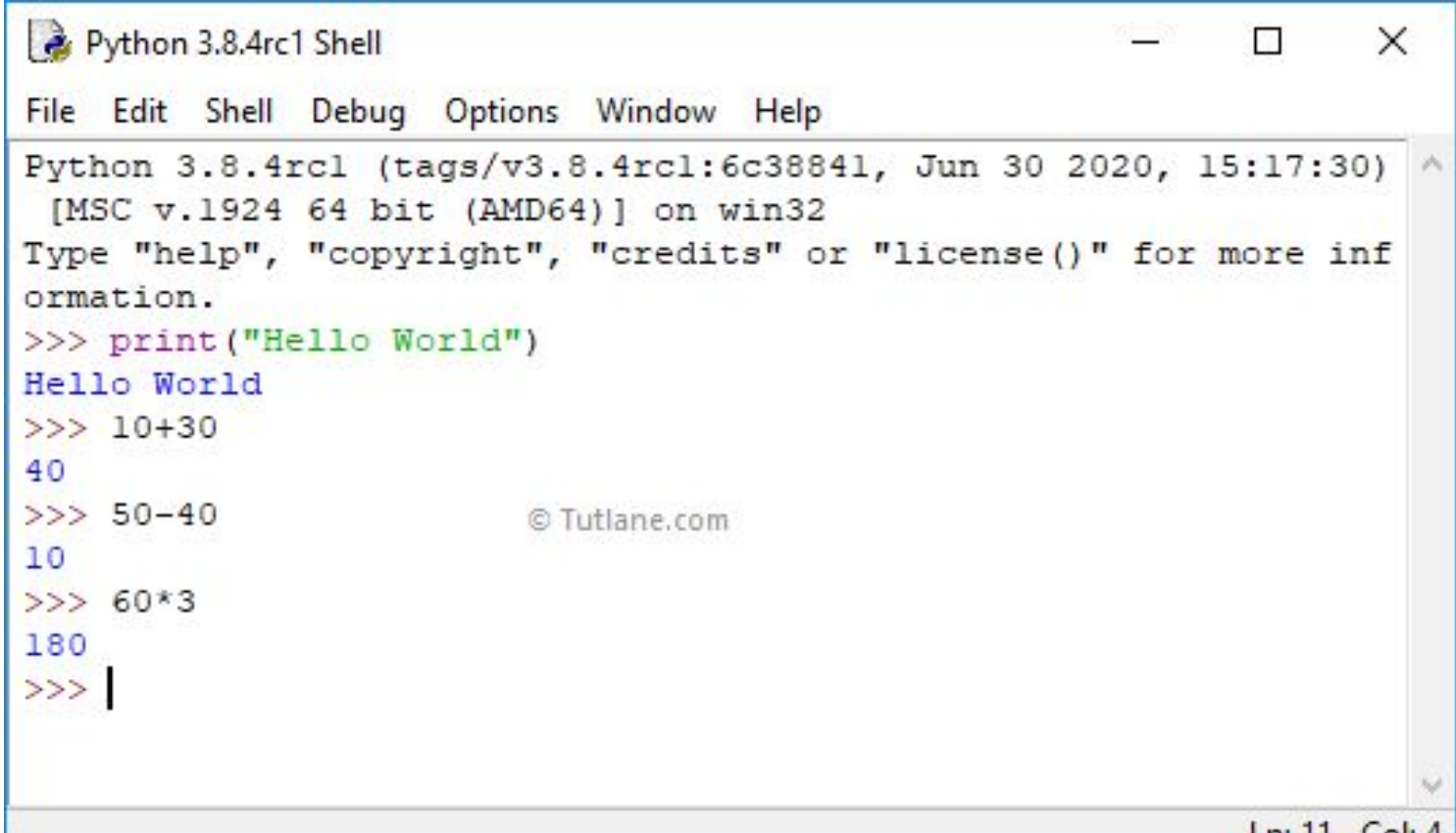
- **Description:** Integrated Development and Learning Environment, comes pre-installed with Python.
- **Use Case:** Suitable for beginners and small-scale projects.

2. PyCharm

- **Versions:** Community Edition (free) and Professional Edition.
- **Features:** Tailored for Python development with advanced debugging, code analysis, and refactoring tools.

3. Visual Studio Code (VS Code)

- **Description:** Free, open source, and highly customizable.
- **Python Support:** Enhanced through extensions like Python and Pylance, ideal for various development tasks.



```
Python 3.8.4rc1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.4rc1 (tags/v3.8.4rc1:6c38841, Jun 30 2020, 15:17:30)
[MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more inf
ormation.
>>> print("Hello World")
Hello World
>>> 10+30
40
>>> 50-40
10
>>> 60*3
180
>>> |
```



Notebook Development Environments

1. Jupyter Notebook

- **Description:** Ideal for interactive Python sessions, especially in data science.
- **Access:** Available via Anaconda distribution or `pip install notebook`.



2. Google Colab

- **Versions:** Free cloud-based Jupyter notebook environment.
- **Advantages:** Provides GPU/TPU support, easy sharing, and no local setup required.





01-python-crash-course-part-01.ipynb

File Edit View Insert Runtime Tools Help Last edited on Dec 31, 2019

+ Code + Text

Part 3

- Functions
- Lambda Expressions
- Map and Filters

This review is not intended to be comprehensive, but provides to the main features of the Python programming language, which should good enough to proceed further.

+ Code + Text

▼ 1. Basic Syntax

We start by printing a simple Hello World text message using the method 'print'. Note that in Python 3 is necessary to put the text into paranthesis, where as in Python 2 it was optional

```
[ ] #print hello world
    print ("Hello world, Python3!")
```

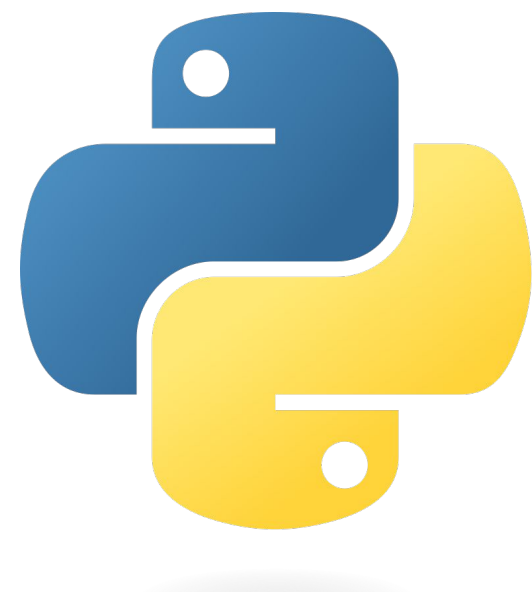
Hello world, Python3!

▼ 1.1 Indentation

Indentation is very important in Python. In termine blocks as no parenthesis are used. This is an example of an if statement

```
[ ] i = 0
    if (i==0):
        i=i+1
        print ("i=",i)
    n=1
    print ("n=",n)
```

i= 1
n= 1



Python Libraries

Prof. Anis Koubaa



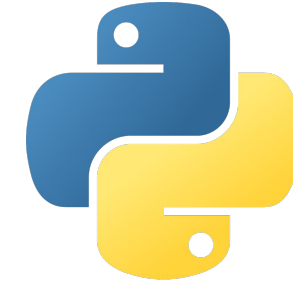
SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

PYTHON FOR DATA SCIENCE LIBRARIES/PACKAGES



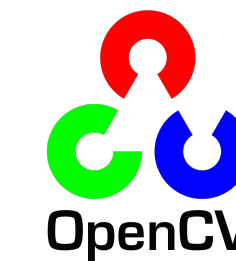
DATA STRUCTURES



VISUALIZATION



SCIENTIFIC COMPUTING



STORAGE AND DATABASES



DEEP LEARNING



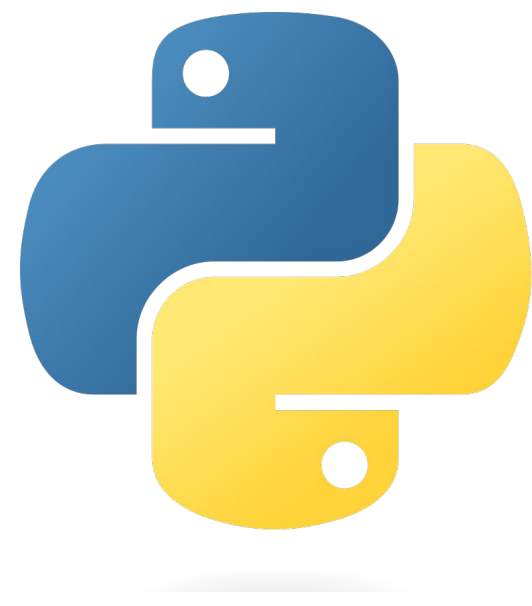
CS316 INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2 Python for Data Science

LECTURE 2 *Introduction to Python Programming* *Virtual Envs, Setting-up a Python Package*

Prof. Anis Koubaa

SEP 2024



Python Package Structure

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Python Module

1. Definition

- A single file containing Python code, typically with a `.py` extension.

2. Example

- `example.py`

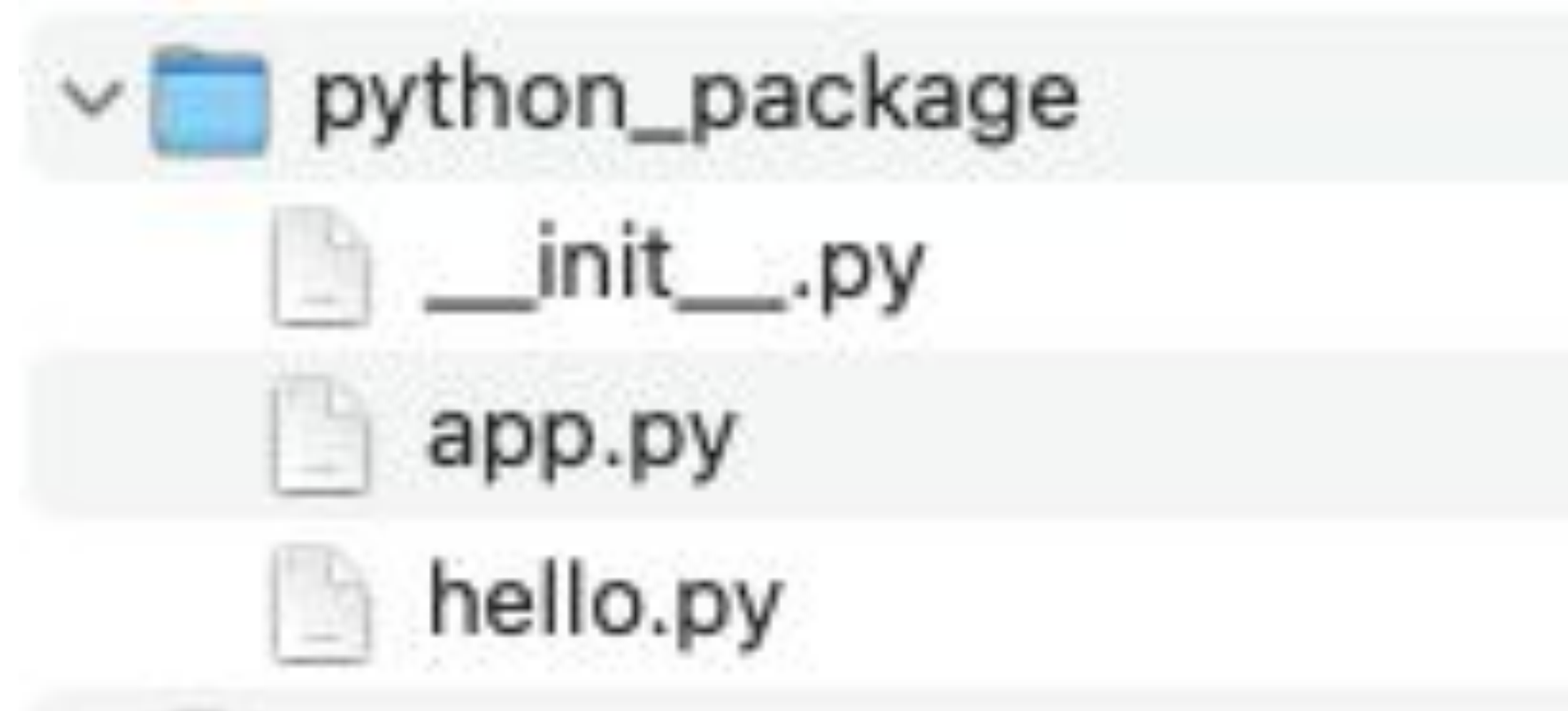
3. Usage

- Functions, classes, and variables defined in the file can be imported into other Python scripts.
- Import Syntax: `import example`
- Example Code:

```
python Copy code

# In example.py
def greet():
    print("Hello, world!")

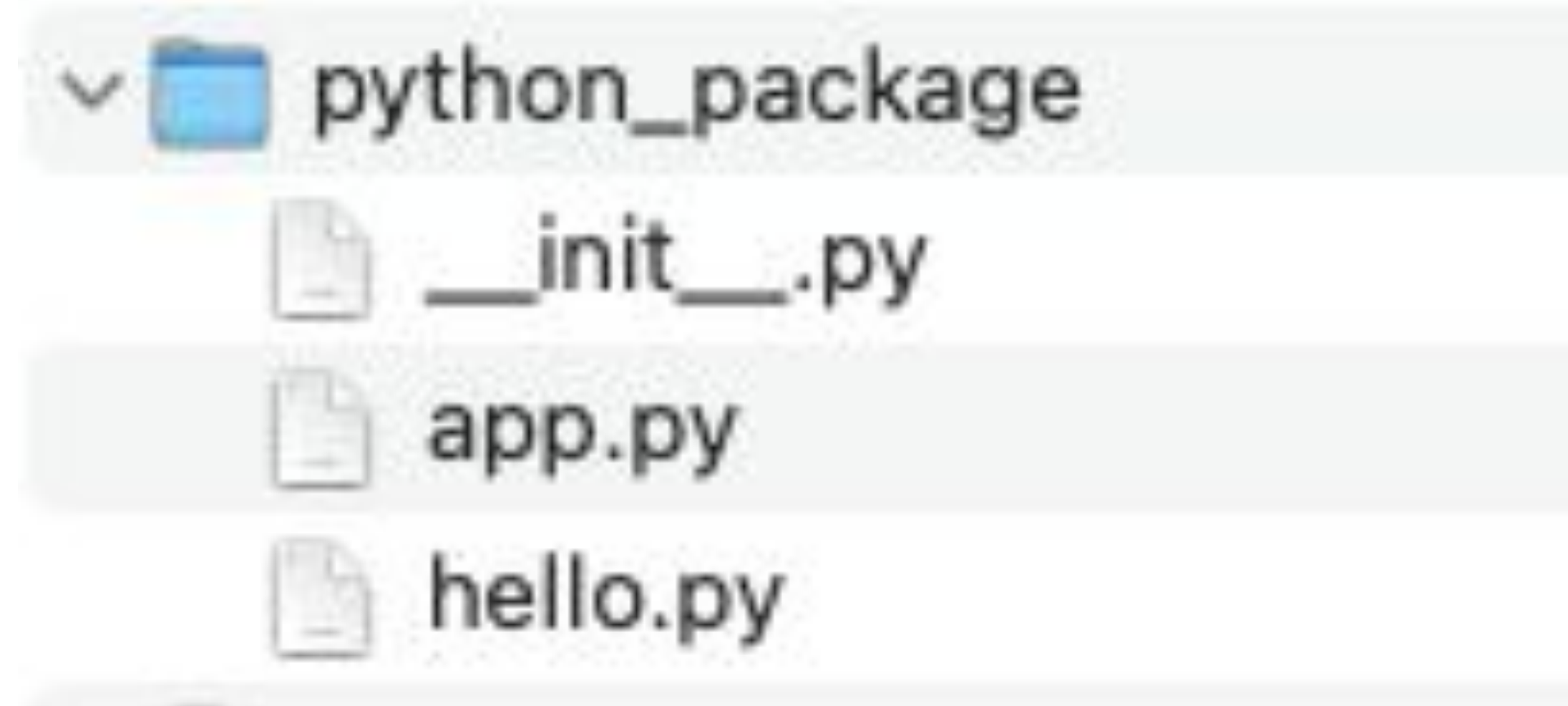
# In another script
import example
example.greet()
```



Python Package

2. Package

- **Definition:** A directory (folder) containing multiple modules, and an `__init__.py` file.
- **Purpose:** Organizes related modules under a single namespace.
- **Example Structure:**
 - ``mypackage/``
 - `__init__.py` (can be empty)
 - `module1.py`
 - `module2.py`
- **Import Syntax:**
 - ``from mypackage import module1``
 - ``import mypackage.module2``



Python Libraries

3. Standard Library

- **Description:** A rich set of modules and packages included with Python by default.
- **Purpose:** Provides tools for various tasks such as file I/O, system calls, and web development.
- **Examples:** ``os``, ``sys``, ``datetime``, ``json``

4. Third-Party Packages

- **Definition:** External libraries developed by the community, available through package managers like ``pip``.
- **Purpose:** Extend Python's functionality for specific use cases, such as data science or web development.
- **Examples:** ``numpy``, ``requests``, ``pandas``

```
python Copy code

# Importing standard library module
import datetime

# Importing third-party library module
import pandas as pd

def main():
    # Using datetime to get today's date
    today = datetime.date.today()

    # Using pandas to create a DataFrame with today's date
    df = pd.DataFrame({'Date': [today]})
    print(df)

if __name__ == "__main__":
    main()
```

PYTHON FILE STRUCTURE

MODULE vs. PACKAGE

PACKAGE

A collection of Python module. It has a `__init__.py` file to distinguish it from a classical director

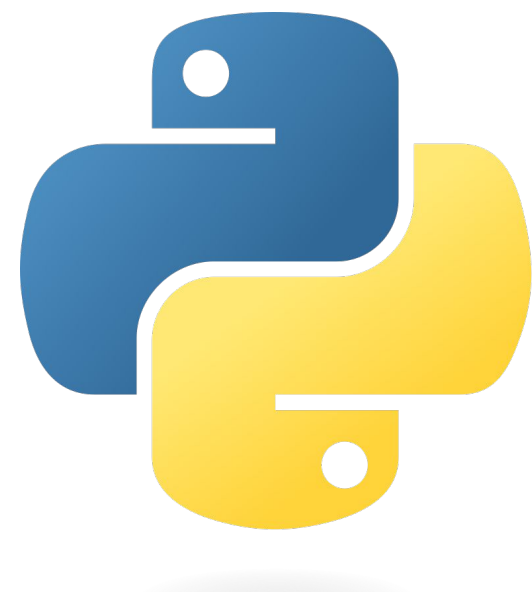
MODULE-1

A module is a file that contains a Python script in runtime for the code specified to the users.

MODULE-2

A module is a file that contains a Python script in runtime for the code specified to the users.





Python Virtual Environment

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Creating a Virtual Environment with **pip**

2. Creating a Virtual Environment with `pip`

- Steps:

1. Install `venv` (if not installed):

```
bash Copy code  
  
python -m ensurepip --upgrade
```

2. Create the Environment:

```
bash Copy code  
  
python -m venv myenv
```

3. Activate the Environment:

- Windows:

```
bash Copy code  
  
myenv\Scripts\activate
```

- macOS/Linux:

```
bash Copy code  
  
source myenv/bin/activate
```

4. Deactivate:

```
bash Copy code  
  
deactivate
```

Creating a Virtual Environment with conda

3. Creating a Virtual Environment with `conda`

- Steps:

1. Create the Environment:

```
bash Copy code  
  
conda create --name myenv
```

2. Activate the Environment:

```
bash Copy code  
  
conda activate myenv
```

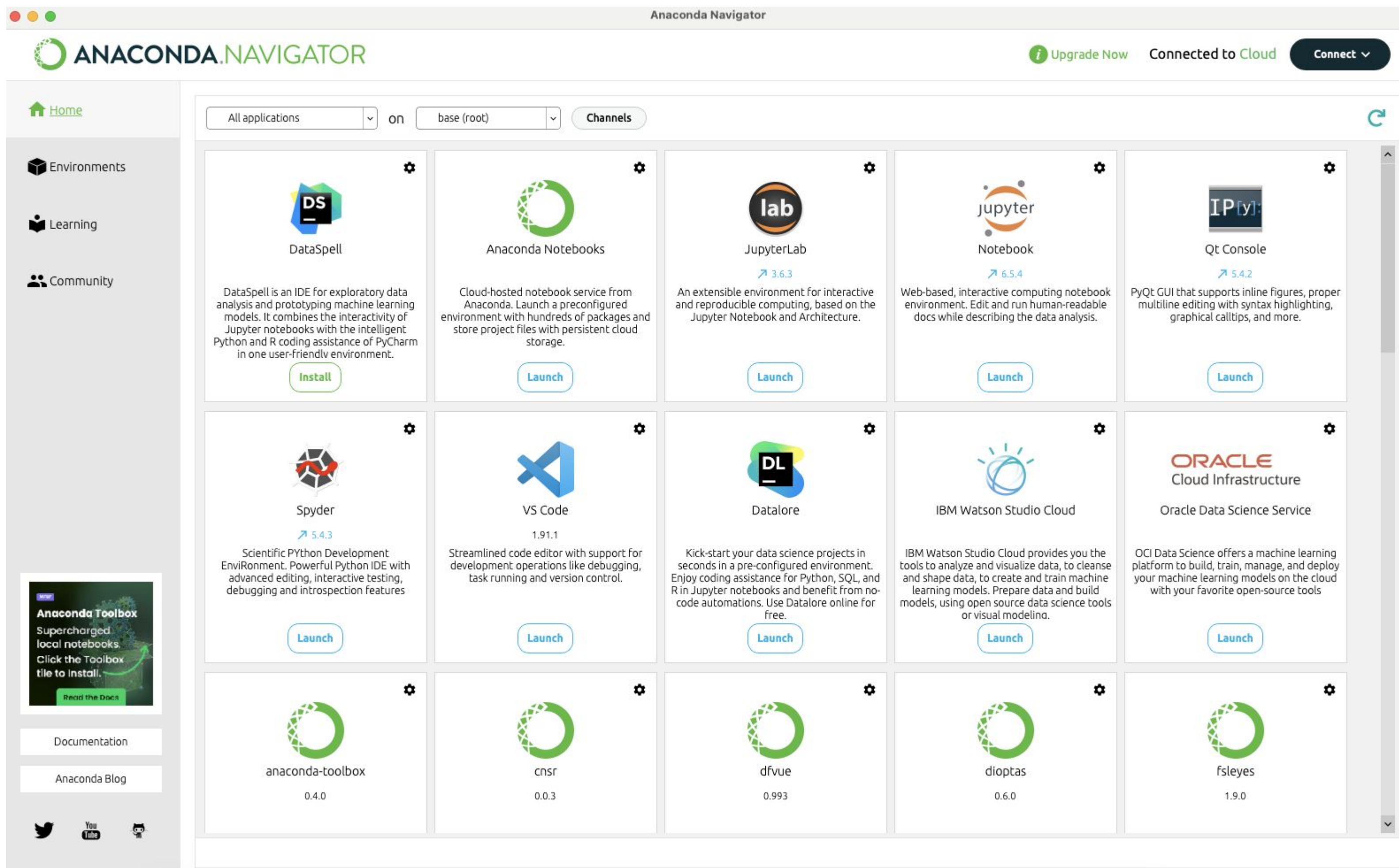
3. Deactivate:

```
bash Copy code  
  
conda deactivate
```

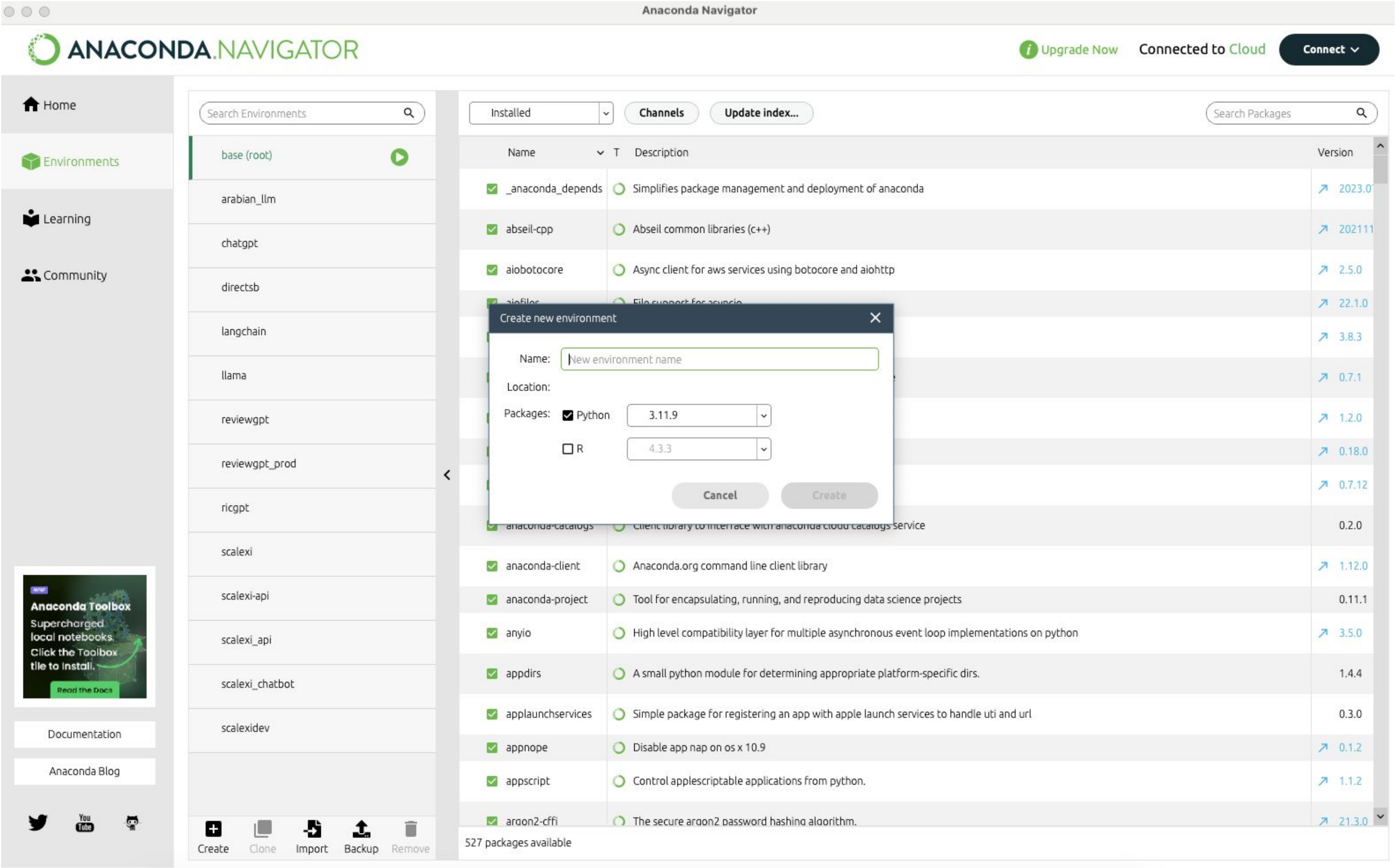
4. Remove the Environment:

```
bash Copy code  
  
conda remove --name myenv --all
```

Creating a Virtual Environment with conda



Creating a Virtual Environment with conda



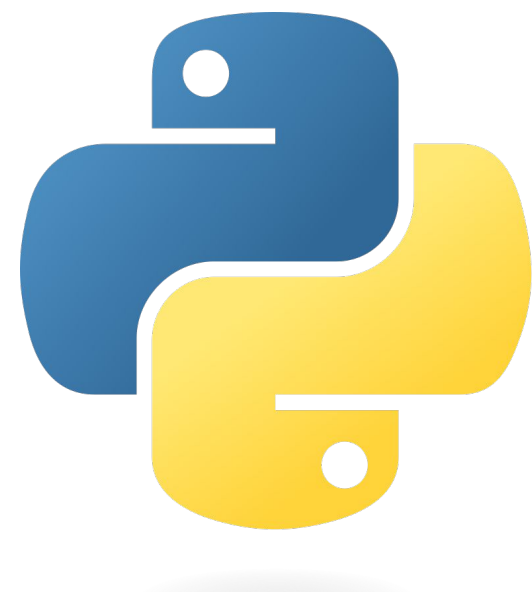
Managing Packages within the Environment

4. Managing Packages within the Environment

- Using `pip`:
 - Install a package: `pip install package_name`
 - List installed packages: `pip list`
- Using `conda`:
 - Install a package: `conda install package_name`
 - List installed packages: `conda list`

5. Considerations

- `pip`: Works with the standard Python distribution; great for most projects.
- `conda`: Ideal for data science projects with complex dependencies, especially when working with packages that require compiled binaries (e.g., `numpy`, `scipy`).



Setup a Python Package

1. Package Structure

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Python Package Structure

1. Package Structure

- Structure Overview:

```
plaintext Copy code

mypackage/
├── mypackage/
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
├── tests/
│   ├── test_module1.py
│   └── test_module2.py
├── setup.py
├── README.md
└── LICENSE
```

- Key Files:

- `__init__.py`: Marks the directory as a Python package.
- `setup.py`: Script for packaging and distribution.
- `README.md`: Documentation for the package.
- `LICENSE`: License information.

Writing `setup.py`

2. Basic `setup.py` Script Example

```
python Copy code

from setuptools import setup, find_packages

setup(
    name="mypackage",
    version="0.1.0",
    author="Your Name",
    author_email="your.email@example.com",
    description="A short description of the package",
    long_description=open("README.md").read(),
    long_description_content_type="text/markdown",
    url="https://github.com/yourusername/mypackage",
    packages=find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    python_requires='>=3.6',
    install_requires=[
        "numpy>=1.21.0", # Numerical computations
        "pandas>=1.3.0", # Data manipulation and analysis
        "requests>=2.26.0", # HTTP library for making requests
        "matplotlib>=3.4.0" # Plotting and visualization
    ],
    entry_points={
        'console_scripts': [
            'mypackage-cli=mypackage.command_line:main',
        ],
    },
)
```

1. Purpose

- `setup.py` is the script used for packaging and distributing your Python project. It contains metadata about the package and specifies dependencies.

3. Explanation

- `name`: The name of the package.
- `version`: The current version of the package.
- `author` and `author_email`: Your details.
- `description`: A brief description of the package.
- `long_description`: Detailed description, often pulled from `README.md`.
- `url`: The URL of the project's repository or website.
- `packages`: Automatically find all packages to include in the distribution.
- `classifiers`: Metadata about the package (e.g., programming language, license).
- `python_requires`: Specifies the Python version compatibility.
- `install_requires`: Lists the dependencies that will be installed with the package.
 - Examples:
 - `numpy`: A library for numerical computations.
 - `pandas`: A library for data manipulation and analysis.
 - `requests`: A library for making HTTP requests.
 - `matplotlib`: A library for plotting and visualization.
- `entry_points`: Defines entry points for creating command-line scripts.

README

MyPackage

Overview

MyPackage is a simple Python package designed for educational purposes. It provides basic utility functions that demonstrate key concepts in Python, such as string manipulation, mathematical operations, and data handling.

Features

- **String Utilities:** Functions to manipulate and analyze strings.
- **Math Utilities:** Basic mathematical operations, including addition, subtraction, multiplication, and division.
- **Data Utilities:** Simple data processing functions.

Installation

You can install **MyPackage** using `pip`:

```
bash
pip install mypackage
```

Alternatively, you can clone the repository and install it manually:

```
bash
git clone https://github.com/yourusername/mypackage.git
cd mypackage
pip install .
```

Usage

Below are some examples of how to use **MyPackage**:

1. String Utilities

```
python
from mypackage import string_utils

# Reverse a string
reversed_string = string_utils.reverse_string("hello")
print(reversed_string) # Output: "olleh"

# Count vowels in a string
vowel_count = string_utils.count_vowels("hello")
print(vowel_count) # Output: 2
```

README

Usage

Below are some examples of how to use MyPackage:

1. String Utilities

```
python
Copy code

from mypackage import string_utils

# Reverse a string
reversed_string = string_utils.reverse_string("hello")
print(reversed_string) # Output: "olleh"

# Count vowels in a string
vowel_count = string_utils.count_vowels("hello")
print(vowel_count) # Output: 2
```

2. Math Utilities

```
python
Copy code

from mypackage import math_utils

# Add two numbers
result = math_utils.add(5, 3)
print(result) # Output: 8

# Divide two numbers
result = math_utils.divide(10, 2)
print(result) # Output: 5.0
```

3. Data Utilities

```
python
Copy code

from mypackage import data_utils

# Sum of a list of numbers
total = data_utils.sum_list([1, 2, 3, 4])
print(total) # Output: 10

# Find maximum in a list of numbers
maximum = data_utils.find_max([1, 2, 3, 4])
print(maximum) # Output: 4
```

Modules

1. string_utils

- `reverse_string(s: str) -> str`: Reverses the given string `s`.
- `count_vowels(s: str) -> int`: Counts the number of vowels in the given string `s`.

2. math_utils

- `add(a: int, b: int) -> int`: Returns the sum of `a` and `b`.
- `subtract(a: int, b: int) -> int`: Returns the difference between `a` and `b`.
- `multiply(a: int, b: int) -> int`: Returns the product of `a` and `b`.
- `divide(a: int, b: int) -> float`: Returns the division of `a` by `b`.

3. data_utils

- `sum_list(numbers: list) -> int`: Returns the sum of all numbers in the list.

README

Contributing

Contributions are welcome! If you have any suggestions or find any bugs, please feel free to open an issue or submit a pull request.

Steps to Contribute:

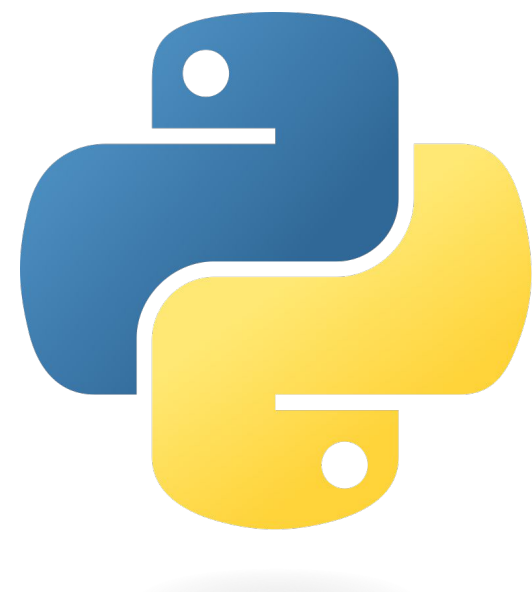
1. Fork the repository.
2. Create a new branch (`git checkout -b feature-branch`).
3. Make your changes.
4. Commit your changes (`git commit -m 'Add new feature'`).
5. Push to the branch (`git push origin feature-branch`).
6. Open a pull request.

License

This project is licensed under the MIT License - see the [LICENSE](#) file for details.

Contact

For any questions or suggestions, feel free to contact me at your.email@example.com.



Setup a Python Package

2. PyPi Repository

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Understanding PyPI

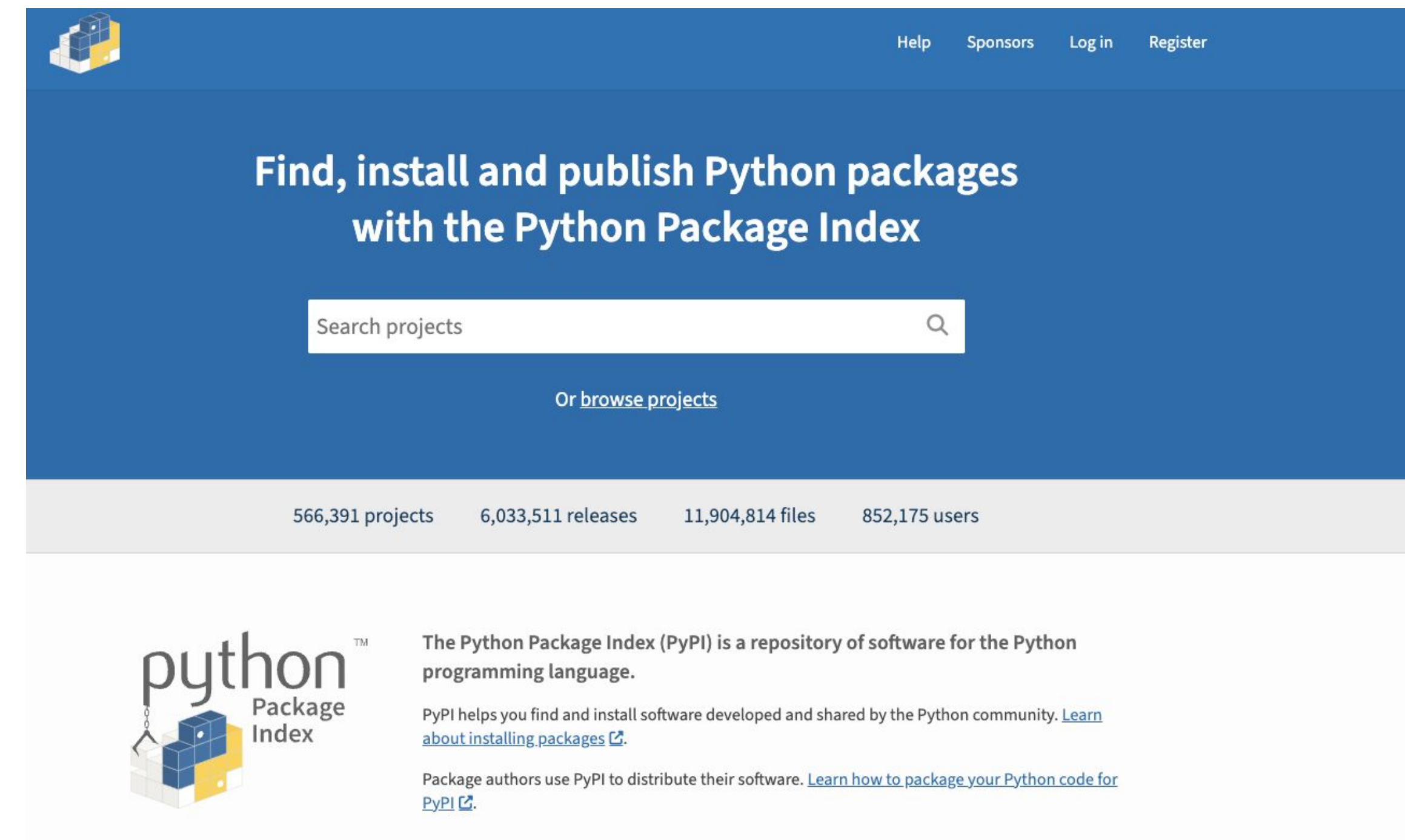
<https://pypi.org/>

1. What is PyPI?

- **Definition:** The Python Package Index (PyPI) is the official third-party software repository for Python.
- **Purpose:** Allows developers to distribute their Python software and libraries to users.

2. Key Features

- **Package Management:** Easy installation and updating of packages using `pip`.
- **Searchable:** Users can search for and discover packages based on functionality and popularity.
- **Community-Driven:** Contributions from developers around the world.



Understanding PyPI



numpy

HelpSponsorsLog inRegister

Filter by [classifier](#)

Framework

Topic

Development Status

License

Programming Language

Operating System

Environment

Intended Audience

Natural Language

Typing

10,000+ projects for "numpy"

Order by

Relevance

 **numpy 2.1.0**

Aug 19, 2024

Fundamental package for array computing in Python

 **numpy1 0.0.1**

Jan 24, 2021

Python

 **numpy4 0.7**

Apr 26, 2024

A package for doing cool stuff

 **numpy5 0.1**

Apr 28, 2024

None

 **numpy6 2.0**

Apr 29, 2024

None

 **gdal2numpy 0.0.336**

Aug 6, 2024

An utils functions package

Example

Your account

Your projects

Your organizations

Account settings

Publishing

Your projects1

scalexiSOLE OWNER

Last released Aug 7, 2024

The scalexi package is a versatile open-source Python library that focuses on facilitating low-code development and fine-tuning of diverse Large Language Models (LLMs). It extends beyond its initial OpenAI models integration, offering a scalable framework for various LLMs.

Manage

View

CS316: INTRODUCTION TO DATA SCIENCE

INTRODUCTION TO DATA SCIENCE

ANIS KOUBAA | 2024 413

Example



 scalexi ▾

scalexi 0.4.7.16

✓ Latest version

Released: Aug 7, 2024

`pip install scalexi`



The scalexi package is a versatile open-source Python library that focuses on facilitating low-code development and fine-tuning of diverse Large Language Models (LLMs). It extends beyond its initial OpenAI models integration, offering a scalable framework for various LLMs.

Manage project

Navigation

Project description

Release history

Download files

Verified details [\(What is this?\)](#)

These details have been verified by PyPI

Maintainers

 scalexi

Project description

Overview

scalexi is a versatile open-source Python library that focuses on facilitating low-code development and fine-tuning of diverse Large Language Models (LLMs). It extends beyond its initial OpenAI models integration, offering a scalable framework for various LLMs.

Key to scalexi is its low-code approach, significantly reducing the complexity of dataset preparation and manipulation. It features advanced dataset conversion tools, adept at transforming raw contextual data into structured datasets fulfilling LLMs fine-tuning requirements. These tools support multiple question formats, like open-ended, closed-ended, yes-no, and reflective queries, streamlining the creation of customized datasets for LLM fine-tuning.

A standout feature is the library's automated dataset generation, which eases the workload involved in LLM training.

Best Practices for PyPI

1. Versioning

- Use Semantic Versioning: Follow the `MAJOR.MINOR.PATCH` format.
- Example: `1.0.0` for initial stable release.

2. Documentation

- Include a `README.md`: Helps users understand the package's purpose and usage.
- Provide Examples: Demonstrate how to use your package.

3. Testing Before Upload

- Use TestPyPI: Upload to TestPyPI to ensure everything works correctly before the official release.
- Continuous Integration: Implement CI/CD pipelines to automate testing and deployment.

Setting Up PyPI Credentials for Uploading a Wheel

1. Create a PyPI Account

- Steps:
 1. Go to pypi.org.
 2. Click on "Register" and create a new account.
 3. Verify your email address to activate your account.

2. Install `twine`

- **Purpose:** `twine` is the recommended tool for securely uploading your distribution packages to PyPI.
- **Command:**

```
bash
```

[Copy code](#)

```
pip install twine
```

Setting Up PyPI Credentials for Uploading a Wheel

3. Set Up PyPI Credentials

- Option 1: Use `.pypirc` File (Recommended)
 1. Create a `.pypirc` file in your home directory (`~/.pypirc` on Linux/macOS or `%USERPROFILE%\pypirc` on Windows).
 2. Add the following configuration:

```
ini                                                                    Copy code

[distutils]
index-servers =
    pypi

[pypi]
username = your-username
password = your-password
```

3. Explanation:

- The `username` and `password` are your PyPI credentials.
- This file allows you to avoid entering your credentials every time you upload a package.

Setting Up PyPI Credentials for Uploading a Wheel

3. Set Up PyPI Credentials

- Option 1: Use `.pypirc` File (Recommended)
 1. Create a `.pypirc` file in your home directory (`~/.pypirc` on Linux/macOS or `%USERPROFILE%\pypirc` on Windows).
 2. Add the following configuration:

```
ini Copy code

[distutils]
index-servers =
    pypi

[pypi]
username = your-username
password = your-password
```

3. Explanation:

- The `username` and `password` are your PyPI credentials.
- This file allows you to avoid entering your credentials every time you upload a package.

Option 2: Manual Entry

- When using `twine` without a `.pypirc` file:

```
bash Copy code

twine upload dist/*
```

- Prompt: `twine` will prompt you for your PyPI username and password during the upload process.

4. Upload the Wheel to PyPI

- Build the Wheel:

```
bash Copy code

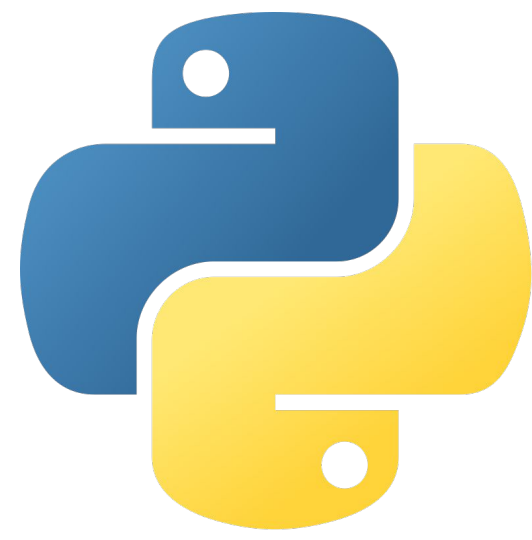
python setup.py bdist_wheel
```

- Upload the Package:

```
bash Copy code

twine upload dist/*
```

- Outcome: Your package will be uploaded to PyPI and can be installed by others using `pip install your-package-name`.



Setup a Python Package

3. Create a Wheel (Python Distribution)

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Python Wheel

1. What is a Wheel?

- **Definition:** A **Wheel** is a built distribution format for Python packages. It is the standard format for distributing Python libraries and applications and is designed to be faster and simpler to install compared to source distributions (`.tar.gz`).
- **File Extension:** `.whl`

2. Advantages of Using Wheels

- **Faster Installation:** Wheels are pre-compiled, which speeds up the installation process because they don't require building or compiling code on the target machine.
- **Compatibility:** Wheels ensure compatibility across different environments, as they bundle all the necessary files and metadata.
- **Ease of Use:** Installing a wheel is as simple as running `pip install package_name.whl`.

Python Package Structure for Wheel Distribution

1. Package Structure Overview

plaintext

Copy code

```
mypackage/
├── mypackage/
│   ├── __init__.py      # Initializes the package
│   ├── module1.py       # Example module 1
│   └── module2.py       # Example module 2
├── tests/
│   ├── test_module1.py  # Unit tests for module 1
│   └── test_module2.py  # Unit tests for module 2
├── dist/                # Distribution files, including .whl
│   └── mypackage-0.1.0-py3-none-any.whl # The Wheel file
├── build/               # Build artifacts created by setup.py
├── mypackage.egg-info/  # Metadata about the package
├── setup.py             # Script for packaging and distribution
├── README.md            # Package description and usage
└── LICENSE              # License information
```

Key Components

- `mypackage/`` Directory: Contains the actual Python package with modules.
- `__init__.py``: Initializes the package and can define what is exposed to users.
- `tests/`` Directory: Contains unit tests for the package.
- `dist/`` Directory: Stores the distribution files, including the `.whl`` file after building.
- `build/`` Directory: Contains temporary build artifacts created during the package build process.
- `mypackage.egg-info/`` Directory: Contains metadata about the package, including version, dependencies, and more.
- `setup.py``: Script that defines the package details and is used to build the distribution files.
- `README.md``: Provides a detailed description of the package, including installation and usage instructions.
- `LICENSE``: Specifies the licensing terms under which the package is distributed.

Building the Package and Creating a Wheel

1. Building the Package

- Command:

```
bash
python setup.py sdist bdist_wheel
```

- Explanation:

- ``sdist``: Creates a source distribution.
- ``bdist_wheel``: Creates a binary distribution (wheel).

2. Resulting Files

- The command generates the following files in the ``dist/`` directory:
 - ``mypackage-0.1.0.tar.gz``: Source distribution.
 - ``mypackage-0.1.0-py3-none-any.whl``: Wheel file.

1. Package Structure Overview

```
plaintext
mypackage/
├── mypackage/
│   ├── __init__.py    # Initializes the package
│   ├── module1.py     # Example module 1
│   └── module2.py     # Example module 2
├── tests/
│   ├── test_module1.py # Unit tests for module 1
│   └── test_module2.py # Unit tests for module 2
├── dist/              # Distribution files, including .whl
│   └── mypackage-0.1.0-py3-none-any.whl # The Wheel file
├── build/             # Build artifacts created by setup.py
├── mypackage.egg-info/ # Metadata about the package
├── setup.py           # Script for packaging and distribution
├── README.md          # Package description and usage
└── LICENSE            # License information
```

Python Package Distribution Files

1. Generated Files After Building the Package

After running the build command (`python setup.py sdist bdist_wheel`), two key files are generated in the `dist/` directory:

- `mypackage-0.1.0.tar.gz`
 - **Type:** Source Distribution
 - **Description:** This is a compressed archive containing the source code of the package. It includes the Python files, `setup.py`, and any other resources needed to build the package.
 - **Purpose:** Useful for users who want to build the package from source, possibly in environments where pre-built wheels aren't available or applicable.
- `mypackage-0.1.0-py3-none-any.whl`
 - **Type:** Wheel File
 - **Description:** This is a built distribution format, commonly referred to as a "wheel". It is a binary package format that contains everything needed to install the package quickly.
 - **Purpose:** Provides a fast and efficient installation method because the package is already built and does not require compiling on the user's system. The `py3` indicates compatibility with Python 3, and `none-any` signifies that this wheel is universal, meaning it can be installed on any platform without modification.

Using Wheels to Install Python Package Locally

2. Usage

- Installing the Source Distribution:

```
bash
```

[Copy code](#)

```
pip install mypackage-0.1.0.tar.gz
```

- Installing the Wheel File:

```
bash
```

[Copy code](#)

```
pip install mypackage-0.1.0-py3-none-any.whl
```

- Installing Directly from the Source Directory:

```
bash
```

[Copy code](#)

```
pip install .
```

- **Explanation:** The command `pip install .` installs the package directly from the current directory where `setup.py` is located. This is useful during development or when testing a package before distribution.

3. Best Practices

- **Distribute Both:** Offer both the source distribution (`.tar.gz`) and the wheel file (`.whl`) to give users flexibility based on their environment.
- **Use Wheels When Possible:** Recommend the wheel file for faster installations, especially for packages with complex dependencies.

Python Package Structure for Wheel Distribution

1. Package Structure Overview

```
plaintext Copy code

mypackage/
├── mypackage/
│   ├── __init__.py      # Initializes the package
│   ├── module1.py       # Example module 1
│   └── module2.py       # Example module 2
├── tests/
│   ├── test_module1.py  # Unit tests for module 1
│   └── test_module2.py  # Unit tests for module 2
├── dist/                # Distribution files, including .whl
│   └── mypackage-0.1.0-py3-none-any.whl # The Wheel file
├── build/               # Build artifacts created by setup.py
├── mypackage.egg-info/  # Metadata about the package
├── setup.py             # Script for packaging and distribution
├── README.md            # Package description and usage
└── LICENSE              # License information
```

2. Key Components

- `mypackage/`` Directory: Contains the actual Python package with modules.
- `__init__.py``: Initializes the package and can define what is exposed to users.
- `tests/`` Directory: Contains unit tests for the package.
- `dist/`` Directory: Stores the distribution files, including the `.whl`` file after building.
- `build/`` Directory: Contains temporary build artifacts created during the package build process.
- `mypackage.egg-info/`` Directory: Contains metadata about the package, including version, dependencies, and more.
- `setup.py``: Script that defines the package details and is used to build the distribution files.
- `README.md``: Provides a detailed description of the package, including installation and usage instructions.
- `LICENSE``: Specifies the licensing terms under which the package is distributed.

Python Wheel

5. Uploading a Wheel to PyPI

- Steps:

1. Build the Wheel:

```
bash  
  
python setup.py bdist_wheel
```

 Copy code

2. Upload to PyPI using Twine:

```
bash  
  
twine upload dist/*.whl
```

 Copy code

- Explanation:

- This process uploads the wheel to PyPI, making it available for others to install.

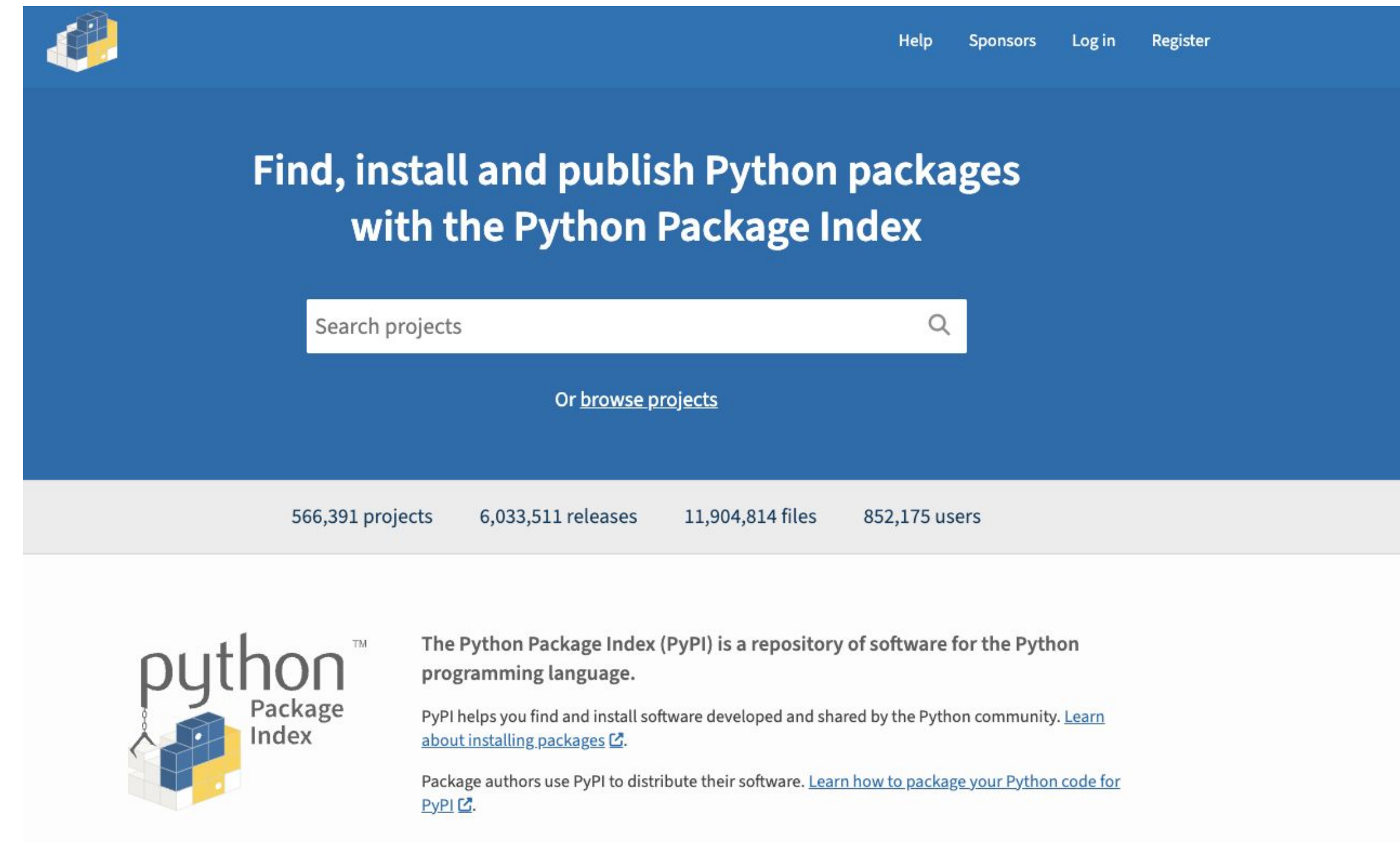
Relationship Between Distributions and PyPI

1. What is PyPI?

- **Definition:** The Python Package Index (PyPI) is the official repository where Python packages are uploaded and shared with the Python community.
- **Purpose:** Enables easy distribution, installation, and management of Python packages via `pip`.

2. Role of Distributions on PyPI

- **Source Distribution (`.tar.gz`)**
 - **Purpose:** Allows users to download and build the package from source.
 - **Usage on PyPI:** Uploaded to PyPI for environments that may require or prefer building from source, such as when specific platform optimizations or custom build steps are needed.
- **Wheel File (`.whl`)**
 - **Purpose:** Provides a pre-built, binary distribution of the package.
 - **Usage on PyPI:** Uploaded to PyPI for easy and fast installation across various environments without the need for compilation, making it the preferred distribution format for most users.



Uploading the Package to PyPI

1. Install `twine`

- Command:

```
bash
pip install twine
```

Copy code

2. Upload to PyPI

- Command:

```
bash
twine upload dist/*
```

Copy code

- Explanation:

- Uploads the package files from the `dist/` directory to PyPI.
- You'll need your PyPI credentials.

3. Test on TestPyPI (Optional)

- Command:

```
bash
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Copy code

- Purpose: Test the upload process before uploading to the official PyPI.

Good Practice Define a Shell Script

```
upload_package() {
  echo "Removing old distribution..."
  rm -rf dist/*
  sleep 2 # Waits for 2 seconds

  echo "Building the distribution..."
  python setup.py sdist bdist_wheel
  sleep 2 # Waits for 2 seconds

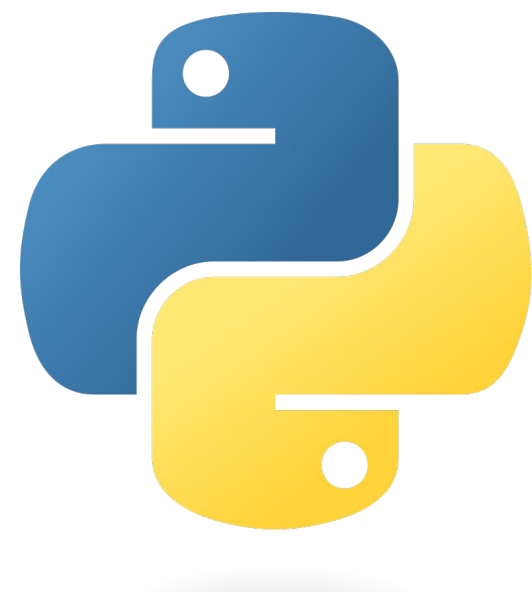
  echo "Uploading the distribution..."
  pip install twine
  sleep 1
  twine upload dist/*.whl dist/*.tar.gz --verbose
}
```

- Availability: Once uploaded, the package becomes available to anyone using `pip`:

```
bash
pip install package_name
```

Copy code

- Example: Users can install your package directly from PyPI with a simple command.



Setup a Python Package

2. GitHub Repository

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

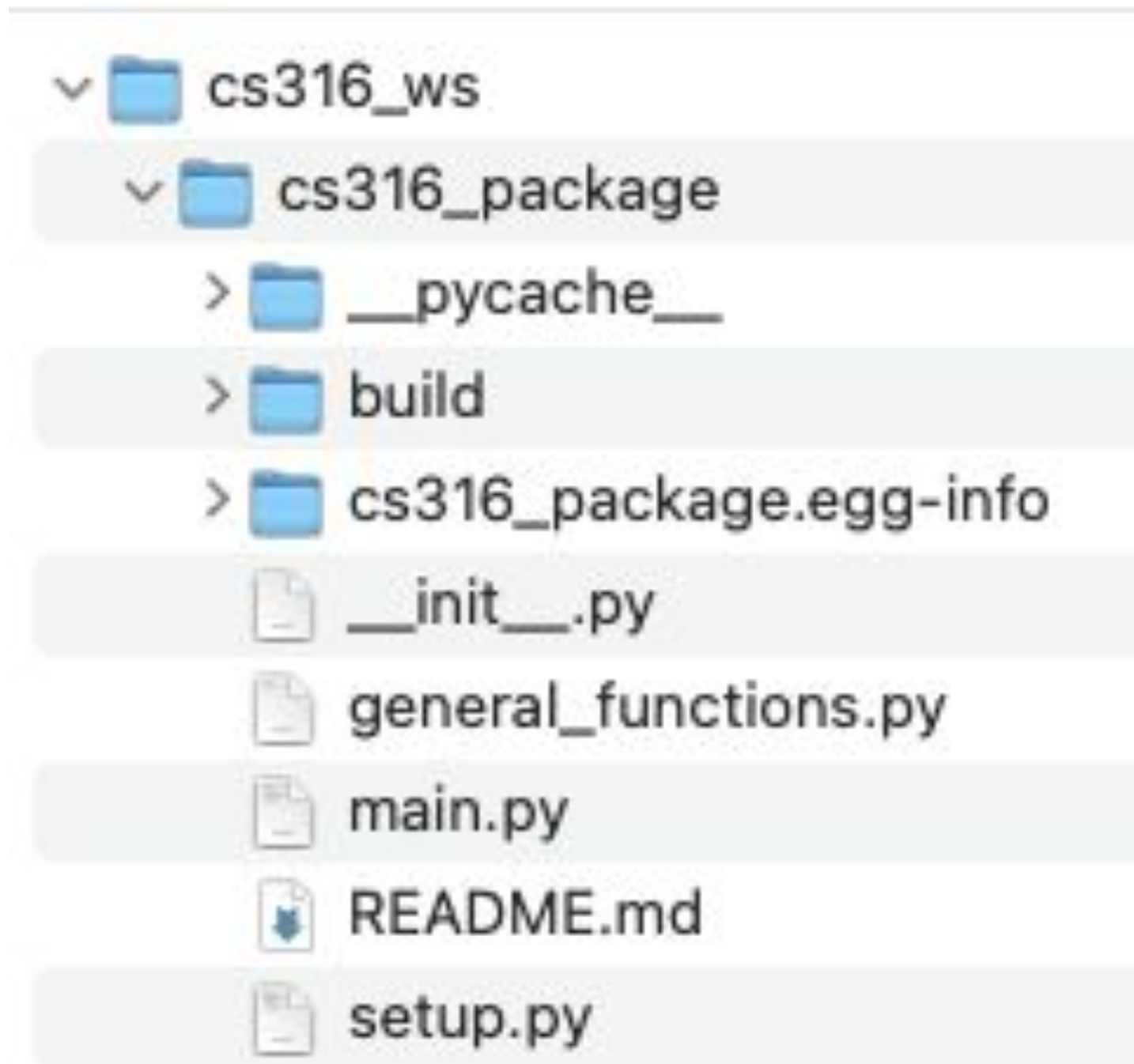
Setting Up a GitHub Account and Uploading a Python Package

1. Creating a GitHub Account

- Visit github.com.
- Click on "Sign up" and fill in your details.
- Verify your email to activate the account.
- Set up a new repository after logging in.

2. Setting Up a Python Package

- Directory Structure:
 - Root directory: ``cs316_ws``
 - Package directory: ``cs316_package/``
 - Essential files:
 - ``__init__.py``: Marks the directory as a Python package.
 - ``general_functions.py``: Contains general utility functions.
 - ``main.py``: Main execution file.
 - ``setup.py``: Used for package installation.
 - ``README.md``: Project description.



Setting Up a GitHub Account and Uploading a Python Package

3. Uploading Package to GitHub

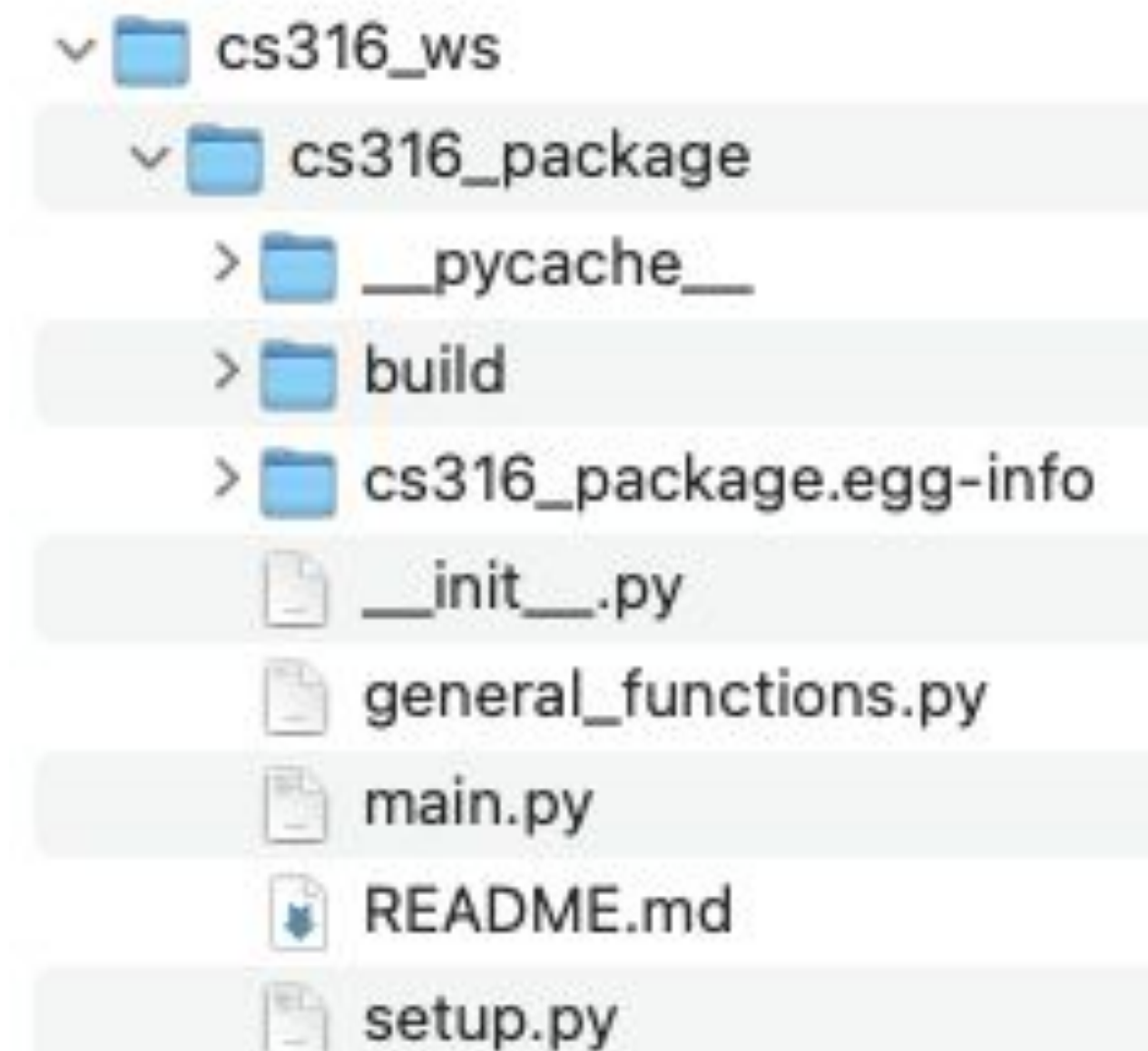
- Initialize Local Repository:
 - ``git init`` inside ``cs316_ws/``.
- Add Files:
 - ``git add .`` to stage all files.
- Commit Changes:
 - ``git commit -m "Initial commit"``
- Push to GitHub:

- Link repository:

```
bash Copy code  
  
git remote add origin https://github.com/yourusername/cs316_package.git
```

- Push changes:

```
bash Copy code  
  
git push -u origin master
```



Setting Up a GitHub Account and Uploading a Python Package

3. Uploading Package to GitHub

- Initialize Local Repository:
 - ``git init`` inside ``cs316_ws/``.
- Add Files:
 - ``git add .`` to stage all files.
- Commit Changes:
 - ``git commit -m "Initial commit"``
- Push to GitHub:
 - Link repository:

bash

Copy code

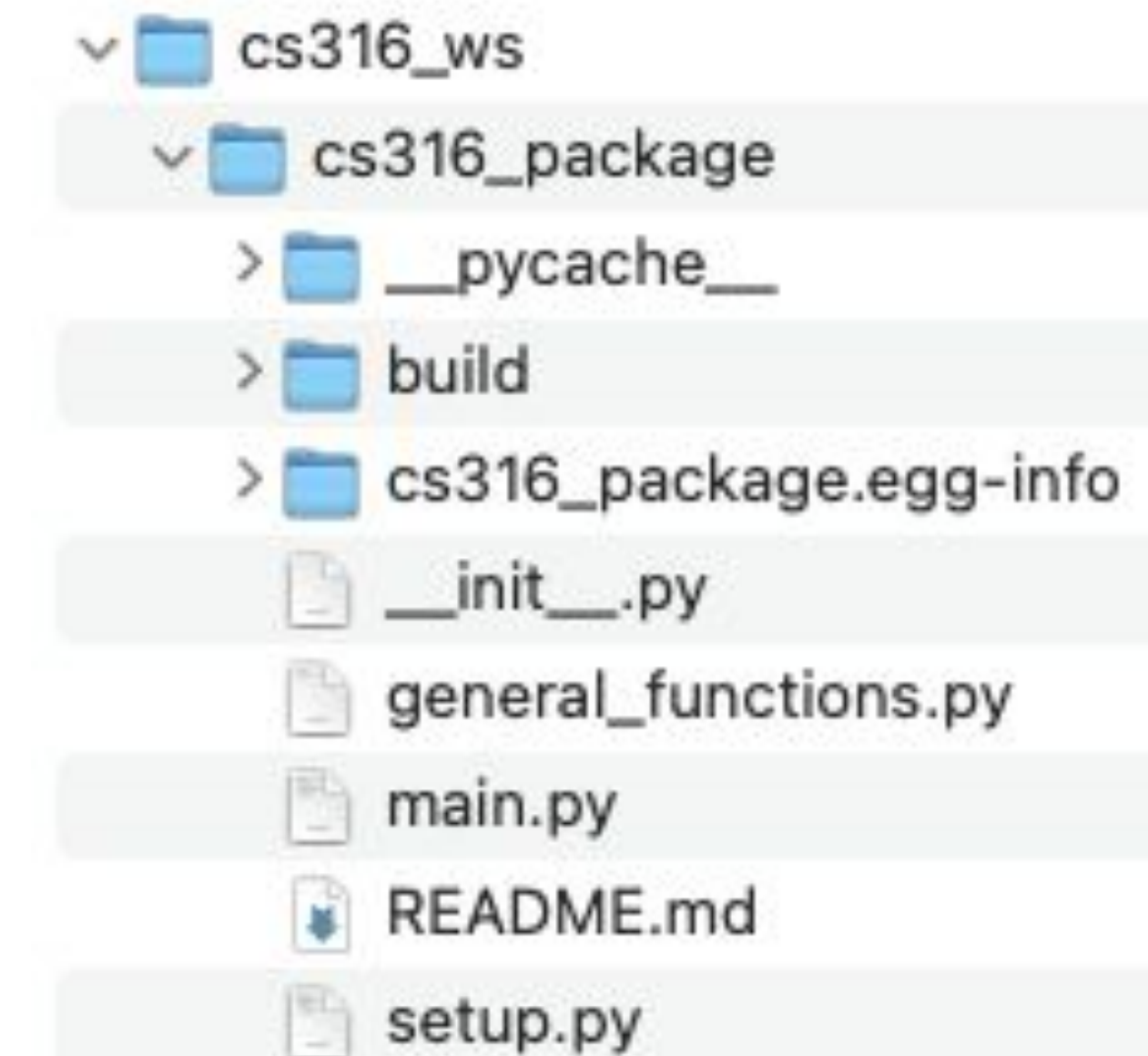
```
git remote add origin https://github.com/yourusername/cs316_package.git
```

- Push changes:

bash

Copy code

```
git push -u origin master
```



4. Verifying on GitHub

- Check the repository on GitHub to ensure all files are uploaded.
- Update ``README.md`` with project details.

CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2
Python for Data Science

LECTURE 3
Python Programming Language
Basic Syntax

Prof. Anis Koubaa

SEP 2024

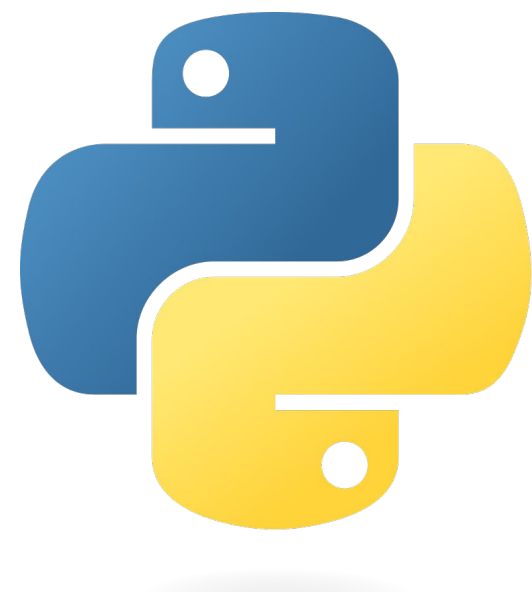
CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2
Python for Data Science

LECTURE 3 – Demo
Python Programming Language
Basic Syntax

Prof. Anis Koubaa

SEP 2024



Python Basic Syntax

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Python Basic Syntax

1. Introduction to Python Syntax

- **Overview:** Python syntax is designed to be simple and easy to read, making code more intuitive and accessible.
- **Key Features:**
 - **Whitespace:** Indentation is used to define code blocks (e.g., loops, functions).
 - **Minimal Syntax:** No need for semicolons or curly braces.
 - **Readability:** Emphasizes clear, human-readable code.

2. Code Example

```
python Copy code  
  
# Define a function  
def greet(name):  
    print(f"Hello, {name}!")  
  
# Call the function  
greet("Alice")
```

- **Explanation:**
 - **Indentation:** The function body is indented to indicate it belongs to the `greet` function.
 - **Function Call:** The function is called with an argument (`"Alice"`), and the output is `Hello, Alice!`.

Basic Syntax Elements

1. Indentation

- **Description:** Python uses indentation to define blocks of code, unlike other languages that use braces `{}`.
- **Standard Practice:** Use four spaces per indentation level.
- **Example:**

python

 Copy code

```
if x > 0:  
    print("Positive number")
```

- **Explanation:** The indented line belongs to the `if` statement block.

Basic Syntax Elements

2. Comments

- **Description:** Use the hash `#` symbol to write comments. Python ignores anything after `#` on a line.
- **Usage:** Comments are used for notes, documentation, and explanations within the code.
- **Example:**

```
python Copy code  
  
# This is a comment  
print("Hello, world!") # This prints a message
```

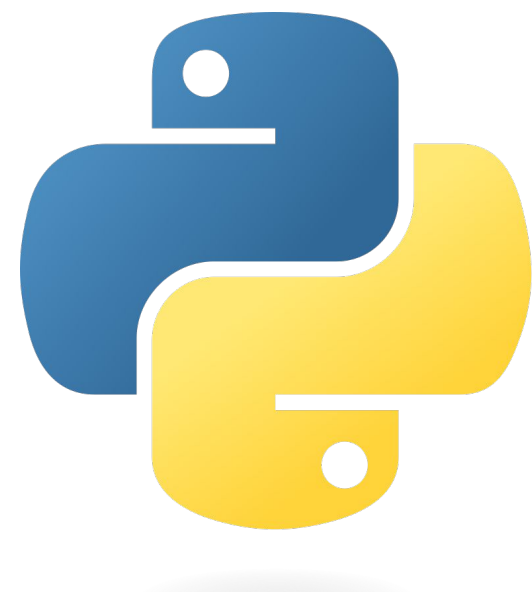
Basic Syntax Elements

3. Case Sensitivity

- **Description:** Python is case-sensitive, meaning that variables like ``myVariable``, ``MyVariable``, and ``myvariable`` are distinct.
- **Example:**

```
python Copy code  
  
myVariable = 10  
MyVariable = 20  
print(myVariable) # Outputs: 10  
print(MyVariable) # Outputs: 20
```

- **Explanation:** Different capitalization creates different variables.



Python Variables and Data Types

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Variables and Data Types

1. Declaring Variables

- **Description:** Python automatically allocates memory when you assign a value to a variable—no need for explicit declaration.
- **Example:**

python

 Copy code

```
x = 10      # Integer
y = 3.14    # Float
name = "Alice" # String
is_valid = True # Boolean
```

- **Explanation:** The variables `x`, `y`, `name`, and `is_valid` are automatically created with their respective data types based on the assigned values.

Variables and Data Types

2. Basic Data Types

- **Integers:** Whole numbers, positive or negative, without decimals.
 - Example: `x = 10`
- **Floats:** Decimal numbers, used for more precise calculations.
 - Example: `y = 3.14`
- **Strings:** Sequences of characters, enclosed in single or double quotes.
 - Example: `name = "Alice"`
- **Booleans:** Represent logical values, either `True` or `False`.
 - Example: `is_valid = True`

Non-Basic Python Data Types

Data Type	Description	Example	Characteristics
List	Ordered, mutable collection of items	<code>`my_list = [1, 2, 3]`</code>	Dynamic size, supports mixed data types, accessed by index
Tuple	Ordered, immutable collection of items	<code>`my_tuple = (1, 2, 3)`</code>	Fixed size, supports mixed data types, accessed by index
Set	Unordered collection of unique items	<code>`my_set = {1, 2, 3}`</code>	No duplicates, unordered, supports set operations
Dictionary	Unordered collection of key-value pairs	<code>`my_dict = {'a': 1, 'b': 2}`</code>	Keys must be unique, values accessed by keys
Complex	Numbers with real and imaginary parts	<code>`z = 3 + 4j`</code>	Precision depends on float representation for real/imaginary parts
Bytes	Immutable sequence of bytes	<code>`my_bytes = b'hello'`</code>	Used for binary data, immutable
Bytearray	Mutable sequence of bytes	<code>`my_bytearray = bytearray(b'hello')`</code>	Mutable version of <code>`bytes`</code> , can be modified in place
NoneType	Represents the absence of a value	<code>`value = None`</code>	Singleton, only one instance (<code>`None`</code>) exists

Finding the Type of a Variable with `type()`

1. Using `type()` Function

- Purpose: `type()` is a built-in Python function used to determine the type of a variable.

2. Example Code

```
python Copy code

# Define variables of different types
x = 10          # Integer
y = 3.14        # Float
name = "Alice"  # String
is_valid = True # Boolean

# Use type() to find the type of each variable
print(type(x))    # Output: <class 'int'>
print(type(y))    # Output: <class 'float'>
print(type(name)) # Output: <class 'str'>
print(type(is_valid)) # Output: <class 'bool'>
```

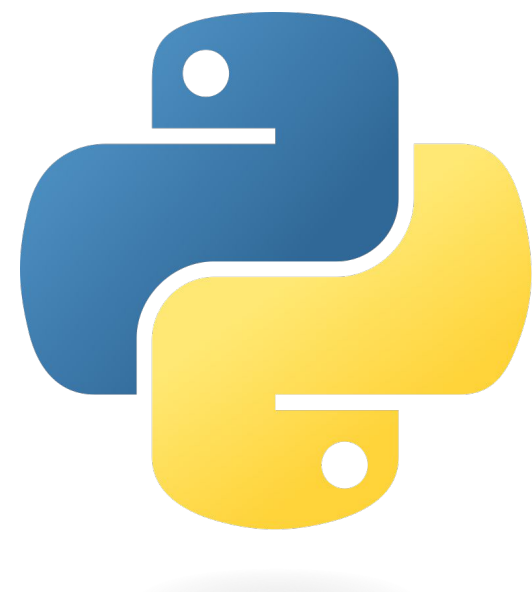
3. Explanation

- The `type()` function returns the type of the variable passed as an argument.
- The output shows the class of the variable (`'int'`, `'float'`, `'str'`, `'bool'`), which corresponds to its data type.

Finding the Type of a Variable with type()

```
1 #String variable
2 print ("--- STRING VARIABLES --- ")
3 name = "Mohamed"
4 print ("student name: ", name)
5 print ("the type of the variable name is: ", type(name))
6
7
8 #Boolean variable
9 print ("--- BOOLEAN VARIABLES --- ")
10 flag = True
11 print ("Flag: ", flag)
12 print ("the type of the variable flag is: ", type(flag))
13
14 #Hexadecimal Number
15 print ("--- HEXADECEMAL VARIABLES --- ")
16 w = 0xfa2
17 v=hex(547)
18 print ("w: ", w)
19 print ("v: ", v)
20 print ("the type of the variable w is: ", type(w))
21 print ("the type of the variable v is: ", type(v))
```

```
--- STRING VARIABLES ---
student name: Mohamed
the type of the variable name is: <class 'str'>
--- BOOLEAN VARIABLES ---
Flag: True
the type of the variable flag is: <class 'bool'>
--- HEXADECEMAL VARIABLES ---
w: 4002
v: 0x223
the type of the variable w is: <class 'int'>
the type of the variable v is: <class 'str'>
```



Type Conversion in Python

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Implicit Type Conversion

1. Implicit Type Conversion

- **Description:** Python automatically converts one data type to another when necessary.
- **Example:**

python

 Copy code

```
x = 5      # Integer
y = 3.2    # Float
z = x + y  # Implicitly converts x to float
print(z)   # Output: 8.2 (float)
```

- **Explanation:** In the above example, the integer `x` is implicitly converted to a float during the addition.

Explicit Type Conversion

2. Explicit Type Conversion (Type Casting)

- **Description:** Converting one data type to another manually using built-in functions.
- **Common Functions:**
 - `int(x)`: Converts `x` to an integer.
 - `float(x)`: Converts `x` to a float.
 - `str(x)`: Converts `x` to a string.
 - `bool(x)`: Converts `x` to a boolean.

4. Considerations

- **Loss of Data:** Converting from a float to an integer truncates the decimal part.
- **Errors:** Converting incompatible types (e.g., `int("abc")`) will raise a `ValueError`.

3. Examples of Explicit Type Conversion

- Integer to String:

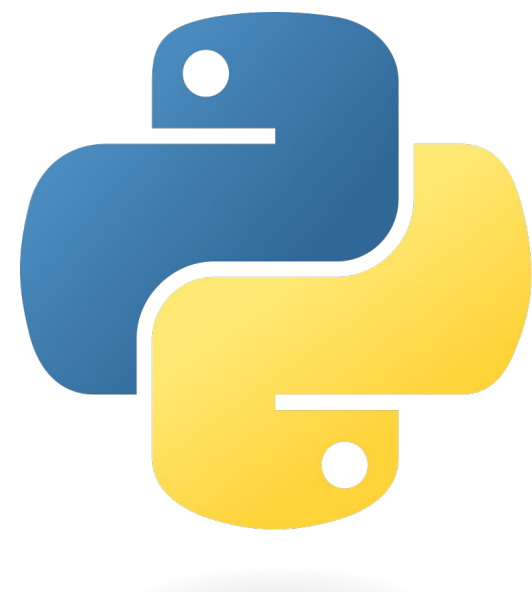
```
python
num = 10
num_str = str(num)
print(num_str) # Output: "10" (string)
```

- String to Integer:

```
python
s = "123"
num = int(s)
print(num) # Output: 123 (integer)
```

- Float to Integer:

```
python
pi = 3.14
int_pi = int(pi)
print(int_pi) # Output: 3 (integer, truncated)
```



Python Basic Operators

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Basic Operators in Python

Operator Type	Operator	Description	Example	Result
Arithmetic Operators	<code>`+`</code>	Addition	<code>`5 + 3`</code>	<code>`8`</code>
	<code>`-`</code>	Subtraction	<code>`5 - 3`</code>	<code>`2`</code>
	<code>`*`</code>	Multiplication	<code>`5 * 3`</code>	<code>`15`</code>
	<code>`/`</code>	Division (returns a float)	<code>`5 / 2`</code>	<code>`2.5`</code>
	<code>`%`</code>	Modulus (remainder of division)	<code>`5 % 2`</code>	<code>`1`</code>
	<code>`**`</code>	Exponentiation (power)	<code>`5 ** 3`</code>	<code>`125`</code>
Comparison Operators	<code>`==`</code>	Equal to	<code>`5 == 3`</code>	<code>`False`</code>
	<code>`!=`</code>	Not equal to	<code>`5 != 3`</code>	<code>`True`</code>
	<code>`>`</code>	Greater than	<code>`5 > 3`</code>	<code>`True`</code>
	<code>`<`</code>	Less than	<code>`5 < 3`</code>	<code>`False`</code>
	<code>`>=`</code>	Greater than or equal to	<code>`5 >= 3`</code>	<code>`True`</code>
	<code>`<=`</code>	Less than or equal to	<code>`5 <= 3`</code>	<code>`False`</code>

Basic Operators in Python

```
1 a = 2
2 b = 3
3
4 add = a+b
5 sub = a-b
6 mult = a*b
7 div = a/b
8
9 mod = a%b
10 e = a ** b
11
12 print ("a = ",a)
13 print ("b = ",b)
14 print ("add = a+b = ",add)
15 print ("sub = a-b = ",sub)
16 print ("mult = a*b = ",mult)
17 print ("div = a/b = ",div)
18 print ("mod = a%b = ",mod)
19 print ("e = a**b = ",e)
20
21
```

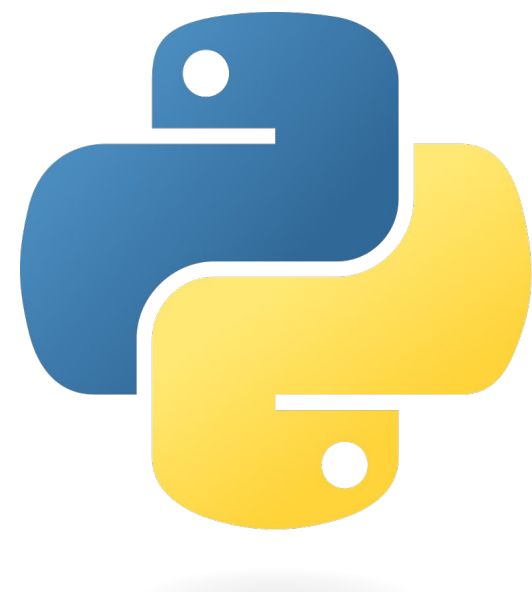
```
⇒ a = 2
b = 3
add = a+b = 5
sub = a-b = -1
mult = a*b = 6
div = a/b = 0.6666666666666666
mod = a%b = 2
e = a**b = 8
```

Typing Constraints

- **Type Compatibility:** Python operators generally require operands to be of compatible types.

For example:

- **Arithmetic Operators:** Operands must typically be of numeric types (`int`, `float`), though some operators like `+` can also concatenate strings.
 - Example: `"Hello" + " World"` results in `"Hello World"`
- **Comparison Operators:** Can compare numbers, strings, and other comparable types, but comparing incompatible types (e.g., `int` and `str`) will raise a `TypeError`.
 - Example: `5 > "3"` will result in a `TypeError`
- **Implicit Type Conversion:** Python may perform implicit type conversion (e.g., `int` to `float`) when necessary, but explicit conversion using functions like `int()`, `float()`, or `str()` is often required to avoid errors.



Simple Input/Output in Python

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Simple Input/Output in Python

1. Input: `input()` Function

- **Purpose:** Used to get user input from the keyboard.
- **Syntax:** `variable = input("Prompt message")`
- **Example:**

python

 Copy code

```
name = input("Enter your name: ")
```

- **Explanation:** The input provided by the user is stored as a string in the variable `name`.

Simple Input/Output in Python

2. Output: `print()` Function

- Purpose: Used to display output to the screen.
- Syntax: `print(value1, value2, ..., sep=' ', end='\n')`
- Example:

```
python Copy code  
  
print("Hello, World!")  
print("Welcome,", name)
```

- Explanation:
 - The first `print()` displays the message `"Hello, World!"`.
 - The second `print()` uses the variable `name` to display a personalized welcome message.

Simple Input/Output in Python

3. Combined Example

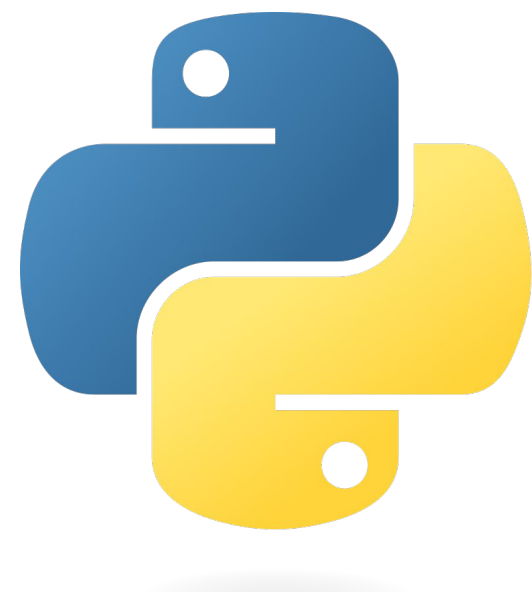
python

 Copy code

```
# Getting user input
age = input("Enter your age: ")

# Displaying the input
print("You are", age, "years old.")
```

- **Explanation:** This code prompts the user for their age and then displays it back to them in a complete sentence.



Formatted Output in Python

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Formatted Output in Python with print()

1. Basic `print()`

- **Description:** Displays one or more values separated by a space (default).
- **Example:**

”

python

Copy code

```
print("Hello", "World!")
```

- **Output:**
`Hello World`

2. Using `sep` Parameter

- **Description:** Customizes the separator between multiple values.
- **Example:**

python

Copy code

```
print("Hello", "World", sep="-")
```

- **Output:**
`Hello-World`

Formatted Output in Python with print()

3. Using `end` Parameter

- **Description:** Customizes what is printed at the end of the output (default is a newline ``\n``).
- **Example:**

python

 Copy code

```
print("Hello", end=", ")  
print("World!")
```

- **Output:**
`Hello, World!`

Formatted Output in Python with print()

4. String Formatting with `%` Operator

- Description: Uses placeholders (`%s`, `%d`, `%f`) to embed values in a string.
- Example:

```
python Copy code  
  
name = "Alice"  
age = 30  
print("My name is %s and I am %d years old." % (name, age))
```

- Output:
`My name is Alice and I am 30 years old.`

Formatted Output in Python with print()

5. `str.format()` Method

- **Description:** A more versatile way to format strings, using curly braces `{}` as placeholders.
- **Example:**

python

 Copy code

```
name = "Bob"  
score = 95.5  
print("Name: {}, Score: {:.1f}".format(name, score))
```

- **Output:**
``Name: Bob, Score: 95.5``

Formatted Output in Python with print()

6. f-Strings (Python 3.6+)

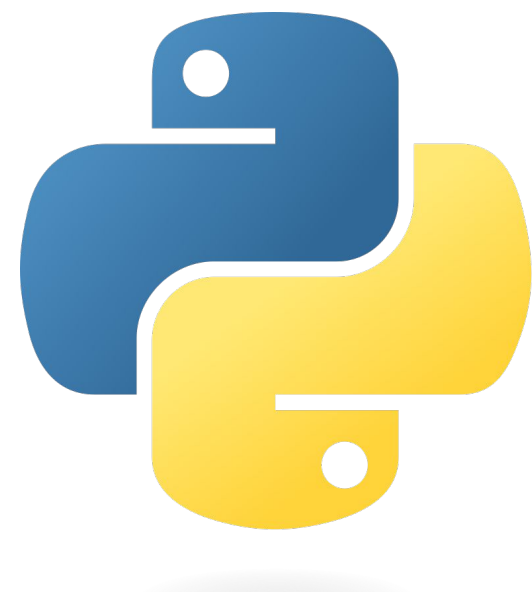
- **Description:** An easy and efficient way to embed expressions inside string literals using `{}` with an `f` prefix.

- **Example:**

```
python Copy code  
  
name = "Charlie"  
height = 1.75  
print(f"{name} is {height} meters tall.")
```

- **Output:**

```
Charlie is 1.75 meters tall.
```



Math Functions in Python

Prof. Anis Koubaa



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Common Mathematical Functions in Python

Function	Description	Example	Result
<code>`abs(x)`</code>	Returns the absolute value of <code>`x`</code>	<code>`abs(-5)`</code>	<code>`5`</code>
<code>`round(x, n)`</code>	Rounds <code>`x`</code> to <code>`n`</code> decimal places	<code>`round(3.14159, 2)`</code>	<code>`3.14`</code>
<code>`max(x1, x2, ...)`</code>	Returns the largest value among arguments	<code>`max(3, 7, 1)`</code>	<code>`7`</code>
<code>`min(x1, x2, ...)`</code>	Returns the smallest value among arguments	<code>`min(3, 7, 1)`</code>	<code>`1`</code>
<code>`pow(x, y)`</code>	Returns <code>`x`</code> raised to the power of <code>`y`</code>	<code>`pow(2, 3)`</code>	<code>`8`</code>
<code>`sum(iterable)`</code>	Returns the sum of all items in an iterable	<code>`sum([1, 2, 3, 4])`</code>	<code>`10`</code>
<code>`math.sqrt(x)`</code>	Returns the square root of <code>`x`</code>	<code>`math.sqrt(16)`</code>	<code>`4.0`</code>
<code>`math.exp(x)`</code>	Returns <code>`e`</code> raised to the power of <code>`x`</code>	<code>`math.exp(1)`</code>	<code>`2.718281828459045`</code>
<code>`math.log(x, base)`</code>	Returns the logarithm of <code>`x`</code> to the specified base (default is <code>`e`</code>)	<code>`math.log(8, 2)`</code>	<code>`3.0`</code>
<code>`math.sin(x)`</code>	Returns the sine of <code>`x`</code> (in radians)	<code>`math.sin(math.pi / 2)`</code>	<code>`1.0`</code>
<code>`math.cos(x)`</code>	Returns the cosine of <code>`x`</code> (in radians)	<code>`math.cos(0)`</code>	<code>`1.0`</code>

Basic Operators in Python

Operator Type	Operator	Description	Example	Result
Arithmetic Operators	`+`	Addition	`5 + 3`	`8`
	`-`	Subtraction	`5 - 3`	`2`
	`*`	Multiplication	`5 * 3`	`15`
	`/`	Division (returns a float)	`5 / 2`	`2.5`
	`%`	Modulus (remainder of division)	`5 % 2`	`1`
	`**`	Exponentiation (power)	`5 ** 3`	`125`
Comparison Operators	`==`	Equal to	`5 == 3`	`False`
	`!=`	Not equal to	`5 != 3`	`True`
	`>`	Greater than	`5 > 3`	`True`
	`<`	Less than	`5 < 3`	`False`
	`>=`	Greater than or equal to	`5 >= 3`	`True`
	`<=`	Less than or equal to	`5 <= 3`	`False`

Common Mathematical Functions in Python

```
1 x = -2.56
2 y = 3.54
3 n = 3
4
5 from math import *
6
7
8 print ('the absolute value of x is : ', abs(x)) #we can use abs or fabs, math.fabs
9 print ('-----')
10 print ('the ceiling of x is : ', ceil(x))
11 print ('the floor of x is : ', floor(x))
12 print ('the ceiling of y is : ', ceil(y))
13 print ('the floor of y is : ', floor(y))
14 print ('-----')
15 print ('the log of y is : ', log(y)) #log only applies for positive values
16 print ('the log10 of x is : ', log10(y)) #log10 only applies for positive values
17 print ('the exponential of y is : ', exp(y))
18 print ('the square root of y is : ', sqrt(y))
19 print ('the power of x by y is : ', pow(x,n))
```

↳ the absolute value of x is : 2.56

the ceiling of x is : -2
the floor of x is : -3
the ceiling of y is : 4
the floor of y is : 3

the log of y is : 1.264126727145683
the log10 of x is : 0.5490032620257879
the exponential of y is : 34.46691919085739
the square root of y is : 1.8814887722226779
the power of x by y is : -16.777216000000003

CS316

INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2

Python for Data Science

LECTURE 4

Functional Programming

First-Class Objects, Higher-Order Functions, Iterables
Lambda Expressions – Filter/Map/Reduce

Prof. Anis Koubaa

SEP 2024

Why Functional Programming Matters in Data Science

1. What is Functional Programming?

- **Simple Definition:** A way of writing software programs that treats computations as evaluations of mathematical functions and avoids using mutable data.

2. Key Benefits for Data Science

- **Safe Data Handling:** Keeps data unchanged (immutable), which prevents errors during data analysis where original data must remain unaltered.
- **Easy to Use Tools:** Utilizes powerful, easy-to-use functions like `map`, `filter`, and `reduce` that help in handling and analyzing large data sets efficiently.

3. Streamlining Data Workflows

- **Building Blocks:** Encourages the use of small, reusable pieces of code that can be easily tested and combined to handle complex data tasks.
- **Clear Workflows:** Helps in putting together data transformation steps in a clear and logical way, making the analysis easier to understand and modify.

- **Cleaning Data:** Quickly removing unwanted data entries:

```
python Copy code

# Example: Removing missing values (None) from a list of data
data = [1, None, 3, 4, None]
clean_data = list(filter(None, data)) # [1, 3, 4]
```

- **Adjusting Data Features:** Applying changes to data elements to prepare for analysis:

```
python Copy code

# Example: Applying a transformation to numerical data
import math
values = [10, 20, 30, 40]
adjusted_values = list(map(math.log, values)) # [log(10), log(20), log(30), log(40)]
```

Data Immutability

Example 1: Using Immutable Data Types Python's tuple is an immutable data type. When using tuples in a functional style, you ensure that data cannot be changed unintentionally.

python

 Copy code

```
def add_element_immutable(data, element):  
    # Instead of modifying the original data, return a new tuple with the element added  
    return data + (element,)  
  
# Create a tuple, which is an immutable data structure  
original_data = (1, 2, 3)  
new_data = add_element_immutable(original_data, 4)  
  
print("Original Data:", original_data) # Output: (1, 2, 3)  
print("New Data:", new_data) # Output: (1, 2, 3, 4)
```

Data Immutability

Example 2: Immutability in List Operations Python lists are mutable, but you can use them in an immutable way by avoiding methods that alter the list directly.

python

 Copy code

```
def process_data_immutable(data):  
    # Instead of modifying the list directly, return a new list with modifications  
    return [x * 2 for x in data]  
  
# Original list  
original_list = [1, 2, 3]  
# Processed data without altering the original list  
processed_list = process_data_immutable(original_list)  
  
print("Original List:", original_list) # Output: [1, 2, 3]  
print("Processed List:", processed_list) # Output: [2, 4, 6]
```

Functional Programming in Data Science

1. Introduction to Functional Programming

- **Definition:** A programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

2. Core Benefits of Functional Programming in Data Science

- **Immutability:** Reduces side effects and ensures data integrity, which is crucial in data analysis processes where data should not be altered unintentionally.
- **Higher-Order Functions:** Simplifies data manipulation and analysis by using functions like `map`, `filter`, and `reduce` to operate on collections of data.

3. Enhancing Data Pipelines

- **Modularity:** Functional programming promotes writing smaller, reusable, and more modular functions that can be easily tested and integrated into data pipelines.
- **Composability:** Functions can be composed together to build complex data transformation workflows more clearly and succinctly.

- **Cleaning Data:** Quickly removing unwanted data entries:

```
python Copy code  
  
# Example: Removing missing values (None) from a list of data  
data = [1, None, 3, 4, None]  
clean_data = list(filter(None, data)) # [1, 3, 4]
```

- **Adjusting Data Features:** Applying changes to data elements to prepare for analysis:

```
python Copy code  
  
# Example: Applying a transformation to numerical data  
import math  
values = [10, 20, 30, 40]  
adjusted_values = list(map(math.log, values)) # [log(10), log(20), log(30), log(40)]
```

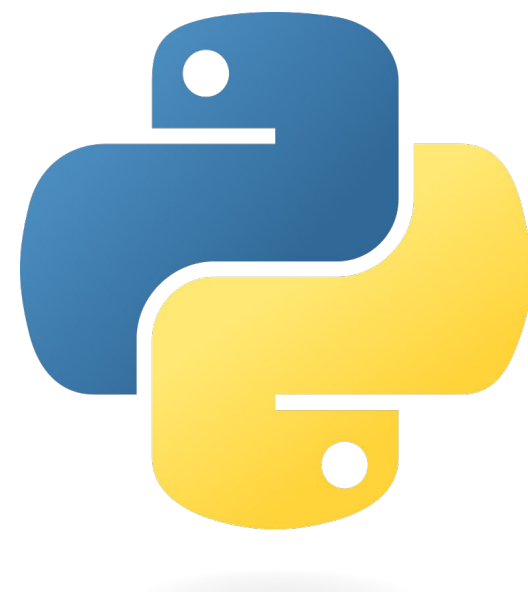
Python Functional Programming

1. Introduction to Functional Programming

- Functional programming is a paradigm that treats computation as the evaluation of mathematical functions.
- Avoids changing-state and mutable data.

2. Key Concepts

- **First-Class Functions:** Functions are treated as first-class citizens, meaning they can be variables, passed as arguments, returned from other functions, and stored in data structures.
- **Higher-Order Functions:** Functions that can take other functions as arguments or return them as results.
- **Pure Functions:** Functions that have no side effects and return a value that depends only on their input.



Pure Functions in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Pure Functions in Python

1. Definition of Pure Functions

- **Core Concept:** A pure function is a function where the output value is determined solely by its input values, without observable side effects.

2. Characteristics of Pure Functions

- **Deterministic:** Always returns the same result given the same input.
- **No Side Effects:** Does not modify any external state (no global variables, no I/O operations).

3. Benefits of Pure Functions

- **Predictability:** Easy to predict outcomes, making debugging and testing simpler.
- **Reusability:** Since they don't depend on external state, they are highly reusable across different parts of a program.
- **Concurrency:** Safe to use in concurrent programming because they don't alter state.

4. Simple Example: Multiplying Numbers

- **Function Code:**

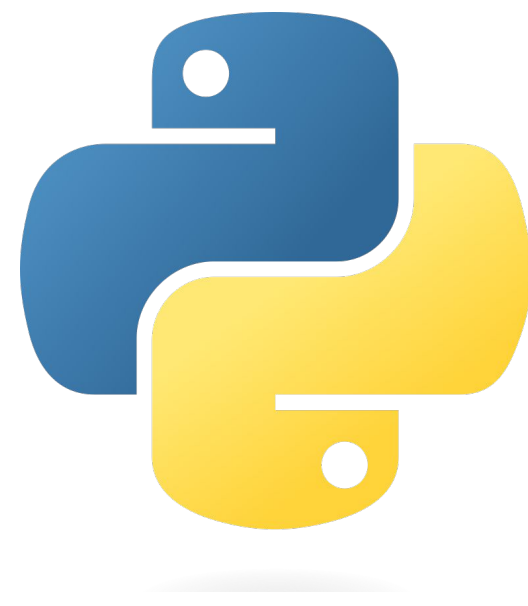
```
python Copy code

def multiply(a, b):
    """
    A pure function that multiplies two numbers.
    The output depends only on the input values a and b,
    and it does not modify any external state.
    """
    return a * b
```

- **Usage:**

```
python Copy code

result = multiply(5, 3)
print("Result of multiplication:", result) # Output: 15
```



Higher-Order Functions in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Higher-Order Functions in Python

1. What are Higher-Order Functions?

- **Definition:** Functions that take other functions as arguments, or return them as results.
- **Purpose:** They enable abstracting over actions, not just values.

2. Benefits of Higher-Order Functions

- **Modularity:** Break down a task into simpler, reusable functions.
- **Reusability:** Use the same function in various contexts with different functions.
- **Flexibility:** More flexible code that can adapt to changing requirements.

5. Conclusion

- Higher-order functions are a powerful feature of Python that allow for writing expressive and concise code.
- They exemplify a key aspect of functional programming by enabling functions to operate on other functions.

3. Simple Example: Doubling Numbers

- **Function Definition:**

```
python Copy code  
  
def double_number(n):  
    return n * 2
```

- **Higher-Order Function:**

```
python Copy code  
  
def process_list(func, numbers):  
    result = []  
    for number in numbers:  
        result.append(func(number))  
    return result
```

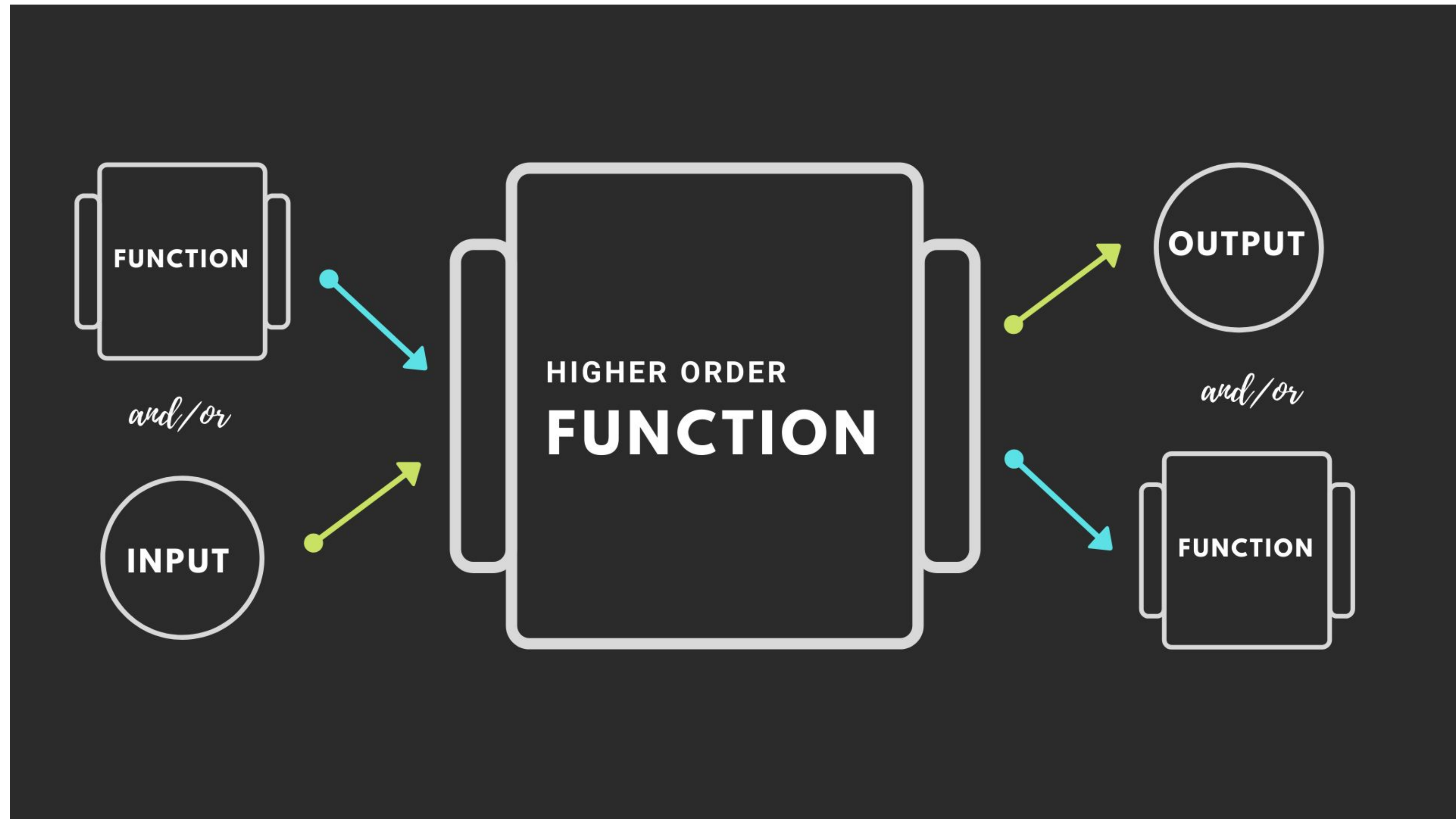
4. Usage Example

- **List of Numbers:** [1, 2, 3, 4, 5]
- **Applying Higher-Order Function:**

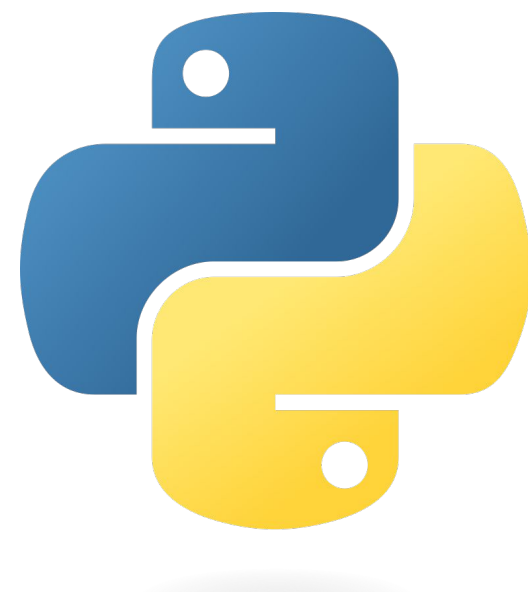
```
python Copy code  
  
doubled_numbers = process_list(double_number, [1, 2, 3, 4, 5])  
print("Doubled Numbers:", doubled_numbers)
```

Output: [2, 4, 6, 8, 10]

Higher-Order Functions in Python



<https://medium.com/analytics-vidhya/higher-order-functions-python-716f508a8f41>



First-Class Objects in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

First-Class Objects in Python

Key Concept:

- **First-Class Objects** are entities that can be:
 - Passed as arguments to functions.
 - Returned from functions.
 - Assigned to variables.
 - Stored in data structures (like lists or dictionaries).

Simple Example:

```
python Copy code  
  
# Assigning a function to a variable  
def greet(name):  
    return f"Hello, {name}!"  
  
say_hello = greet # Function assigned to variable  
  
print(say_hello("Alice")) # Output: Hello, Alice!
```

- In Python, **functions are first-class objects**.
- You can treat functions like any other data type (e.g., integers, strings).

Functions as First-Class Objects

- **First-Class Objects** can be manipulated like any other object, offering flexibility and power in programming.
 - **Functions as Arguments:** Pass a function into another function as an argument.
 - **Functions as Return Values:** Return functions from other functions.
 - **Storing Functions:** Store functions in lists, dictionaries, or other structures.

1. Passing a Function as an Argument:

```
python Copy code

def apply_function(func, value):
    return func(value)

def square(x):
    return x * x

result = apply_function(square, 5) # Function passed as an argument
print(result) # Output: 25
```

Functions as First-Class Objects

- **First-Class Objects** can be manipulated like any other object, offering flexibility and power in programming.
 - **Functions as Arguments:** Pass a function into another function as an argument.
 - **Functions as Return Values:** Return functions from other functions.
 - **Storing Functions:** Store functions in lists, dictionaries, or other structures.

2. Returning a Function from Another Function:

python

 Copy code

```
def get_multiplier(n):  
    return lambda x: x * n  
  
doubler = get_multiplier(2) # Function returned from another function  
print(doubler(5)) # Output: 10
```

Functions as First-Class Objects

- **First-Class Objects** can be manipulated like any other object, offering flexibility and power in programming.
 - **Functions as Arguments:** Pass a function into another function as an argument.
 - **Functions as Return Values:** Return functions from other functions.
 - **Storing Functions:** Store functions in lists, dictionaries, or other structures.

1. Storing Functions in a List:

```
python Copy code

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

# Storing functions in a list
operations = [add, subtract]

result = operations[0](10, 5) # Call the first function (add)
print(result) # Output: 15
```

Functions as First-Class Objects

- **First-Class Objects** can be manipulated like any other object, offering flexibility and power in programming.
 - **Functions as Arguments:** Pass a function into another function as an argument.
 - **Functions as Return Values:** Return functions from other functions.
 - **Storing Functions:** Store functions in lists, dictionaries, or other structures.

2. Storing Functions in a Dictionary:

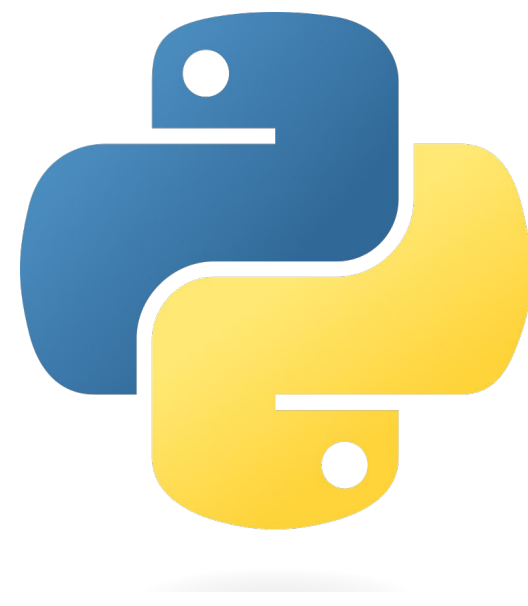
```
python Copy code

def multiply(x, y):
    return x * y

def divide(x, y):
    return x / y

# Storing functions in a dictionary
operations = {
    'multiply': multiply,
    'divide': divide
}

result = operations['multiply'](10, 3) # Call the multiply function
print(result) # Output: 30
```



Python Iterable

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316 INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2 Python for Data Science

LECTURE 1 *Introduction to Python Programming*

Iterables in Python

1. What is an Iterable?

- **Definition:** An iterable in Python is any object capable of returning its members one at a time, permitting it to be iterated over in a loop.
- **Common Examples:** Lists, tuples, dictionaries, strings, and sets.

2. How Iterables Work

- **Iteration Protocol:** Python iterables work by implementing the `__iter__()` method which returns an iterator. This iterator then allows you to go through the elements of the iterable using the `next()` function until no more elements are left.
- **Iteration with Loops:** Typically accessed using loops like `for` or `while`. For example, `for element in iterable:` automatically handles the creation of an iterator and fetching elements.

- **Iterating Over a List:**

python

 Copy code

```
my_list = [1, 2, 3]
for number in my_list:
    print(number)
```

- **Using a Dictionary:**

python

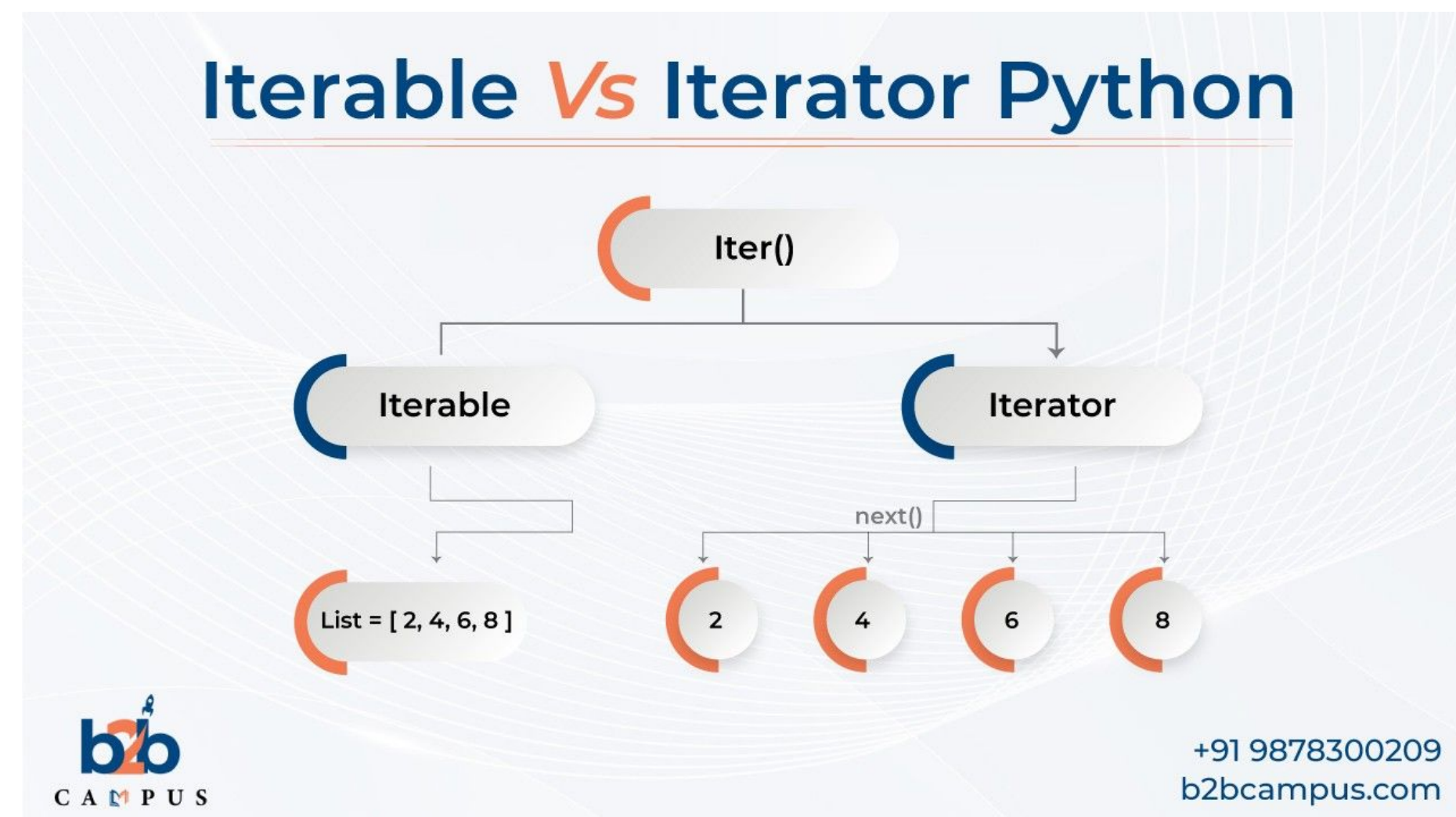
 Copy code

```
my_dict = {'a': 1, 'b': 2}
for key in my_dict:
    print(key, my_dict[key])
```

Iterables in Python

3. Key Concepts

- **Iterator vs. Iterable:** An iterable is an object that can return an iterator, while an iterator is an object that keeps state and produces the next value when you call `next()` on it.
- **Infinite Iterables:** Python supports infinite iterables that, when iterated over, never run out of data. Examples include generators and certain custom classes with a specific `__iter__()` method implementation.



- **Iterating Over a List:**

python

Copy code

```
my_list = [1, 2, 3]
for number in my_list:
    print(number)
```

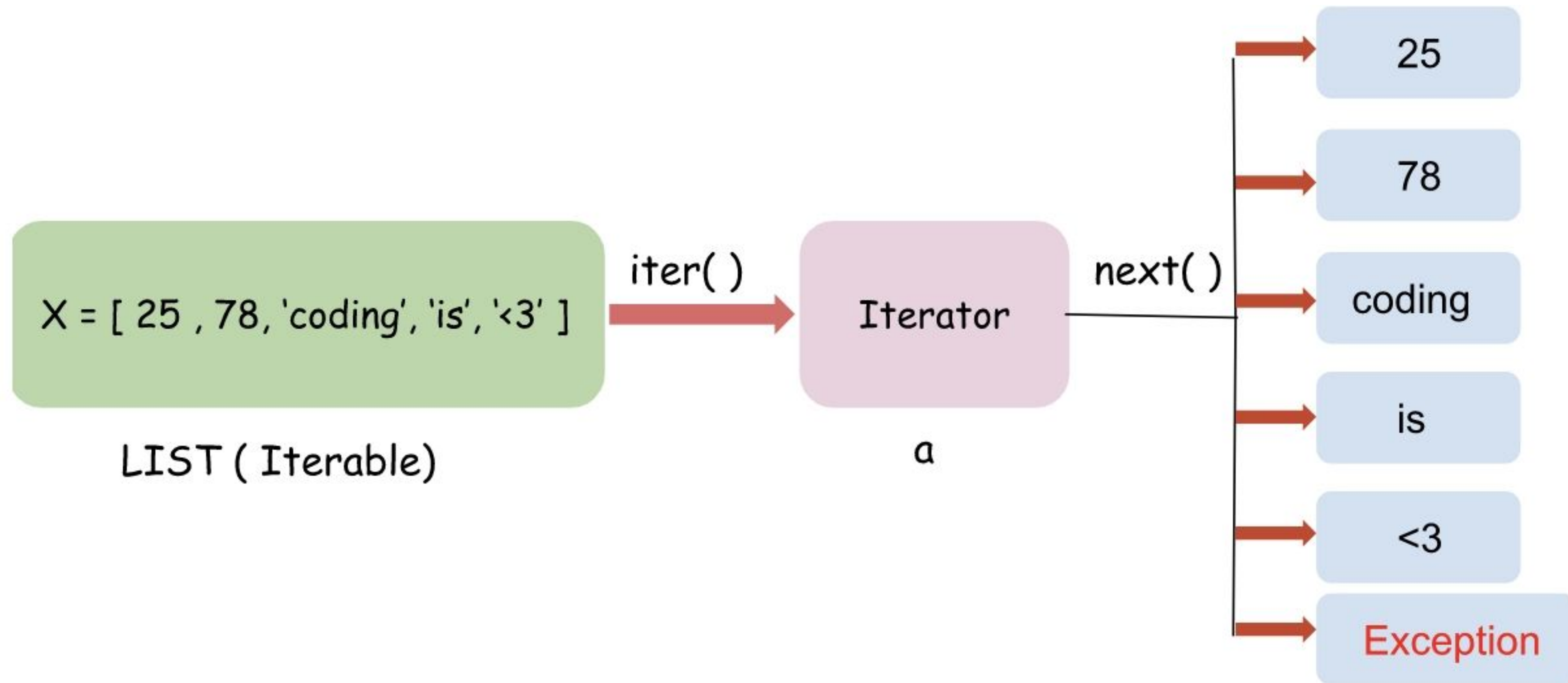
- **Using a Dictionary:**

python

Copy code

```
my_dict = {'a': 1, 'b': 2}
for key in my_dict:
    print(key, my_dict[key])
```

Iterables in Python



Advanced Use: Custom Iterables

1. Overview of Custom Iterable: The `Count` Class

- **Class Design:** The `Count` class is a custom iterable designed to count from a low number to a high number.
- **Key Methods:**
 - `__init__(self, low, high)` : Initializes the starting (`low`) and ending (`high`) values.
 - `__iter__(self)` : Returns the iterator object (itself in this case).
 - `__next__(self)` : Provides the next number in the sequence or raises `StopIteration` when done.

```
python Copy code  
  
class Count:  
    def __init__(self, low, high):  
        self.current = low  
        self.high = high  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.current > self.high:  
            raise StopIteration  
        else:  
            self.current += 1  
            return self.current - 1
```

Advanced Use: Custom Iterables

1. Overview of Custom Iterable: The `Count` Class

- **Class Design:** The `Count` class is a custom iterable designed to count from a low number to a high number.
- **Key Methods:**
 - `__init__(self, low, high)`: Initializes the starting (`low`) and ending (`high`) values.
 - `__iter__(self)`: Returns the iterator object (itself in this case).
 - `__next__(self)`: Provides the next number in the sequence or raises `StopIteration` when done.

- **Scenario:** Let's say you need to generate a sequence of numbers within a specific range to perform a series of actions for each number.

- **Usage Example:**

python

 Copy code

```
# Create an instance of Count to generate numbers from 1 to 5
counter = Count(1, 5)

# Iterate over the counter using a for loop
for number in counter:
    print(f"Processing number: {number}")
```

- **Expected Output:**

yaml

 Copy code

```
Processing number: 1
Processing number: 2
Processing number: 3
Processing number: 4
Processing number: 5
```

Custom Iterable: The **Range** Class in Python

1. Overview of the `Range` Class

- **Purpose:** Provides a flexible way to generate a sequence of numbers, similar to Python's built-in `range` but with customizable functionality.
- **Features:** Allows iteration over a sequence defined by start, stop, and step parameters.

2. Key Attributes

- **Start:** The starting value of the sequence.
- **Stop:** The endpoint (not included) of the sequence.
- **Step:** The increment (or decrement) between each number in the sequence.

3. Implementation Details

- **Constructor (`__init__`):** Initializes start, stop, and step values.
- **Iterator Method (`__iter__`):** Resets the iterator to the starting point.
- **Next Method (`__next__`):** Computes the next value in the sequence or ends the iteration.

4. Code Example

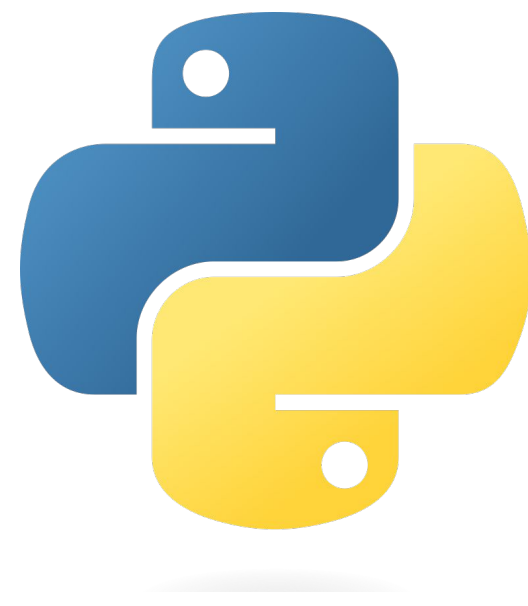
```
python Copy code

class Range:
    def __init__(self, start, stop, step=1):
        self.start = start
        self.stop = stop
        self.step = step
        self.current = start

    def __iter__(self):
        self.current = self.start # Reset for new iterations
        return self

    def __next__(self):
        if (self.step > 0 and self.current < self.stop) or (self.step < 0 and self.current > self.stop):
            number = self.current
            self.current += self.step
            return number
        else:
            raise StopIteration

# Usage
for number in Range(1, 10, 2):
    print(number) # Outputs: 1, 3, 5, 7, 9
```



Python

Functional Programming

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316

INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2

Python for Data Science

LECTURE 1

Introduction to Python Programming

Python Functional Programming

Key Concepts:

- What is **Functional Programming (FP)**?
 - A style of programming where **functions** are the main building blocks.
 - In FP, we focus on *what* to do rather than *how* to do it.

Procedural Programming

In procedural programming, the process is step-by-step, using functions to operate on data:

```
python Copy code

def square_number(num):
    return num * num

# Procedural use
number = 5
result = square_number(number)
print("Procedural Programming:")
print("The square of", number, "is", result)
```

Functional Programming with Lambda

```
python Copy code

# Lambda function to compute the square of a number
square_lambda = lambda x: x * x

# Use the lambda function
number = 5
result = square_lambda(number)
print("Functional Programming with Lambda:")
print("The square of", number, "is", result)
```

Python Functional Programming

1. Introduction to Functional Programming

- Functional programming is a paradigm that treats computation as the evaluation of mathematical functions.
- Avoids changing-state and mutable data.

2. Key Concepts

- **First-Class Functions:** Functions are treated as first-class citizens, meaning they can be variables, passed as arguments, returned from other functions, and stored in data structures.
- **Higher-Order Functions:** Functions that can take other functions as arguments or return them as results.
- **Pure Functions:** Functions that have no side effects and return a value that depends only on their input.

Procedural Programming

In procedural programming, the process is step-by-step, using functions to operate on data:

```
python Copy code

def square_number(num):
    return num * num

# Procedural use
number = 5
result = square_number(number)
print("Procedural Programming:")
print("The square of", number, "is", result)
```

Functional Programming with Lambda

```
python Copy code

# Lambda function to compute the square of a number
square_lambda = lambda x: x * x

# Use the lambda function
number = 5
result = square_lambda(number)
print("Functional Programming with Lambda:")
print("The square of", number, "is", result)
```

Python Functional Programming

3. Core Concepts

- **Expressions Over Statements:** Focus on expressions that produce values rather than statements that change state.
- **Declarative Approach:** Describes what to solve rather than how to solve it, promoting clarity and predictability.
- **Recursion Over Looping:** Prefers recursion for repeating operations instead of traditional loops.

4. Benefits

- Enhances predictability and facilitates debugging.
- Naturally supports concurrent execution.
- Reduces bugs by avoiding side effects and mutable state.

5. Application Areas

- Ideal for academic settings, large-scale data processing, and financial systems where reliability and concurrency are crucial.

```
python Copy code

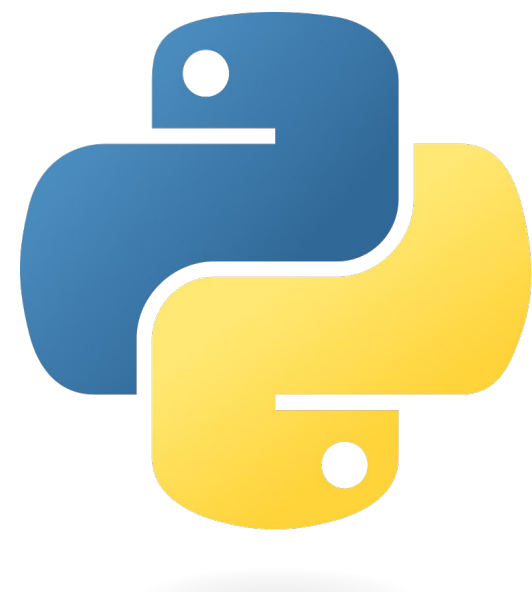
# Core Concept 1: Expressions Over Statements
# Here, we use a list comprehension, which is an expression that produces a list,
# instead of a loop that would change state by appending to an existing list.
squares = [x**2 for x in range(10)] # Expression that calculates squares of numbers
print("Squares:", squares)

# Core Concept 2: Declarative Approach
# This function declaratively describes what it does (filtering even numbers),
# rather than how to iterate and conditionally add items to a list.
def filter_evens(numbers):
    return [num for num in numbers if num % 2 == 0]

evens = filter_evens([1, 2, 3, 4, 5, 6])
print("Even Numbers:", evens)

# Core Concept 3: Recursion Over Looping
# This recursive function calculates factorial of a number using recursion,
# avoiding the use of mutable state and looping constructs.
def factorial(n):
    # Base case: factorial of 0 is 1
    if n == 0:
        return 1
    # Recursive case: n * factorial of (n-1)
    else:
        return n * factorial(n-1)

fact_of_5 = factorial(5)
print("Factorial of 5:", fact_of_5)
```



Python Functional Programming

Lambda Expressions

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Lambda Expressions in Python

1. What are Lambda Expressions?

- **Definition:** Lambda expressions in Python are small, anonymous functions created with the `lambda` keyword.
- **Syntax:** `lambda arguments: expression`
- **Characteristics:** Can have any number of arguments but only one expression. The result of the expression is automatically returned by the lambda function.

2. Role in Functional Programming

- **First-Class Functions:** Lambda expressions reinforce the idea that functions are first-class citizens in Python. They can be assigned to variables, passed as arguments, and returned from other functions just like any other object.
- **Quick and Disposable:** Perfect for short-lived functions that are not needed beyond their immediate use, reducing the need for formal function definitions.

Example:

python

 Copy code

```
# Regular function to add two numbers
def add(a, b):
    return a + b

# Lambda expression for the same functionality
add_lambda = lambda a, b: a + b

print(add_lambda(3, 5)) # Output: 8
```

Lambda Expressions in Python

- **Lambda expressions** are anonymous, one-line functions used for short, simple tasks.
- **Syntax:** `lambda arguments: expression`
- Ideal for use in places where a small function is needed temporarily.
- Typically used with built-in functions like `map()`, `filter()`, or `sorted()`.

Example:

python

 Copy code

```
# Regular function to add two numbers
def add(a, b):
    return a + b

# Lambda expression for the same functionality
add_lambda = lambda a, b: a + b

print(add_lambda(3, 5)) # Output: 8
```

Lambda as Anonymous Functions

1. Understanding Anonymous Functions

- **Definition:** Anonymous functions, also known as lambda functions in Python, are defined without a name and typically used for short, ad-hoc purposes where defining a formal function might be unnecessary.
- **Characteristics:** Compact, one-line functions that are not bound to a name, making them ideal for one-time use, especially within other functions or constructs.

2. Common Scenarios for Usage

- **Sorting:** Particularly useful in sorting or ordering collections based on a custom key.
- **Quick Calculations:** Efficient for executing small snippets of code like arithmetic operations, string manipulations, etc., without the overhead of defining a standard function.

3. Detailed Example: Sorting a List of Tuples

- **Problem:** Given a list of tuples where each tuple contains a number and a fruit name, sort the list based on the fruit names alphabetically.
- **Solution Using Lambda:**

```
python Copy code

data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
# Sorting tuples by the second element (fruit name) using a lambda function
sorted_data = sorted(data, key=lambda x: x[1])
print("Sorted by fruit name:", sorted_data)
# Output: [(1, 'apple'), (2, 'cherry'), (3, 'banana')]
```

4. Explanation of the Example

- The lambda function `lambda x: x[1]` acts as a key extractor for the `sorted()` function, which uses this key (fruit name) to determine the order of elements.
- This use of lambda eliminates the need for a separate, named function just for extracting the sorting key, showcasing lambda's utility in making the code more succinct and focused.

```
python Copy code

data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]

# Defining a function to get the second element of each tuple
def get_fruit_name(tup):
    return tup[1]

# Sorting tuples by the second element (fruit name) using the defined function
sorted_data = sorted(data, key=get_fruit_name)

print("Sorted by fruit name:", sorted_data)
```

Lambda as Function Arguments

1. Flexibility in Function Arguments

- **Concept Overview:** Lambda expressions can be passed directly to functions that expect a function argument, simplifying code by avoiding extraneous function definitions.
- **Benefits:** This use of lambda enhances code readability and efficiency, especially when the function logic is straightforward or only needed temporarily.

2. Key Applications

- **Sorting and Organizing Data:** Customize how data is sorted, filtered, or reduced directly within function calls.
- **Event Handlers and Callbacks:** Ideal for quick event handling in GUIs or other callback-based programming scenarios.

3. Detailed Example: Customizing List Sorting

- **Problem:** You have a list of names and need to sort them based on their length to prioritize shorter names in a user interface display.
- **Lambda Solution:**

```
python Copy code  
  
names = ['Alice', 'Bob', 'Charlie']  
# Sorting names by length using a lambda function as the key argument  
names.sort(key=lambda name: len(name))  
print("Names sorted by length:", names)  
# Output: ['Bob', 'Alice', 'Charlie']
```

4. Why Use Lambda Here?

- **Conciseness:** The lambda function provides a quick, one-line solution to define the sorting criteria, avoiding the overhead of a named function.
- **Contextual Relevance:** Embedding the lambda directly in the call makes the sorting criterion immediately visible and contextual, improving the maintainability and understandability of the code.

Lambda Expressions and Closures

1. Understanding Closures

- **Definition:** A closure is a function that remembers the environment in which it was created. It retains access to variables from its outer scope even after that scope has closed.

2. Lambdas as Closures

- **Functionality:** Lambda expressions can capture and keep references to variables from their surrounding scope, acting as closures.

3. Example: Capturing State with Lambda

- **Code:**

```
python Copy code  
  
# Capturing 'power' to compute powers of numbers  
power = 3  
calculate_power = lambda x: x ** power  
print("Result:", calculate_power(4)) # Output: 64
```

- **Explanation:** The lambda captures `power` at the time of definition, retaining its value even if `power` changes later.

4. Significance of Closures

- **Benefit:** Closures allow functions to maintain state from their defining scope, enabling dynamic function behavior.

Lambda Expressions and Closures

- **Definition of Closure**

- A closure is a function object that retains access to variables from its enclosing lexical scope, even after the outer function has completed execution.

- **Components of a Closure**

- **Outer Function**

- Defines variables and a nested function.

- **Inner Function**

- Uses variables from the outer function's scope.

- **Return**

- The outer function returns the inner function.

- **Characteristics of Closures**

- **Memory Retention**

- Closures hold onto their enclosing scope's environment, allowing them to access variables that are no longer in scope.

- **Encapsulation**

- Closures can hide information, acting similarly to objects with private variables.

- **Practical Use Cases**

- Maintaining state between function calls without using global variables.
- Data hiding and encapsulation in Python programming.

python

 Copy code

```
def outer_function(msg):  
    message = msg  
    def inner_function():  
        print(message)  
    return inner_function  
  
my_closure = outer_function("Hello, world!")  
my_closure() # Outputs: Hello, world!
```

Lambda Expressions and Closures

1. Understanding Closures

- **Definition:** A closure is a function that remembers the environment in which it was created. It retains access to variables from its outer scope even after that scope has closed.

2. Lambdas as Closures

- **Functionality:** Lambda expressions can capture and keep references to variables from their surrounding scope, acting as closures.

3. Example: Capturing State with Lambda

- **Code:**

```
python Copy code

# Capturing 'power' to compute powers of numbers
power = 3
calculate_power = lambda x: x ** power
print("Result:", calculate_power(4)) # Output: 64
```

- **Explanation:** The lambda captures `power` at the time of definition, retaining its value even if `power` changes later.

4. Significance of Closures

- **Benefit:** Closures allow functions to maintain state from their defining scope, enabling dynamic function behavior.

```
python Copy code

def multiplier(n): # Outer function
    return lambda x: x * n # Inner function as a lambda

# Creating a closure that multiplies by 3
triple = multiplier(3)

# Using the closure
print(triple(10)) # Output: 30
```

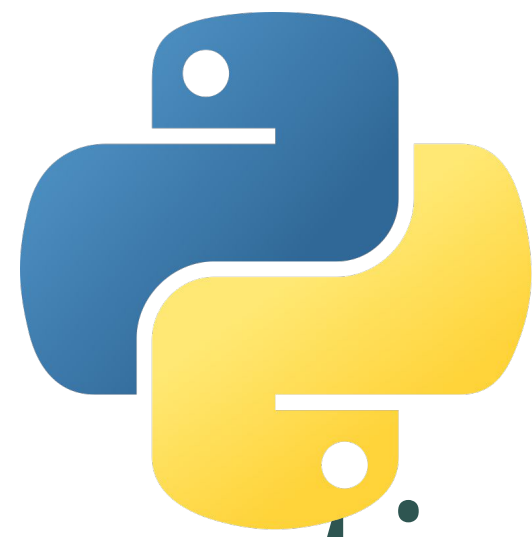
CS316
INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2
Python for Data Science

LECTURE 5
Functional Programming
Functional Programming Tools
Filter/Map/Reduce

Prof. Anis Koubaa

SEP 2024



Functional Programming Lambda Expressions with Functional Programming Tools

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Lambda Expressions in Python

1. What are Lambda Expressions?

- **Definition:** Lambda expressions in Python are small, anonymous functions created with the `lambda` keyword.
- **Syntax:** `lambda arguments: expression`
- **Characteristics:** Can have any number of arguments but only one expression. The result of the expression is automatically returned by the lambda function.

2. Role in Functional Programming

- **First-Class Functions:** Lambda expressions reinforce the idea that functions are first-class citizens in Python. They can be assigned to variables, passed as arguments, and returned from other functions just like any other object.
- **Quick and Disposable:** Perfect for short-lived functions that are not needed beyond their immediate use, reducing the need for formal function definitions.

Example:

python

 Copy code

```
# Regular function to add two numbers
def add(a, b):
    return a + b

# Lambda expression for the same functionality
add_lambda = lambda a, b: a + b

print(add_lambda(3, 5)) # Output: 8
```

Functional Programming Tools

1. Introduction to Functional Programming Tools

- **Overview:** Functional programming tools often involve operations that apply functions to collections of data, such as lists or arrays.
- **Key Tools in Python:** `map()`, `filter()`, and `reduce()` are classic functional programming tools used for transforming, filtering, and accumulating data.

Essential Functional Programming Tools

1. Core Functional Programming Tools

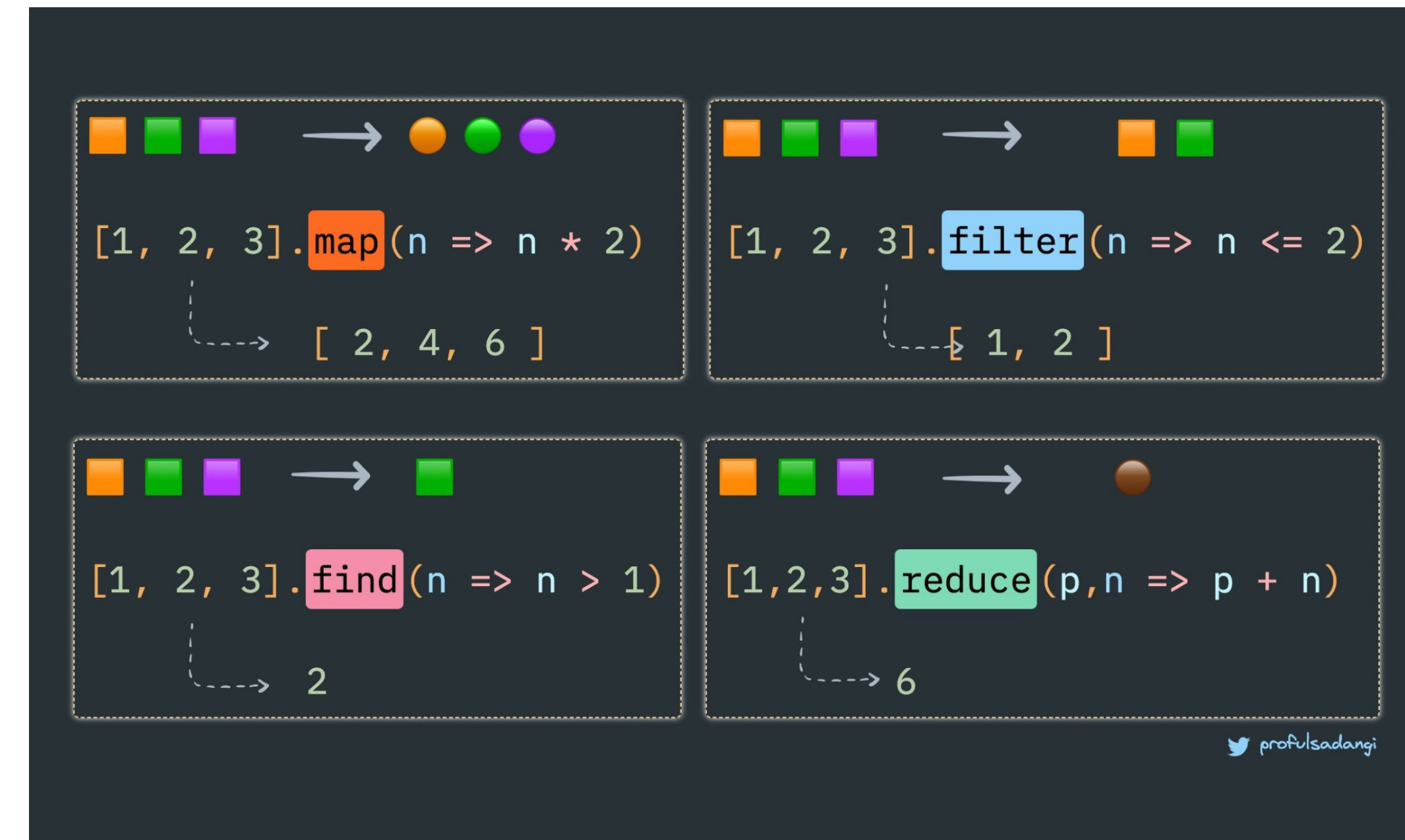
- `map()` : Apply a function to every item of an iterable and return a list of the results.
- `filter()` : Construct an iterator from those elements of iterable for which a function returns true.
- `reduce()` : Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.

2. Additional Functional Techniques

- **List Comprehensions**: Concise syntax for creating lists based on existing lists.
- **Generator Expressions**: Similar to list comprehensions but generates items on the fly without storing them all at once.

3. Useful Functional Modules

- `functools.partial` : Fix certain arguments of a function and generate a new function.
- `itertools` : A collection of tools for handling iterators. Useful for complex data manipulation.
- `functools.reduce` (reemphasis): Powerful tool for performing cumulative computations over a list.



List Comprehension

List Comprehensions in Python

- Definition of List Comprehension

- A concise way to create lists based on existing lists or iterable objects.

- Basic Structure

- `[expression for item in iterable if condition]`

- Components:

- **Expression:** What to do with each item (e.g., `item*2`).
- **Iterable:** The source of data (e.g., `range(10)`).
- **Condition (optional):** A filter to include only certain items (e.g., `if item % 2 == 0`).

- Benefits of Using List Comprehensions

- **Conciseness:** Reduces the number of lines of code.
- **Readability:** Easier to read and write once familiar.
- **Performance:** Generally faster than equivalent for-loops for creating lists.

- Example Code

python

 Copy code

```
# Creating a list of squares of even numbers from 1 to 10
squares = [x**2 for x in range(1, 11) if x % 2 == 0]
print(squares) # Output: [4, 16, 36, 64, 100]
```

- Comparison with Traditional Loops

python

 Copy code

```
# Traditional loop approach
squares = []
for x in range(1, 11):
    if x % 2 == 0:
        squares.append(x**2)
```

List Comprehensions in Python

Example: Filtering and Transforming Data

[expression for item in iterable if condition]

Suppose you have a list of dictionaries, each representing a person's details, and you want to extract the names of people who are over 18 years old.

python

Copy code

```
people = [
    {'name': 'Alice', 'age': 25},
    {'name': 'Bob', 'age': 17},
    {'name': 'Cara', 'age': 32},
    {'name': 'Dave', 'age': 15}
]

# Using list comprehension to filter and extract names of adults
adult_names = [person['name'] for person in people if person['age'] > 18]
print(adult_names) # Output: ['Alice', 'Cara']
```

- **people** is a list of dictionaries where each dictionary contains the keys `'name'` and `'age'`.
- The list comprehension iterates over each dictionary (each `person`) in the list.
- The **expression** `person['name']` extracts the value associated with the key `'name'` from each dictionary.
- The **condition** `if person['age'] > 18` filters out dictionaries where the associated value for the key `'age'` is 18 or less.
- **adult_names** will contain the names of the people who are older than 18 years, as a result of this comprehension.

Generator Expressions

Generator Expressions in Python

1. What is a Generator Expression?

- **Definition:** A generator expression is a concise way to create a generator — a special type of iterator that lazily generates values on the fly without storing them in memory.
- **Syntax:** `(expression for item in iterable if condition)`, resembling list comprehensions but using parentheses instead of square brackets.

2. How Generator Expressions Work

- **Process:** Takes an expression followed by a for clause and optionally, a filtering if clause. It is evaluated to produce items one at a time, only as needed.
- **Returns:** A generator object that can be iterated over to lazily evaluate each item as requested, using minimal memory.

3. Benefits of Using Generator Expressions

- **Memory Efficiency:** Consumes less memory as items are generated and consumed on demand, suitable for large data streams.
- **Performance:** Can lead to performance gains, especially in scenarios with large data sets where not all data needs to be stored in memory simultaneously.
- **Composability:** Can be easily combined with other Python functionalities like `sum()`, `max()`, `min()`, and wrapped inside functions.

• Simple Generator Example:

python

Copy code

```
# Generate squares of numbers up to 10
squares = (x**2 for x in range(11))
for square in squares:
    print(square)
```

• Using with Built-in Functions:

python

Copy code

```
# Calculate the sum of the squares of numbers up to 10
sum_of_squares = sum(x**2 for x in range(11))
print("Sum of squares:", sum_of_squares)
```

5. Comparison with List Comprehensions

- **List Comprehensions:** Create a list of all items at once, consuming more memory.
- **Generator Expressions:** Generate items one at a time, only when needed, significantly reducing memory usage.

6. Practical Use Cases

- **Data Processing:** Ideal for processing streams of data that might be too large to fit into memory.
- **Function Arguments:** Often used in function arguments for aggregation functions (`sum`, `min`, `max`) to process items without creating temporary data structures.

Example: Filtering and Processing Employee Data

```
cs316_ws > cs316_package > generator.py > ...
1  # List of employee dictionaries
2  employees = [
3      {'name': 'Alice', 'age': 30, 'department': 'Engineering'},
4      {'name': 'Bob', 'age': 25, 'department': 'Marketing'},
5      {'name': 'Charlie', 'age': 35, 'department': 'Engineering'},
6      {'name': 'David', 'age': 40, 'department': 'HR'},
7      {'name': 'Eve', 'age': 23, 'department': 'Marketing'}
8  ]
9
10 # Generator expression to filter and extract ages of employees in Engineering
11 engineering_ages = (emp['age'] for emp in employees
12                     if emp['department'] == 'Engineering')
13
14 print("Engineering Ages:", list(engineering_ages))
15
16
17 # Calculate the average age using the generator
18 if engineering_ages:
19     average_age = sum(engineering_ages) / sum(1 for emp in employees
20                                             if emp['department'] == 'Engineering')
21     print("Average age in Engineering:", average_age)
22
```

Generator Expression: The first generator expression `(emp['age'] for emp in employees if emp['department'] == 'Engineering')` is used to lazily evaluate and yield ages of employees who are in the Engineering department.

The issue with your code arises because generator expressions in Python can only be iterated over once. After the first iteration (when you print the ages using `list(engineering_ages)`), the generator is exhausted. This means that there are no values left in `engineering_ages` when it's used again to calculate the average age.

When you attempt to calculate the sum of `engineering_ages` for the average age computation, it returns `0` because there are no items left in the generator to iterate over. Thus, the sum is `0` and the division results in `0.0`.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

> **TERMINAL**

- (reviewgpt) Aniss-MacBook-Pro:cs316_package akoubaa\$ python generator.py
Engineering Ages: [30, 35]
Average age in Engineering: 0.0
- (reviewgpt) Aniss-MacBook-Pro:cs316_package akoubaa\$

Example: Filtering and Processing Employee Data

```
cs316_ws > cs316_package > generator.py > ...
1  # List of employee dictionaries
2  employees = [
3      {'name': 'Alice', 'age': 30, 'department': 'Engineering'},
4      {'name': 'Bob', 'age': 25, 'department': 'Marketing'},
5      {'name': 'Charlie', 'age': 35, 'department': 'Engineering'},
6      {'name': 'David', 'age': 40, 'department': 'HR'},
7      {'name': 'Eve', 'age': 23, 'department': 'Marketing'}
8  ]
9
10 # Generator expression to filter and extract ages of employees in Engineering
11 engineering_ages = (emp['age'] for emp in employees
12                     if emp['department'] == 'Engineering')
13
14 print("Engineering Ages:", engineering_ages)
15
16
17 # Calculate the average age using the generator
18 if engineering_ages:
19     average_age = sum(engineering_ages) / sum(1 for emp in employees
20         if emp['department'] == 'Engineering')
21     print("Average age in Engineering:", average_age)
22
```

Generator Expression: The first generator expression `(emp['age'] for emp in employees if emp['department'] == 'Engineering')` is used to lazily evaluate and yield ages of employees who are in the Engineering department.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

>

▼ TERMINAL

● (reviewgpt) Aniss-MacBook-Pro:cs316_package akoubaa\$ python generator.py

Engineering Ages: <generator object <genexpr> at 0x100a57510>

Average age in Engineering: 32.5

○ (reviewgpt) Aniss-MacBook-Pro:cs316_package akoubaa\$ █

Example: Filtering and Processing Employee Data

```
cs316_ws > cs316_package > generator.py > ...
1  # List of employee dictionaries
2  employees = [
3      {'name': 'Alice', 'age': 30, 'department': 'Engineering'},
4      {'name': 'Bob', 'age': 25, 'department': 'Marketing'},
5      {'name': 'Charlie', 'age': 35, 'department': 'Engineering'},
6      {'name': 'David', 'age': 40, 'department': 'HR'},
7      {'name': 'Eve', 'age': 23, 'department': 'Marketing'}
8  ]
9
10 # Creating a list to extract ages of employees in Engineering for multiple uses
11 engineering_ages = [emp['age'] for emp in employees if emp['department'] == 'Engineering']
12
13 print("Engineering Ages:", engineering_ages)
14
15 # Calculate the average age using the list
16 if engineering_ages:
17     average_age = sum(engineering_ages) / len(engineering_ages)
18     print("Average age in Engineering:", average_age)
19 else:
20     print("No employees in Engineering")
21
```

- I changed the generator expression to a list comprehension. This allows `engineering_ages` to be used multiple times.
- The average is calculated using `len(engineering_ages)` which correctly counts the number of elements in the list, ensuring it works even if the list is used before.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

> **TERMINAL**

• (reviewgpt) Aniss-MacBook-Pro:cs316_package akoubaa\$ python generator.py

Engineering Ages: [30, 35]

Average age in Engineering: 32.5

○ (reviewgpt) Aniss-MacBook-Pro:cs316_package akoubaa\$ █

The `map()` Function

The `map()` Function in Python

1. What is `map()` ?

- **Definition:** The `map()` function applies a given function to each item of an iterable (such as list or tuple) and returns a map object (an iterator).
- **Syntax:** `map(function, iterable, ...)`

2. How `map()` Works

- **Process:** Takes a function and one or more iterables as arguments. The function is applied to each item of the iterable. Multiple iterables can be passed if the function takes multiple arguments.
- **Returns:** An iterator that computes the function using arguments from each iterable on demand.

4. Benefits of Using `map()`

- **Efficiency:** Lazy evaluation (elements are processed on demand) saves memory and potentially computing time.
- **Clean and Expressive:** Simplifies code, avoiding explicit loops for applying operations on iterables.
- **Versatility:** Works with any function that can be applied to the elements of an iterable, including user-defined functions.

• Single Iterable Example:

python

 Copy code

```
# Doubling each item in a list
numbers = [1, 2, 3, 4]
result = map(lambda x: x * 2, numbers)
print(list(result)) # Output: [2, 4, 6, 8]
```

• Multiple Iterables Example:

python

 Copy code

```
# Adding elements of two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = map(lambda x, y: x + y, list1, list2)
print(list(result)) # Output: [5, 7, 9]
```

The `map()` Function in Python

Equivalent Code Without Using `map()`

```
python Copy code

# Define a function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(c):
    return (c * 9/5) + 32

# List of temperatures in Celsius
temperatures_celsius = [0, 10, 20, 30, 40]

# List to store the converted temperatures
temperatures_fahrenheit = []

# Loop through the list and apply the conversion function
for temp in temperatures_celsius:
    temperatures_fahrenheit.append(celsius_to_fahrenheit(temp))

# Print the results
print(temperatures_fahrenheit) # Output: [32.0, 50.0, 68.0, 86.0, 104.0]
```

Using the `map()` Function

```
python Copy code

# Define a function to convert Celsius to Fahrenheit
def celsius_to_fahrenheit(c):
    return (c * 9/5) + 32

# List of temperatures in Celsius
temperatures_celsius = [0, 10, 20, 30, 40]

# Use map to apply the conversion function to the list
temperatures_fahrenheit = map(celsius_to_fahrenheit, temperatures_celsius)

# Convert map object to a list to print the results
print(list(temperatures_fahrenheit)) # Output: [32.0, 50.0, 68.0, 86.0, 104.0]
```

Advanced Concepts and Performance of `map()` Function in Python

- **Error Handling in `map()`**
 - `map()` will immediately stop and raise an error if the function it applies throws an exception for any item.
- **`map()` vs. List Comprehensions**
 - List comprehensions offer more flexibility and readability for simple expressions.
 - `map()` is more memory efficient for large data sets because it evaluates items lazily.
- **Performance Insights**
 - Best used when the transformation function is an already optimized built-in function.
 - Preferable in scenarios where not all results may need to be computed immediately.
- **Integrating with Other Functions**
 - Example using a built-in function: `list(map(str.upper, ['hello', 'world']))`
 - Efficient when used with `filter()`: `list(filter(None, map(f, iterable)))`
- **Memory Usage**
 - Does not store all results in memory, unlike list comprehensions, making it suitable for large or infinite data streams.

The `map()` Function Memory Efficiency

python

 Copy code

```
import sys

# Define a large range
large_range = range(1000000)

# Using list comprehension
large_list = [x * 2 for x in large_range]
print("Memory (List Comprehension):", sys.getsizeof(large_list), "bytes")

# Using map function
large_map = map(lambda x: x * 2, large_range)
print("Memory (Map Function):", sys.getsizeof(large_map), "bytes")

# Print the type to emphasize it's not a list but a map object
print("Type of result from map():", type(large_map))
```

1. Memory Consumption:

- The `large_list` created with list comprehension immediately allocates memory for all results, evident from the size output.
- The `large_map` created using `map()` shows significantly lower memory usage because it only stores the state and logic needed to produce the items on demand, not the items themselves.

2. Type Check:

- This demonstrates that the result from `map()` is not a list but a `map` object, which is an iterator.

3. Performance Note:

- If you need to access only a few elements or iterate over them once without keeping them in memory, `map()` is more efficient.
- This efficiency becomes particularly important with very large or potentially infinite data streams where you cannot or do not want to store all items in memory at once.

```
Memory (List Comprehension): 8448728 bytes
```

```
Memory (Map Function): 48 bytes
```

```
Type of result from map(): <class 'map'>
```

The `filter()` Function

The `filter()` Function in Python

1. What is `filter()` ?

- **Definition:** The `filter()` function applies a function to each item of an iterable (such as a list or tuple) and returns a filter object (an iterator) that contains only those elements for which the function returns `True`.
- **Syntax:** `filter(function, iterable)`

2. How `filter()` Works

- **Process:** Takes a function and an iterable as arguments. The function is applied to each item of the iterable, and the function should return a boolean value.
- **Returns:** An iterator that yields only the items from the iterable where the function returned `True`.

3. Benefits of Using `filter()`

- **Efficiency:** Lazy evaluation—elements are processed on demand, saving memory and potentially computing time.
- **Clean and Expressive:** Simplifies code by abstracting the looping and condition checking, which makes the code more readable.
- **Versatility:** Works with any function that returns a Boolean value, allowing for flexible filtering criteria.



• Single Iterable Example:

python

Copy code

```
# Filtering to get only even numbers in a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
result = filter(lambda x: x % 2 == 0, numbers)
print(list(result)) # Output: [2, 4, 6, 8]
```

• Using None as a Function:

python

Copy code

```
# Filtering out non-true values in a list
mixed_values = [0, "Hello", "", None, 42, False]
result = filter(None, mixed_values)
print(list(result)) # Output: ['Hello', 42]
```

The `filter()` Function in Python

Example with Lambda

```
python Copy code

# List of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using filter with a lambda function to get even numbers
even_numbers_lambda = filter(lambda x: x % 2 == 0, numbers)
print("Even numbers with lambda:", list(even_numbers_lambda))
```

Example without Lambda (using a defined function)

```
python Copy code

# List of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Defining a function to check if a number is even
def is_even(num):
    return num % 2 == 0

# Using filter with a defined function to get even numbers
even_numbers_function = filter(is_even, numbers)
print("Even numbers with function:", list(even_numbers_function))
```

The `filter()` Function in Python

3. Using Lambda with `filter()`

- **Purpose:** Filters items out of an iterable by applying a function that returns True or False.
- **Example:**

python

 Copy code

```
# Filtering out odd numbers from a list using filter and lambda
evens = list(filter(lambda x: x % 2 == 0, numbers))
print("Even Numbers:", evens)
```

Advanced Concepts in Using the `filter()` Function in Python

1. Error Handling in `filter()`

- **Behavior:** `filter()` will continue processing items even if the function it applies returns `None`. It doesn't stop or raise an error unless there is an actual exception in the function logic.

2. `filter()` vs. List Comprehensions

- **Flexibility and Readability:** While list comprehensions provide inline filtering with more complex logic, `filter()` excels with simpler, well-defined functions.
- **Memory Efficiency:** Like `map()`, `filter()` is more memory efficient for large data sets because it generates results lazily.

3. Performance Insights

- **Optimal Usage:** Best when the filtering condition is a simple function that doesn't require the overhead of an explicit loop.
- **Scenarios:** Particularly effective in scenarios where only a subset of the data needs to be processed, reducing unnecessary computation.

4. Integrating with Other Functions

- **Combining with `map()`:** Example: `list(filter(None, map(str.isalpha, ['abc', '123', 'xyz'])))` filters and processes elements based on alpha checks.
- **Chaining with `reduce()`:** Can be used prior to `reduce()` to filter out unwanted data elements, optimizing the subsequent reduction process.

5. Memory Usage

- **Lazy Evaluation:** Does not create an intermediate list, generating items one by one and only when needed, making it efficient for memory use.
- **Single Condition Example:**

```
python Copy code

# Filter out non-numeric characters from a list
chars = ['a', '1', 'b', '2']
numeric_chars = filter(str.isdigit, chars)
print(list(numeric_chars)) # Output: ['1', '2']
```

- **Complex Conditions Example:**

```
python Copy code

# Filter employees by multiple conditions
employees = [
    {'name': 'Alice', 'department': 'Engineering', 'status': 'Full-time'},
    {'name': 'Bob', 'department': 'HR', 'status': 'Part-time'}
]
engineering_full_time = filter(lambda x: x['department'] == 'Engineering' and x['status'] == 'Full-time', employees)
print(list(engineering_full_time)) # Output: [{'name': 'Alice', 'department': 'Engineering', 'status': 'Full-time'}]
```

Advanced Concepts in Using the `filter()` Function in Python

- Single Condition Example:

```
python Copy code  
  
# Filter out non-numeric characters from a list  
chars = ['a', '1', 'b', '2']  
numeric_chars = filter(str.isdigit, chars)  
print(list(numeric_chars)) # Output: ['1', '2']
```

- Complex Conditions Example:

```
python Copy code  
  
# Filter employees by multiple conditions  
employees = [  
    {'name': 'Alice', 'department': 'Engineering', 'status': 'Full-time'},  
    {'name': 'Bob', 'department': 'HR', 'status': 'Part-time'}  
]  
engineering_full_time = filter(lambda x: x['department'] == 'Engineering' and x['status'] == 'Full-time', employees)  
print(list(engineering_full_time)) # Output: [{'name': 'Alice', 'department': 'Engineering', 'status': 'Full-time'}]
```

The `reduce()` Function

The `reduce()` Function in Python

1. What is `reduce()` ?

- **Definition:** The `reduce()` function applies a given binary function to the items of an iterable, typically to reduce these items to a single value. It is part of the `functools` module.
- **Syntax:** `from functools import reduce; reduce(function, iterable[, initializer])`

2. How `reduce()` Works

- **Process:** Takes a binary function (function with two arguments) and an iterable as arguments. The function is applied successively to the elements of the iterable. An optional initializer can be used as the initial value.
- **Returns:** A single value that results from the successive reduction process.

3. Benefits of Using `reduce()`

- **Efficiency:** Useful for performing computations where each step depends on the previous one.
- **Compact Code:** Reduces the need for explicit loops for cumulative operations, making the code cleaner.
- **Flexibility:** Can be used with any function that takes two arguments, making it suitable for various reduction strategies.

• Summing an Iterable:

```
python Copy code  
  
from functools import reduce  
numbers = [1, 2, 3, 4, 5]  
result = reduce(lambda x, y: x + y, numbers)  
print(result) # Output: 15
```

• Finding the Maximum:

```
python Copy code  
  
from functools import reduce  
numbers = [5, 8, 6, 10, 9]  
max_result = reduce(lambda x, y: x if x > y else y, numbers)  
print(max_result) # Output: 10
```

The `reduce()` Function in Python

Example with Lambda

```
python Copy code

from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Using reduce with a lambda function to calculate the sum
sum_numbers_lambda = reduce(lambda x, y: x + y, numbers)
print("Sum with lambda:", sum_numbers_lambda) # Output: 15
```

Example without Lambda (using a defined function)

```
python Copy code

from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Defining a function to sum two numbers
def sum_numbers(x, y):
    return x + y

# Using reduce with a defined function to calculate the sum
sum_numbers_function = reduce(sum_numbers, numbers)
print("Sum with function:", sum_numbers_function) # Output: 15
```

The `reduce()` Function in Python

Example with Lambda

```
python Copy code

from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Using reduce with a lambda function to calculate the sum
sum_numbers_lambda = reduce(lambda x, y: x + y, numbers)
print("Sum with lambda:", sum_numbers_lambda) # Output: 15
```

Example without Lambda (using a defined function)

```
python Copy code

from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Defining a function to sum two numbers
def sum_numbers(x, y):
    return x + y

# Using reduce with a defined function to calculate the sum
sum_numbers_function = reduce(sum_numbers, numbers)
print("Sum with function:", sum_numbers_function) # Output: 15
```

The `reduce()` Function in Python

Finding the Maximum:

python

Copy code

```
from functools import reduce
numbers = [5, 8, 6, 10, 9]
max_result = reduce(lambda x, y: x if x > y else y, numbers)
print(max_result) # Output: 10
```

python

Copy code

```
from functools import reduce

# List of numbers
numbers = [5, 8, 6, 10, 9]

# Defining a function to find the maximum of two numbers
def max_of_two(x, y):
    return x if x > y else y

# Using reduce with a defined function to find the maximum value
max_result = reduce(max_of_two, numbers)
print(max_result) # Output: 10
```

The `reduce()` Function in Python

4. Using Lambda with `reduce()`

- **Purpose:** Reduces an iterable to a single cumulative value by applying a binary function cumulatively to the items of an iterable.
- **Example:**

python

 Copy code

```
from functools import reduce
# Summing all numbers in a list using reduce and lambda
total = reduce(lambda x, y: x + y, numbers)
print("Total Sum:", total)
```

The `reduce()` Function in Python

1. Error Handling in `reduce()`

- **Behavior:** `reduce()` will stop processing and raise an error if the function it applies throws an exception at any step of the reduction. It does not handle errors internally.
- **Recovery:** Errors must be handled outside of `reduce()` or within the function itself using try-except blocks.

2. `reduce()` vs. Loops

- **Flexibility and Readability:** While explicit loops provide more control over the iteration and can handle complex logic, `reduce()` is more concise for simple accumulations or transformations.
- **Memory Efficiency:** `reduce()` can be more memory efficient than loops that require temporary lists or complex data structures, as `reduce()` consolidates all operations into a single return value.

3. Performance Insights

- **Optimal Usage:** Best when used to apply a function that combines elements into a single result, especially for operations like summing, finding maximum or minimum, or accumulating custom metrics.
- **Scenarios:** Ideal for sequences where the result depends on a binary operation applied cumulatively, reducing overhead compared to manual loop implementations.

4. Integrating with Other Functions

- **Example using `map()`:** Preprocessing elements before reduction, such as normalizing or adjusting values, which can then be summed or merged.
- **Chaining with `filter()`:** Filtering data before reducing can optimize the process by eliminating unnecessary computations.

5. Common Use Cases

- **Mathematical Operations:** Summing numbers, multiplying elements, or computing averages.
- **Data Transformation:** Reducing a list of dictionaries into a single dictionary by merging or accumulating values based on keys.

6. Practical Example Code

```
python Copy code

from functools import reduce

# Reducing a list of numbers to calculate the product
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print("Product of numbers:", product) # Output: 120
```

7. Key Takeaway

- `reduce()` is a powerful functional programming tool for efficiently condensing a list or iterable into a single value, enabling clean and expressive code solutions for specific types of problems.

CS316

INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2

Python for Data Science

LECTURE 6

Python Built-In Data Structures

List, Tuple, Set, Dictionary

Prof. Anis Koubaa

SEP 2024

Introduction to Data Structures

What Are Data Structures?

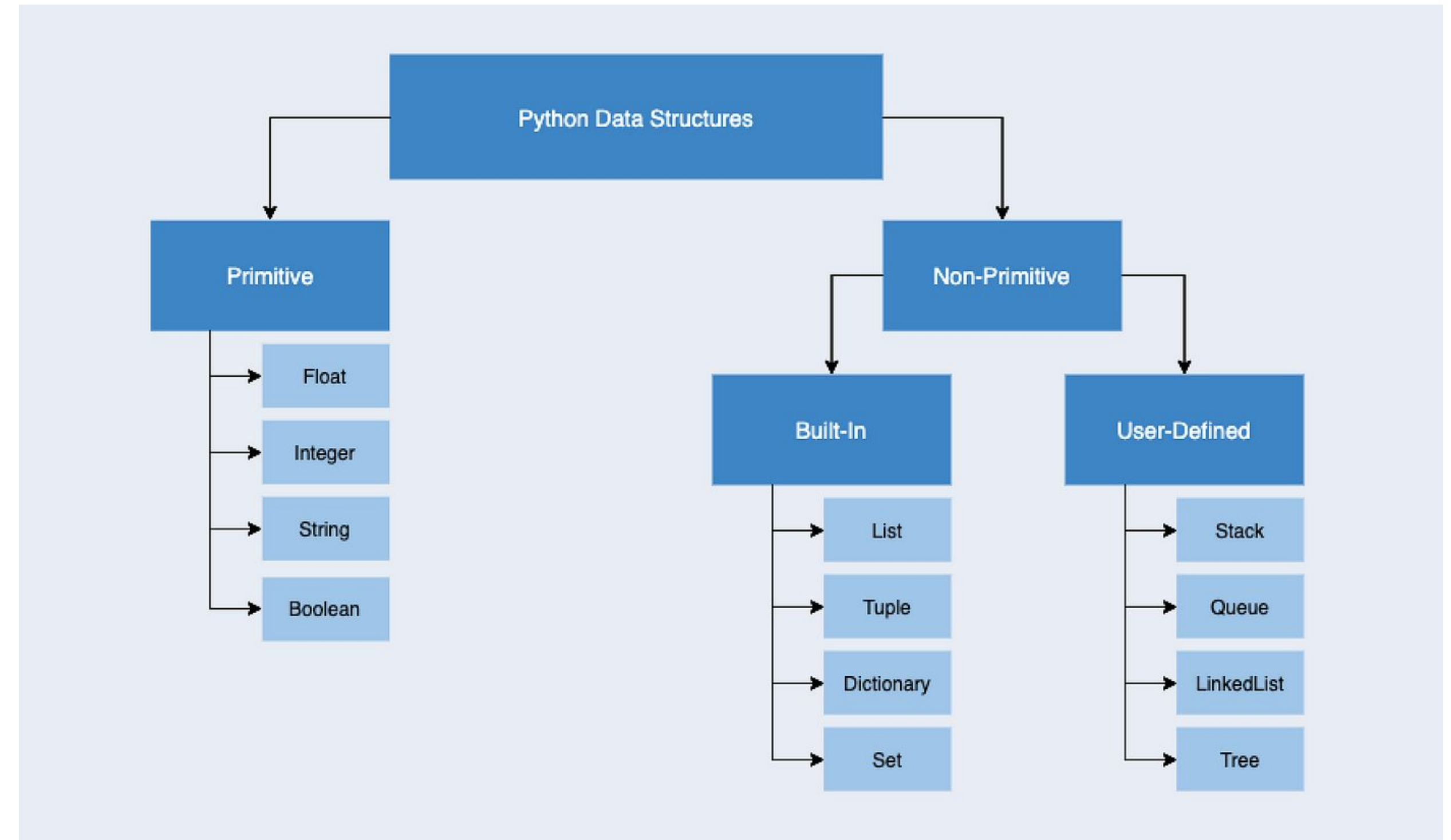
- Structures that store and organize data
- Facilitate efficient access and modification

Why Are Data Structures Important?

- Crucial for writing efficient algorithms
- Impact performance of software applications
- Essential for managing complexity in large data sets

Types of Data Structures

- **Built-in Data Structures:** Ready-to-use in programming languages (e.g., Python lists, dictionaries)
- **Custom Data Structures:** User-defined to solve specific problems (e.g., linked lists, trees, graphs)



<https://towardsdatascience.com/when-python-data-structure-should-you-use-fa1edd82946c>

Built-in vs. Custom Data Structures

- **Built-in:**
 - Provided by Python
 - Fast and reliable
 - Simplify common data storage tasks
- **Custom:**
 - Designed by the programmer
 - Tailored to specific problems
 - Can be optimized for performance

Conclusion

- Choosing the right data structure affects code efficiency and simplicity
- Python's built-in options provide robust tools for most needs

Impact of Data Structure Choice on Efficiency

Objective:

- Demonstrate how the choice of data structure affects the performance of membership tests.

Python Code Example:

- **Setup:** Create a list and a set with 1,000,000 elements each.
- **Test:** Measure how long it takes to find an element at the end.

Key Takeaways:

- Lists are slower for membership tests, especially with large data sizes.
- Sets offer significantly faster membership checking due to hashing.
- Choosing the correct data structure can drastically improve code efficiency.

Discussion:

- Why do you think the set performs so much better in this scenario?
- How might these results influence our choices in real-world applications?

Time to find an element in the list: **0.401631** seconds
Time to find an element in the set: **0.000007** seconds

```
python Copy code

import timeit

# Create a list and a set with a million elements
large_list = list(range(1000000))
large_set = set(range(1000000))

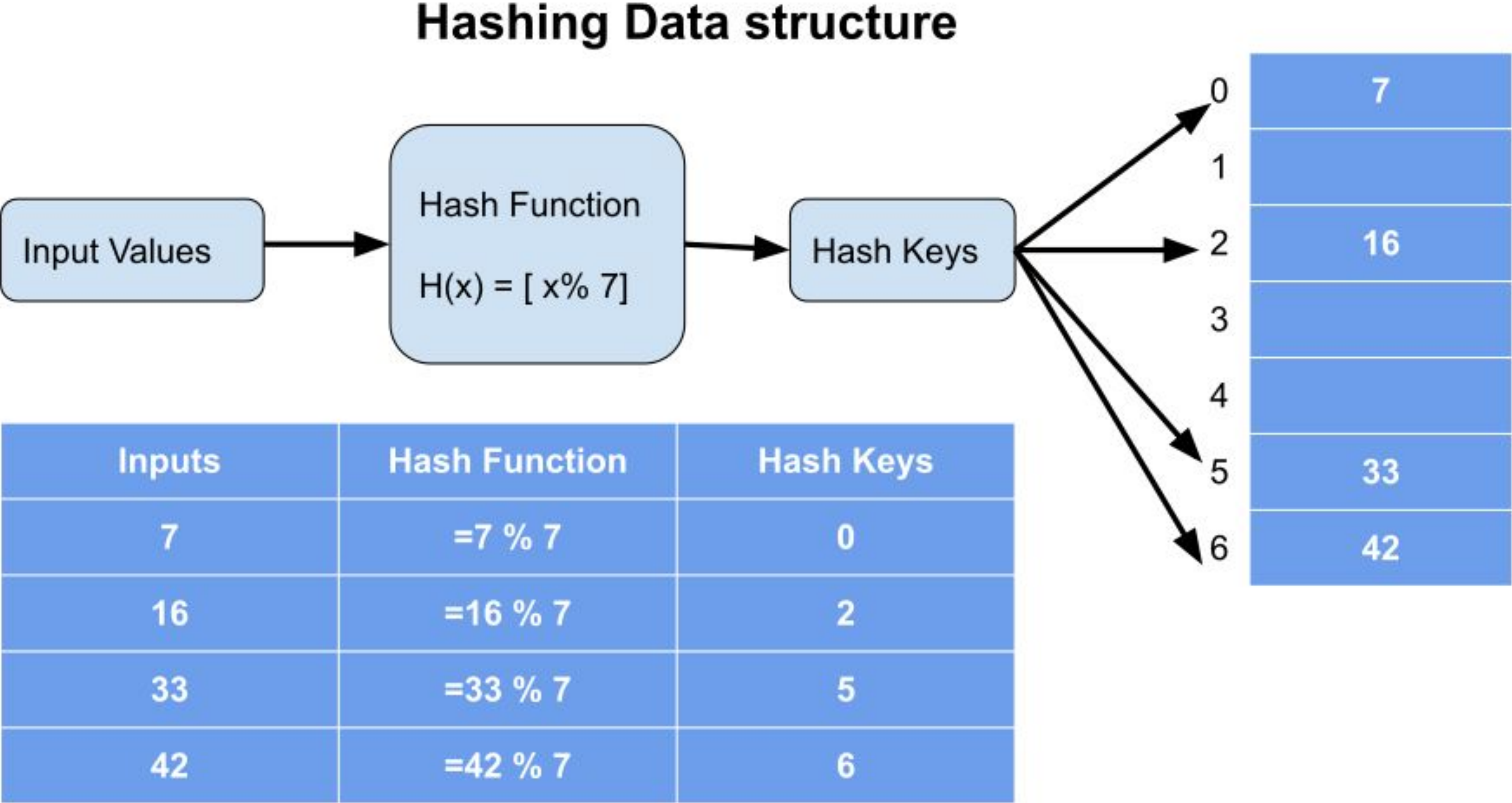
# Define functions to test membership
def test_list_membership():
    return 999999 in large_list

def test_set_membership():
    return 999999 in large_set

# Measure the time it takes to find an element at the end of the list and set
list_time = timeit.timeit(test_list_membership, number=100)
set_time = timeit.timeit(test_set_membership, number=100)

print(f"Time to find an element in the list: {list_time:.6f} seconds")
print(f"Time to find an element in the set: {set_time:.6f} seconds")
```

Hashing in Data Structures



<https://technicalsand.com/hashing-in-data-structure/>

Efficiency Comparison: List and NumPy Array

Objective:

- Understand how NumPy arrays enhance performance for numerical operations compared to lists.

Python Code Example:

- **Setup:** Create both a list and a NumPy array with 1,000,000 numerical elements.
- **Operation:** Sum all elements in the data structure.

Key Takeaways:

- NumPy arrays provide significant performance improvements for operations involving large numerical data.
- The efficiency gain is due to NumPy's internal optimizations and C-based processing.
- Using NumPy can lead to more scalable and faster code for data-intensive tasks.

Discussion:

- What implications do these results have for data processing and analysis?
- In what scenarios might you still prefer to use a list over a NumPy array?

Time to sum elements in the list: **0.302939** seconds

Time to sum elements in the NumPy array: **0.009983** seconds

python

 Copy code

```
import timeit
import numpy as np

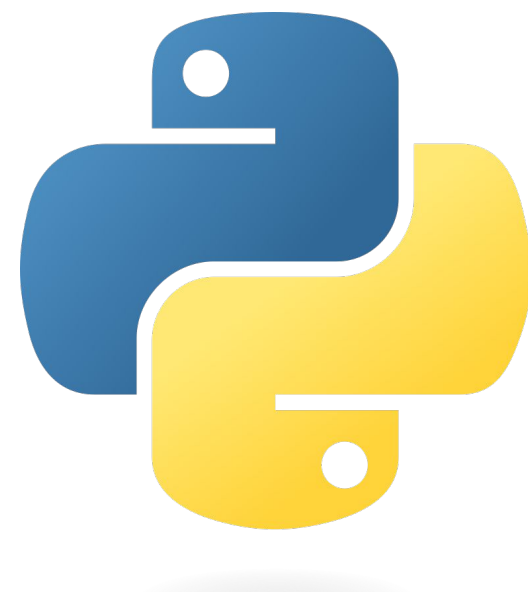
# Create a large list and a NumPy array with a million elements each
large_list = list(range(1000000))
large_array = np.array(large_list)

# Define functions to sum elements
def sum_list():
    return sum(large_list)

def sum_array():
    return np.sum(large_array)

# Measure the time it takes to sum the elements
list_time = timeit.timeit(sum_list, number=100)
array_time = timeit.timeit(sum_array, number=100)

print(f"Time to sum elements in the list: {list_time:.6f} seconds")
print(f"Time to sum elements in the NumPy array: {array_time:.6f} seconds")
```



List in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316 INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2 Python for Data Science

LECTURE 1 *Introduction to Python Programming*

Understanding Lists in Python

Definition of Lists

- Lists are ordered collections of items
- Items can be of any data type (e.g., integers, strings, objects)
- Lists are mutable, allowing modification of elements

Creating Lists

- Lists are defined using square brackets `[]`
- Elements are separated by commas

Examples of List Creation:

- Empty list: `my_list = []`
- List of integers: `int_list = [1, 2, 3, 4, 5]`
- Mixed data types: `mixed_list = [1, "Hello", 3.14, True]`

Key Characteristics:

- Lists maintain the order of elements
- Elements can be accessed by their position (index)
- Lists can be nested (a list within another list)

Methods of Creating Lists in Python

Method	Description	Example
Literal Syntax	Directly listing items within square brackets.	<code>my_list = [1, 2, 3]</code>
<code>list()</code> Constructor	Using the list constructor for creating lists from iterables.	<code>my_list = list('abc')</code>
List Comprehension	Compact syntax for creating lists based on existing lists.	<code>squares = [x**2 for x in range(5)]</code>
Using <code>*</code> Operator	Create a list with repeated elements.	<code>zero_list = [0] * 10</code>
Using <code>split()</code>	Splitting a string into a list based on a delimiter.	<code>words = 'hello world'.split()</code>
Using <code>map()</code>	Applying a function to each item in an iterable and creating a list.	<code>lengths = list(map(len, ['hello', 'world']))</code>

Accessing Lists in Python

Accessing List Elements

- Elements in a list can be accessed using their index.
- Indices start at 0 for the first element.

Indexing

- Access a single element by its position.
- Example: `element = my_list[2]` (Accesses the third element)

Negative Indexing

- Access elements from the end of the list.
- Example: `last_element = my_list[-1]` (Accesses the last element)

Slicing

- Access a range of elements within a list.
- Example: `sub_list = my_list[1:4]` (Accesses elements at indices 1, 2, and 3)

Looping

- Access each element in the list by iterating over it.
- Example: `for item in my_list: print(item)`

```
python Copy code

# Define a list of fruits
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']

# Access by positive index
print("Positive Indexing:", fruits[2]) # Outputs 'cherry'

# Access by negative index
print("Negative Indexing:", fruits[-2]) # Outputs 'date'

# Access by slicing
print("Slicing:", fruits[1:4]) # Outputs ['banana', 'cherry', 'date']

# Access through looping
print("Looping:")
for fruit in fruits:
    print(fruit) # Outputs each fruit on a new line
```

- **Positive Indexing:** Directly access an element by its position.
- **Negative Indexing:** Useful for accessing elements relative to the end.
- **Slicing:** Extracts a part of the list as a new list.
- **Looping:** Useful for operations on each element.

List Operations – Adding and Removing Elements

Modifying Lists in Python

- Python lists are mutable, allowing for dynamic insertion and deletion of elements.

Adding Elements

- **Append:** Add an element to the end of the list.
 - Example: `fruits.append('fig')` (Adds 'fig' at the end)
- **Insert:** Insert an element at a specific index.
 - Example: `fruits.insert(1, 'grape')` (Inserts 'grape' at index 1)
- **Extend:** Add multiple elements from another iterable.
 - Example: `fruits.extend(['honeydew', 'kiwi'])` (Adds multiple items at the end)

Removing Elements

- **Remove:** Remove the first occurrence of a specified element.
 - Example: `fruits.remove('banana')` (Removes 'banana' from the list)
- **Pop:** Remove an element at a specified index and return it.
 - Example: `fruits.pop(2)` (Removes and returns the third element)
- **Clear:** Remove all elements from the list.
 - Example: `fruits.clear()` (Empties the list)

python

 Copy code

```
# Starting list
fruits = ['apple', 'banana', 'cherry']

# Add elements
fruits.append('date')
fruits.insert(1, 'elderberry')
fruits.extend(['fig', 'grape'])

# Remove elements
fruits.remove('banana')
popped_fruit = fruits.pop(2) # Typically, would print or use this value
fruits.clear()

# Print the list after operations
print(fruits)
```

- Use `append` to add single items and `extend` for multiple items.
- Use `insert` to add an item at a specific position.
- `remove`, `pop`, and `clear` provide different ways for element removal based on needs.

Exploring List Methods

Advanced List Methods in Python

- Lists offer a variety of built-in methods for easy and efficient manipulation.

Sorting and Reversing

- **Sort:** Organize elements in ascending or custom order.
 - Example: `fruits.sort()` (Sorts alphabetically)
 - Custom sort: `fruits.sort(key=len)` (Sorts by length of the element)
- **Reverse:** Reverse the order of elements.
 - Example: `fruits.reverse()` (Reverses the order of the list)

```
python Copy code

# Define a list of fruits
fruits = ['apple', 'banana', 'cherry', 'date', 'banana', 'apple']

# Sorting and reversing
fruits.sort()
fruits.reverse()

# Searching and counting
apple_index = fruits.index('apple')
banana_count = fruits.count('banana')

# Copying and converting
new_fruits = fruits.copy()
fruit_string = ', '.join(fruits)

# Output results
print(fruits)
print(f"Index of 'apple': {apple_index}")
print(f"Count of 'banana': {banana_count}")
print(f"Copy of fruits: {new_fruits}")
print(f"Fruit string: {fruit_string}")
```

- `sort` and `reverse` adjust the order of the list.
- `index` and `count` help find elements and their frequency.
- `copy` provides a way to duplicate lists without affecting the original.
- `join` is useful for creating strings from list elements, particularly handy in text processing.

Exploring List Methods

Searching and Counting

- **Index:** Find the first occurrence of an element and return its index.
 - Example: `index = fruits.index('apple')`
- **Count:** Return the number of times an element appears.
 - Example: `count = fruits.count('apple')`

Copying

- **Copy:** Return a shallow copy of the list.
 - Example: `new_fruits = fruits.copy()`

Conversion to Other Data Structures

- **Join:** Convert a list of strings into a single string with a separator.
 - Example: `fruit_string = ', '.join(fruits)`

```
python Copy code

# Define a list of fruits
fruits = ['apple', 'banana', 'cherry', 'date', 'banana', 'apple']

# Sorting and reversing
fruits.sort()
fruits.reverse()

# Searching and counting
apple_index = fruits.index('apple')
banana_count = fruits.count('banana')

# Copying and converting
new_fruits = fruits.copy()
fruit_string = ', '.join(fruits)

# Output results
print(fruits)
print(f"Index of 'apple': {apple_index}")
print(f"Count of 'banana': {banana_count}")
print(f"Copy of fruits: {new_fruits}")
print(f"Fruit string: {fruit_string}")
```

- `sort` and `reverse` adjust the order of the list.
- `index` and `count` help find elements and their frequency.
- `copy` provides a way to duplicate lists without affecting the original.
- `join` is useful for creating strings from list elements, particularly handy in text processing.

Exploring List Methods

```
['date', 'cherry', 'banana', 'banana', 'apple', 'apple']  
Index of 'apple': 4  
Count of 'banana': 2  
Copy of fruits: ['date', 'cherry', 'banana', 'banana', 'apple', 'apple']  
Fruit string: date, cherry, banana, banana, apple, apple
```

```
python Copy code  
  
# Define a list of fruits  
fruits = ['apple', 'banana', 'cherry', 'date', 'banana', 'apple']  
  
# Sorting and reversing  
fruits.sort()  
fruits.reverse()  
  
# Searching and counting  
apple_index = fruits.index('apple')  
banana_count = fruits.count('banana')  
  
# Copying and converting  
new_fruits = fruits.copy()  
fruit_string = ', '.join(fruits)  
  
# Output results  
print(fruits)  
print(f"Index of 'apple': {apple_index}")  
print(f"Count of 'banana': {banana_count}")  
print(f"Copy of fruits: {new_fruits}")  
print(f"Fruit string: {fruit_string}")
```

Python List Operations and Examples

Operation	Description	Example
<code>len()</code>	Get the number of items in a list.	<code>len(my_list)</code>
<code>sum()</code>	Calculate the total of all numerical items.	<code>sum(my_list)</code>
<code>all()</code>	Check if all elements are true (or if list is empty).	<code>all([item > 0 for item in my_list])</code>
<code>any()</code>	Check if any element is true.	<code>any([item > 0 for item in my_list])</code>
<code>sorted()</code>	Return a new list containing all elements in sorted order.	<code>sorted(my_list, reverse=True)</code>
<code>reversed()</code>	Return a reverse iterator.	<code>list(reversed(my_list))</code>
<code>enumerate()</code>	Return an enumerate object, adding a counter to an iterable.	<code>list(enumerate(my_list))</code>
<code>list()</code>	Create a list, typically from another iterable.	<code>list('hello')</code>
<code>zip()</code>	Combine elements from two or more iterables.	<code>list(zip(list1, list2))</code>

Python List Operations and Examples

```
1  # Define sample lists for demonstration
2  my_list = [5, 3, 9, 0, 2, 8, 6]
3  list1 = [1, 2, 3]
4  list2 = ['a', 'b', 'c']
5
6  # len() - Get the number of items in a list
7  print(f"Length of my_list: {len(my_list)}")
8
9  # sum() - Calculate the total of all numerical items
10 print(f"Sum of my_list: {sum(my_list)}")
11
12 # all() - Check if all elements are true (non-zero in this context)
13 print(f"All elements greater than 0: {all(item > 0 for item in my_list)}")
14
15 # any() - Check if any element is true (non-zero in this context)
16 print(f"Any element greater than 0: {any(item > 0 for item in my_list)}")
17
18
19
```

```
Length of my_list: 7
Sum of my_list: 33
All elements greater than 0: False
Any element greater than 0: True
```

Python List Operations and Examples

```
1  # Define sample lists for demonstration
2  my_list = [5, 3, 9, 0, 2, 8, 6]
3  list1 = [1, 2, 3]
4  list2 = ['a', 'b', 'c']
5
6  # sorted() - Return a new list containing all elements in sorted order
7  print(f"Sorted my_list: {sorted(my_list)}")
8  print(f"Reverse sorted my_list: {sorted(my_list, reverse=True)}")
9
10 # reversed() - Return a reverse iterator
11 print(f"Reversed my_list: {list(reversed(my_list))}")
12
13 # enumerate() - Return an enumerate object, adding a counter to an iterable
14 print("Enumerated my_list:")
15 for index, value in enumerate(my_list):
16     | print(f"Index {index}: Value {value}")
```

```
Sorted my_list: [0, 2, 3, 5, 6, 8, 9]
Reverse sorted my_list: [9, 8, 6, 5, 3, 2, 0]
Reversed my_list: [6, 8, 2, 0, 9, 3, 5]
Enumerated my_list:
Index 0: Value 5
Index 1: Value 3
Index 2: Value 9
Index 3: Value 0
Index 4: Value 2
Index 5: Value 8
Index 6: Value 6
```

Python List Operations and Examples

```
1  # Define sample lists for demonstration
2  my_list = [5, 3, 9, 0, 2, 8, 6]
3  list1 = [1, 2, 3]
4  list2 = ['a', 'b', 'c']
5
6  # list() - Create a list, typically from another iterable
7  print(f"List from string 'hello': {list('hello')}")
8
9  # zip() - Combine elements from two or more iterables
10 print("Zipped list1 and list2:")
11 for num, letter in zip(list1, list2):
12     | print(f"Number {num}, Letter '{letter}')
```

```
⇒ List from string 'hello': ['h', 'e', 'l', 'l', 'o']
Zipped list1 and list2:
Number 1, Letter 'a'
Number 2, Letter 'b'
Number 3, Letter 'c'
```

Weather Analytics Using Python Lists

Average Temperature: 21.7°C
Hottest Day: 25°C
Coldest Day: 18°C
Temperature Histogram
18°C: *
19°C: *
20°C: *
21°C: *
22°C: *
23°C: *
24°C: *
25°C: *

Overview of Application

- Utilize lists to store and analyze daily weather data.

Key Operations

1. Average Temperature Calculation

- Use `sum()` and `len()` to calculate the mean of the temperatures.
- Example: `average_temp = sum(temperatures) / len(temperatures)`

2. Identifying Extremes

- Determine the hottest and coldest days using `max()` and `min()`.
- Examples:
 - Hottest: `max_temp = max(temperatures)`
 - Coldest: `min_temp = min(temperatures)`

3. Histogram Generation

- Visual representation of temperature frequency.
- Function to generate a histogram showing how often each temperature occurred.

Comprehensive Example:

- Code snippet showing list manipulations to perform weather analysis.
- Histogram to visualize temperature distribution.

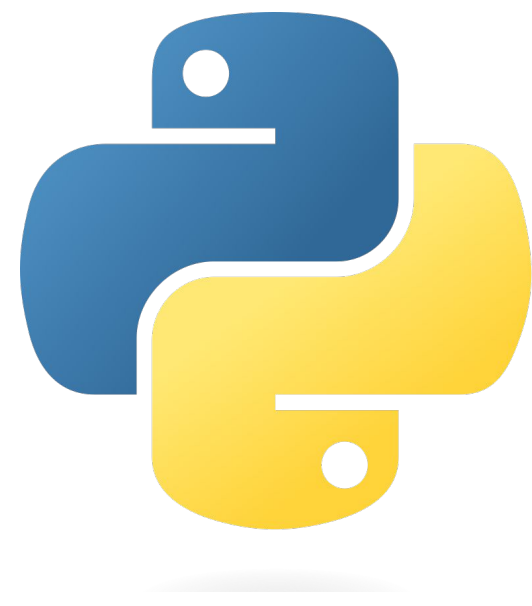
```
python
# Sample data: daily temperatures for a week
temperatures = [22, 24, 19, 23, 25, 21, 18]

# Calculate average temperature
average_temp = sum(temperatures) / len(temperatures)

# Find the maximum (hottest) and minimum (coldest) temperature
max_temp = max(temperatures)
min_temp = min(temperatures)

# Generate a simple histogram of temperatures
def generate_histogram(temps):
    print("Temperature Histogram")
    for temp in range(min(temps), max(temps) + 1):
        count = temps.count(temp)
        print(f"{temp}°C: {'*' * count}")

# Display results
print(f"Average Temperature: {average_temp:.1f}°C")
print(f"Hottest Day: {max_temp}°C")
print(f"Coldest Day: {min_temp}°C")
generate_histogram(temperatures)
```



Tuple in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Understanding Tuples in Python

Introduction to Tuples

- Tuples are immutable sequences used to store collections of items in Python.

Key Characteristics of Tuples

- **Immutable:** Once created, elements cannot be changed.
- **Ordered:** Maintains the order of elements, allowing indexed access.
- **Indexable:** Elements are accessed by their index, starting at 0.
- **Supports Duplicates:** Can contain multiple instances of the same value.
- **Heterogeneous:** Allows different data types within the same tuple.
- **Hashable:** Can be used as keys in dictionaries due to their immutability.

Creating Tuples

- Tuples are created by placing comma-separated values inside parentheses `( )`.
- For a single element, include a trailing comma, e.g., `singleton_tuple = ('hello',)`

Example Code

```
python Copy code

# Creating a tuple
my_tuple = ('apple', 'banana', 'cherry')

# Accessing elements
print(my_tuple[1]) # Outputs 'banana'

# Tuple with mixed data types
mixed_tuple = (1, 'Hello', 3.14, [1, 2, 3])

# Single element tuple
singleton_tuple = ('hello',)
```

Practical Uses

- Ideal for storing configurations or constant values.
- Commonly used for returning multiple values from functions.

Key Points

- Tuples are preferred for fixed data records and ensuring data integrity in multi-threaded environments.

Creating Tuples in Python

Creating Tuples

- Tuples are immutable sequences, ideal for storing fixed collections of items.

Methods of Creating Tuples

- **Standard Creation:**
 - Use parentheses with comma-separated values.
 - Example: `my_tuple = (1, 2, 3)`
- **Single Element Tuple:**
 - Include a comma after the item for a single element.
 - Example: `single_element_tuple = (1,)`
- **Implicit Creation:**
 - Parentheses are optional when the context is clear.
 - Example: `my_tuple = 1, 2, 3`
- **Tuple Constructor:**
 - Convert other iterables to a tuple using `tuple()`.
 - Example: `list_to_tuple = tuple([1, 2, 3])`
- **Unpacking:**
 - Create new tuples by unpacking elements from an existing tuple.

python

 Copy code

```
# Standard tuple creation using parentheses
my_tuple = (1, 2, 3)
print("Standard Tuple:", my_tuple)

# Creating a single element tuple
single_element_tuple = (1,)
print("Single Element Tuple:", single_element_tuple)

# Implicit tuple creation without parentheses
implicit_tuple = 1, 2, 3
print("Implicitly Created Tuple:", implicit_tuple)

# Using the tuple constructor to convert a list to a tuple
list_to_tuple = tuple([1, 2, 3])
print("Tuple from list using tuple constructor:", list_to_tuple)

# Unpacking from another tuple
original_tuple = (1, 2, 3)
a, b, c = original_tuple
new_tuple = (a, b, c)
print("New Tuple from Unpacking:", new_tuple)
```

Creating Tuples in Python

Explanation of the Code:

1. **Standard Tuple Creation:** The script creates a tuple using parentheses which is the most common method.
2. **Single Element Tuple:** It demonstrates how to create a tuple with just one element, which requires a comma to differentiate it from a simple parenthetical expression.
3. **Implicit Tuple Creation:** Shows how tuples can be created without explicit parentheses, which is useful in Python syntax especially during assignments or function returns.
4. **Using the Tuple Constructor:** Converts a list (or any iterable) into a tuple, showcasing the versatility of the tuple constructor.
5. **Unpacking from Another Tuple:** Illustrates how you can unpack values from one tuple directly into another, which is a common pattern in Python for swapping variables or splitting values.

python

 Copy code

```
# Standard tuple creation using parentheses
my_tuple = (1, 2, 3)
print("Standard Tuple:", my_tuple)

# Creating a single element tuple
single_element_tuple = (1,)
print("Single Element Tuple:", single_element_tuple)

# Implicit tuple creation without parentheses
implicit_tuple = 1, 2, 3
print("Implicitly Created Tuple:", implicit_tuple)

# Using the tuple constructor to convert a list to a tuple
list_to_tuple = tuple([1, 2, 3])
print("Tuple from list using tuple constructor:", list_to_tuple)

# Unpacking from another tuple
original_tuple = (1, 2, 3)
a, b, c = original_tuple
new_tuple = (a, b, c)
print("New Tuple from Unpacking:", new_tuple)
```

Accessing Elements in Tuples in Python

Overview of Tuple Access Methods

- Tuples, being ordered and immutable, can be accessed via indices, slices, or through iteration.

Access Methods

- **Indexing:**
 - Access by position using indices, starting from 0 or from -1 for reverse.
- **Slicing:**
 - Retrieve slices of the tuple for subsets of its elements.
- **Looping:**
 - Iterate through each element in a tuple, useful for applying operations to each item.

```
python Copy code

# Define a tuple
my_tuple = ('apple', 'banana', 'cherry')

# Indexing examples
print("First Element:", my_tuple[0]) # Outputs 'apple'
print("Last Element:", my_tuple[-1]) # Outputs 'cherry'

# Slicing example
print("Slice of Tuple:", my_tuple[1:3]) # Outputs ('banana', 'cherry')

# Looping through a tuple
for fruit in my_tuple:
    print(fruit)
```

Key Points

- Tuples are accessed similarly to lists but cannot be modified.
- Indexing and slicing are efficient ways to retrieve elements.
- Looping is ideal for processing each item in a tuple sequentially.

Key Methods for Tuples in Python

Overview of Tuple Methods

- While tuples are immutable and offer fewer methods compared to lists, there are essential functions that facilitate working with tuples effectively.

Method	Description	Example
<code>.count(x)</code>	Counts the number of times <code>x</code> appears in the tuple.	<pre>my_tuple = (1, 2, 2, 3) my_tuple.count(2)</pre>
<code>.index(x)</code>	Returns the index of the first occurrence of <code>x</code> .	<pre>my_tuple = (1, 2, 3) my_tuple.index(2)</pre>

Key Methods for Tuples in Python

- `.count(x)`
 - Use `.count()` to determine how frequently a particular item appears in a tuple. This can be useful for data analysis and checks before operations that depend on item frequency.
 - Example Code:

python

 Copy code

```
# Define the tuple
items = (1, 2, 3, 2, 2, 4)
# Count occurrences of 2
count_two = items.count(2)
print("Number of twos:", count_two) # Output: 3
```

Key Methods for Tuples in Python

- `.index(x)`
 - The `.index()` method is crucial for finding the position of an item in a tuple, especially when the arrangement or order of elements is significant in your code logic.
 - Example Code:

python

 Copy code

```
# Define the tuple
colors = ('red', 'blue', 'green', 'yellow')
# Find index of 'green'
index_green = colors.index('green')
print("Index of green:", index_green) # Output: 2
```

Immutability of Tuples in Python

Understanding Tuple Immutability

- Tuples are immutable data structures in Python, meaning once a tuple is created, its contents cannot be altered.

Characteristics of Tuple Immutability

- **Fixed Content:**
 - The elements and their order within a tuple cannot be changed after creation.
- **Safety:**
 - Immutability makes tuples safe to use as keys in dictionaries and elements in sets because they cannot be modified after creation.
- **Performance:**
 - Immutable objects can be more efficient in memory use and performance because Python can optimize storage and retrieval.

Example Code

```
python Copy code

# Define an immutable tuple
my_tuple = ('apple', 'banana', 'cherry')

# Attempt to modify the tuple (This will raise an error if uncommented)
# my_tuple[1] = 'orange' # Uncomment to see TypeError

# Legal operations that do not change the tuple
print("Count of 'apple':", my_tuple.count('apple'))
print("Index of 'banana':", my_tuple.index('banana'))
```

Key Points

- Tuples' immutability ensures that they are reliable and consistent in programs where fixed data is crucial.
- They are particularly useful in concurrent programming where immutable types prevent data races.

Implications of Immutability

Implications of Immutability

- **No Item Assignment:**

- Attempting to change an element directly results in a `TypeError`.
- Example:

```
python Copy code  
  
my_tuple = (1, 2, 3)  
# my_tuple[0] = 4 # Uncommenting this line will raise a TypeError
```

- **Fixed Methods:**

- Tuples have fewer available methods compared to lists, primarily methods that do not modify the tuple (like `.count()` and `.index()`).

- **Using Immutability:**

- Ideal for storing data that should not change over time, ensuring data integrity and preventing accidental data modification.

Example Code

```
python Copy code  
  
# Define an immutable tuple  
my_tuple = ('apple', 'banana', 'cherry')  
  
# Attempt to modify the tuple (This will raise an error if uncommented)  
# my_tuple[1] = 'orange' # Uncomment to see TypeError  
  
# Legal operations that do not change the tuple  
print("Count of 'apple':", my_tuple.count('apple'))  
print("Index of 'banana':", my_tuple.index('banana'))
```

Key Points

- Tuples' immutability ensures that they are reliable and consistent in programs where fixed data is crucial.
- They are particularly useful in concurrent programming where immutable types prevent data races.

When to Use Tuples Over Lists in Python

Choosing Between Tuples and Lists

- Understanding when to use tuples instead of lists can optimize both the performance and the reliability of your Python programs.

Advantages of Using Tuples

- **Immutability:**
 - Use tuples when you have a collection of items that should not change throughout the program to ensure data integrity and prevent accidental modifications.
- **Hashable:**
 - Tuples can be used as keys in dictionaries or as elements in sets because they are immutable, unlike lists.
- **Performance:**
 - Tuples can be slightly more memory efficient and faster in some contexts due to Python's internal optimizations for immutable objects.

- **Data Packing:**

- Tuples are ideal for data packing and unpacking, allowing easy assignment and manipulation of multiple variables in a single statement.

- Example:

```
python Copy code  
  
data = (100, "Item", 1.23) # Packing  
id, name, price = data    # Unpacking
```

- **Function Return Values:**

- Tuples are commonly used to return multiple values from a function.

- Example:

```
python Copy code  
  
def min_max(items):  
    return min(items), max(items)
```

When to Prefer Lists

- **Mutability:**
 - Use lists if you need to modify the contents, such as adding, removing, or changing elements.
- **Extensive Functionality:**
 - Lists offer a wide range of methods for manipulation, making them versatile for operations that involve frequent modification.
- **Iteration Heavy Operations:**
 - If the data is primarily used for iterative processes that involve modification or ordering changes, lists are more appropriate.

Example Code

```
python Copy code  
  
# Tuple for fixed set of elements  
coordinates = (10.0, 20.0)  
  
# List for modifiable collection of elements  
inventory = ["sword", "armor"]  
inventory.append("shield")
```

Key Points

- Choose tuples for fixed-size collections of items and when you require hashability.
- Opt for lists when the flexibility of data manipulation is needed.

Unpacking Tuples in Python

Exploring Tuple Unpacking

- Tuple unpacking allows for the assignment of tuple elements to named variables in a single operation, which is a concise and readable way to handle multiple data items simultaneously.

Key Features of Tuple Unpacking

- **Direct Assignment:**
 - Each variable on the left side of the assignment corresponds to an element in the tuple based on its position.
- **Example:**

python

 Copy code

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print("a:", a, "b:", b, "c:", c) # Outputs a: 1 b: 2 c: 3
```

Unpacking Tuples in Python

- Extended Unpacking (Python 3.x):
 - Use an asterisk `*` to assign excess items to a list variable.
 - Example:

```
python Copy code  
  
my_tuple = (1, 2, 3, 4, 5)  
first, *middle, last = my_tuple  
print("First:", first, "Middle:", middle, "Last:", last) # Outputs First:
```

Unpacking Tuples in Python

- Ignoring Elements:
 - Use underscores `_` or another naming convention to ignore specific elements during unpacking.
- Example:

python

 Copy code

```
my_tuple = (1, 2, 3, 4)
_, y, _, z = my_tuple
print("y:", y, "z:", z) # Outputs y: 2 z: 4
```

Practical Uses of Tuple Unpacking

- **Multiple Return Values:**

- Useful for functions that return multiple values wrapped in a tuple.
- Example:

```
python Copy code  
  
def get_dimensions():  
    return (1080, 1920)  
height, width = get_dimensions()
```

- **Data Iteration:**

- When iterating through a list of tuples, unpack each tuple into variables for clearer and more accessible data handling.
- Example:

```
python Copy code  
  
list_of_tuples = [(1, 'apple'), (2, 'banana')]  
for index, fruit in list_of_tuples:  
    print("Index:", index, "Fruit:", fruit)
```

Practical Uses of Tuple Unpacking

Example Code

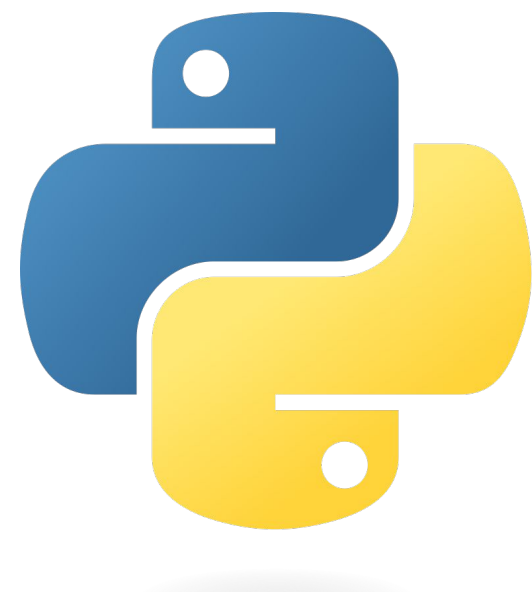
```
python Copy code

# Tuple for unpacking demo
coordinates = (10.0, 20.0, 30.0)
x, y, z = coordinates
print("Coordinates:", x, y, z)

# Using extended unpacking
data = ("red", "blue", "green", "yellow")
color1, *other_colors, color_last = data
print("First Color:", color1, "Other Colors:", other_colors, "Last Color:", color_
```

Key Points

- Tuple unpacking enhances code readability and efficiency, especially when handling multiple data items simultaneously.
- Extended unpacking is particularly useful for dealing with tuples containing variable numbers of elements.



Dictionary in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316
**INTRODUCTION TO AI AND DATA
SCIENCE**

CHAPTER 2
Python for Data Science

LECTURE 1
Introduction to Python Programming

Understanding Dictionaries in Python

Key Features of Dictionaries

- **Key-Value Storage:**

- Each item in a dictionary is stored as a key-value pair, where keys must be unique and hashable (typically strings, numbers, or tuples that contain only immutable elements).

- **Example:**

python

 Copy code

```
my_dict = {'name': 'John', 'age': 30}
```

- **Mutable:**

- Dictionaries can be changed by adding, removing, and modifying their key-value pairs.

- **Example:**

python

 Copy code

```
my_dict['age'] = 31 # Modifying an existing key  
my_dict['address'] = '123 Street' # Adding a new key
```

Dictionaries in Python are mutable data structures used to store data values in key-value pairs. They are highly efficient for looking up, adding, and removing data, thanks to their implementation using hash tables.

- Dictionaries are ideal for cases where items need to be retrieved quickly by a key.
- They are commonly used to represent objects' properties, perform database-like queries, and manage diverse data sets dynamically.

Understanding Dictionaries in Python

- **Dynamic:**
 - The size of a dictionary can grow or shrink as needed, with no need to define an initial size.
- **Unordered (Python < 3.7), Ordered (Python >= 3.7):**
 - In Python versions 3.7 and later, dictionaries maintain the insertion order of items, meaning items will be iterated in the order they were added.
 - In earlier versions, dictionaries are unordered, and the order of items is arbitrary.

Understanding Dictionaries in Python

- Access by Key:
 - Values are accessed using their keys, providing a very efficient way to lookup data.
 - Example:

python

 Copy code

```
print(my_dict['name']) # Outputs 'John'
```

Understanding Dictionaries in Python

- Iterability:

- Dictionaries can be iterated through directly or via their methods like `.keys()`, `.values()`, and `.items()`.

- Example:

python

 Copy code

```
for key, value in my_dict.items():  
    print(key, value)
```

Key-Value Pairing in Python Dictionaries

What Are Key-Value Pairs?

- Key-value pairs are the fundamental elements of a Python dictionary, where each **key** acts as a unique identifier linked to a **value**.

Key Characteristics

- **Immutable Keys:** Each key must be immutable and hashable, typically strings or numbers.
- **Value Flexibility:** Values can be any data type, including lists, tuples, or other dictionaries.

Efficiency of Key Access

- Accessing values through keys is extremely efficient, leveraging hash tables behind the scenes for quick retrieval.

Key-Value Pairing in Python Dictionaries

What Are Key-Value Pairs?

- Key-value pairs are the fundamental elements of a Python dictionary, where each **key** acts as a unique identifier linked to a **value**.

Key Characteristics

- **Immutable Keys:** Each key must be immutable and hashable, typically strings or numbers.
- **Value Flexibility:** Values can be any data type, including lists, tuples, or other dictionaries.

Efficiency of Key Access

- Accessing values through keys is extremely efficient, leveraging hash tables behind the scenes for quick retrieval.

Example of Key-Value Pair

python

 Copy code

```
user_info = {'name': 'Alice', 'age': 28}
print(user_info['name']) # Outputs 'Alice'
```

Key-Value Pairing in Python Dictionaries

How to Modify Dictionary Values

- Values in dictionaries are mutable, allowing changes directly via their keys.

Direct Value Modification

- Update or change values by accessing them through their corresponding keys.
- Example:

python

 Copy code

```
user_info['age'] = 29
print("Updated Age:", user_info['age']) # Outputs 29
```

Adding New Key-Value Pairs

- Extend dictionaries by assigning values to new keys.
- Example:

python

 Copy code

```
user_info['gender'] = 'Female'
print("Added Gender:", user_info['gender']) # Outputs 'Female'
```

Key-Value Pairing in Python Dictionaries

Dictionary Order in Recent Python Versions

- **Order Maintained (Python 3.7+):** Dictionaries preserve insertion order, making data operations predictable.

Example of Ordered Dictionary

python

 Copy code

```
product = {'id': 101, 'name': 'Chair', 'price': 25.50}
product['color'] = 'blue'
for key, value in product.items():
    print(key, value)
# Outputs in the order: id, name, price, color
```

Impact of Ordering

- The order retention simplifies tasks like data serialization and comparison by keeping elements in a consistent sequence.

Accessing Values in Python Dictionaries

Access by Key

- Use the key in square brackets to retrieve the corresponding value.
- Example:

```
python Copy code  
  
settings = {'volume': 75, 'brightness': 50}  
current_volume = settings['volume']  
print("Current Volume:", current_volume) # Outputs 75
```

Handling Missing Keys

- Attempting to access a non-existent key results in a `KeyError`.
- Use `.get()` method for a safer access, which allows a default value if the key is absent.
- Example:

```
python Copy code  
  
current_contrast = settings.get('contrast', 'No contrast setting')  
print(current_contrast) # Outputs 'No contrast setting'
```

Accessing Values in Python Dictionaries

Access by Key

- Use the key in square brackets to retrieve the corresponding value.
- Example:

```
python Copy code  
  
settings = {'volume': 75, 'brightness': 50}  
current_volume = settings['volume']  
print("Current Volume:", current_volume) # Outputs 75
```

Handling Missing Keys

- Attempting to access a non-existent key results in a `KeyError`.
- Use `.get()` method for a safer access, which allows a default value if the key is absent.
- Example:

```
python Copy code  
  
current_contrast = settings.get('contrast', 'No contrast setting')  
print(current_contrast) # Outputs 'No contrast setting'
```

Impact of Direct Key Access

- Direct key access is **fast** and **efficient**, making dictionaries ideal for settings and configurations.

Setting and Updating Values

How to Set and Update Dictionary Values

- Values can be set or updated by assigning a value to a key directly.

Setting New Values

- Assign a value to a new or existing key to update the dictionary.
- Example:

```
python Copy code  
  
settings['balance'] = 100  
print("Balance Setting:", settings['balance']) # Outputs 100
```

Updating Multiple Values

- Use `.update()` method to change multiple values at once.
- Example:

```
python Copy code  
  
settings.update({'volume': 80, 'mode': 'night'})  
print("Updated Settings:", settings) # Outputs updated dictionary
```

Impact of Value Setting

- Setting values is **straightforward** and **flexible**, making dictionaries highly adaptable for dynamic data management.

Key Dictionary Methods in Python

Commonly Used Dictionary Methods

- `.keys()` :
 - Retrieves a view of keys in the dictionary.
 - Example:

```
python Copy code  
  
config = {'width': 1920, 'height': 1080}  
print("Keys:", config.keys()) # Outputs dict_keys(['width', 'height'])
```

- `.values()` :
 - Retrieves a view of values in the dictionary.
 - Example:

```
python Copy code  
  
print("Values:", config.values()) # Outputs dict_values([1920, 1080])
```

Key Dictionary Methods in Python

- `.items()` :
 - Returns a view of key-value pairs (tuples) in the dictionary.
 - Example:

python

 Copy code

```
for key, value in config.items():  
    print(key, value) # Outputs each key and value
```

- `.get(key, default=None)` :
 - Safely retrieves the value for a given key, returns `default` if the key is not found.
 - Example:

python

 Copy code

```
aspect_ratio = config.get('aspect_ratio', '16:9')  
print("Aspect Ratio:", aspect_ratio) # Outputs '16:9'
```

Key Dictionary Methods in Python

- `.update(other)` :

- Updates the dictionary with key-value pairs from another dictionary or an iterable of key-value pairs.

- Example:

python

 Copy code

```
config.update({'color': 'black', 'aspect_ratio': '16:9'})  
print("Updated Config:", config)
```

- `.pop(key, default=None)` :

- Removes the item with the specified key and returns its value. If the key is not found, it returns `default`.

- Example:

python

 Copy code

```
height = config.pop('height', 1080)  
print("Popped Height:", height) # Outputs the value of 'height'
```

Key Dictionary Methods in Python

- `.pop(key, default=None)` :

- Removes the item with the specified key and returns its value. If the key is not found, it returns `default`.

- Example:

python

 Copy code

```
height = config.pop('height', 1080)
print("Popped Height:", height) # Outputs the value of 'height'
```

- `.clear()` :

- Clears all the items from the dictionary, making it empty.

- Example:

python

 Copy code

```
config.clear()
print("Cleared Config:", config) # Outputs {}
```

Iterating Over Dictionaries in Python

Methods for Iterating Over Dictionaries

- Iterating Over Keys:

- By default, iterating over a dictionary traverses its keys.

- Example:

```
python Copy code  
  
user_profile = {'name': 'John', 'age': 30, 'location': 'New York'}  
for key in user_profile:  
    print(key) # Outputs 'name', 'age', 'location'
```

- Iterating Over Values:

- Use `.values()` to iterate over the values directly.

- Example:

```
python Copy code  
  
for value in user_profile.values():  
    print(value) # Outputs 'John', 30, 'New York'
```

Iterating Over Dictionaries in Python

- Iterating Over Key-Value Pairs:
 - Use `.items()` to get tuples of key-value pairs for simultaneous iteration.
 - Example:

```
python Copy code  
  
for key, value in user_profile.items():  
    print(key, value) # Outputs 'name John', 'age 30', 'location New York'
```

- Conditional Iteration:
 - Combine dictionary methods with conditionals to filter results.
 - Example:

```
python Copy code  
  
# Print only numerical values  
for key, value in user_profile.items():  
    if isinstance(value, int):  
        print(key, value) # Outputs 'age 30'
```

Iterating Over Dictionaries in Python

Use Cases and Efficiency

- Iterating over dictionaries is particularly useful in scenarios where data relationships (key to value) need to be maintained.
- Efficient data access and manipulation during iteration make dictionaries ideal for data-heavy applications.

Example Code

```
python Copy code

# Example dictionary
config_settings = {'resolution': '1920x1080', 'volume': 75, 'brightness': 50}

# Iterate and print each setting
for setting, value in config_settings.items():
    print(f"Setting: {setting}, Value: {value}")
```

Key Points

- Iterating over dictionaries allows for flexible data retrieval and manipulation, making it a fundamental technique in Python programming.
- Utilize different iteration methods based on whether keys, values, or both are needed for processing.

Use Cases for Dictionaries in Data Handling

Primary Use Cases for Dictionaries

- Data Storage and Retrieval:

- Store data in a structured format with key-based access, ideal for fast lookups and updates.
- Example:

```
python Copy code  
  
employee_record = {'id': 101, 'name': 'Jane Doe', 'role': 'Developer'}
```

- Configuration Settings:

- Manage application settings or user preferences that require quick access and potential updates.
- Example:

```
python Copy code  
  
app_config = {'theme': 'dark', 'notifications': True, 'language': 'English'}
```

Use Cases for Dictionaries in Data Handling

- Caching Mechanisms:

- Implement caching strategies where results of costly operations are stored and retrieved using unique keys to improve performance.

- Example:

```
python Copy code

cache = {}
def get_data(query):
    if query not in cache:
        result = expensive_data_operation(query)
        cache[query] = result
    return cache[query]
```

- Mapping Relationships:

- Map keys to values to represent relationships, such as linking product IDs to descriptions or names.

- Example:

```
python Copy code

product_map = {1001: 'Apple iPhone', 1002: 'Samsung Galaxy'}
```

Use Cases for Dictionaries in Data Handling

- JSON Data Handling:

- Serialize and deserialize JSON data for web services and APIs, as JSON directly maps to dictionaries.

- Example:

```
python Copy code  
  
import json  
json_data = json.dumps(employee_record) # Serialize  
data_object = json.loads(json_data) # Deserialize
```

- Dynamic Data Access:

- Use dictionaries to dynamically access and update data based on user input or program logic.

- Example:

```
python Copy code  
  
user_input = input("Enter the setting to update (theme/notifications/language)  
new_value = input("Enter new value: ")  
app_config[user_input] = new_value
```

Use Cases for Dictionaries in Data Handling

Example Code

```
python Copy code

# Creating a simple user database
users = {
    1: {'name': 'Alice', 'email': 'alice@example.com'},
    2: {'name': 'Bob', 'email': 'bob@example.com'}
}

# Accessing data
print(users[1]['name']) # Outputs 'Alice'

# Updating data
users[1]['email'] = 'newalice@example.com'
print("Updated Email:", users[1]['email'])
```

Key Points

- Dictionaries are fundamental for efficient data manipulation, making them invaluable in a wide range of programming scenarios from web development to system configuration.

Dictionaries vs. JSON in Python

- Both dictionaries and JSON (JavaScript Object Notation) are used to store and manipulate data in a structured format, but they are utilized in different contexts and environments.

Python Dictionaries

- **Native Data Structure:** Built into Python, used for in-memory data storage.
- **Mutable:** Allows changes to data after creation.
- **Key Types:** Supports any immutable type as keys (strings, numbers, tuples).
- **Use Cases:** Ideal for program internal data handling and operations, including configuration settings, data caching, and more.

JSON

- **Data Interchange Format:** Lightweight text-based format, used primarily for data interchange between servers and web applications.
- **Immutable:** Represented as a string in its raw form.
- **Key Types:** Only string keys are allowed.
- **Use Cases:** Perfect for data transmission over the network, configuration files, and storing data persistently in a readable format.

Dictionaries vs. JSON in Python

1. Python Dictionary Syntax:

- In Python, dictionary keys can be any immutable type and are typically represented using either single quotes (`'`) or double quotes (`"`), though the former is more commonly seen in Python code.

2. JSON Syntax:

- JSON, which stands for JavaScript Object Notation, strictly requires double quotes (`"`) for wrapping keys and string values. This requirement is based on JavaScript's object literal notation but is enforced in JSON to maintain consistency across environments that parse JSON data.
- JSON does not support single quotes (`'`) for delimiting keys or values, which is a common source of confusion for those transitioning between JavaScript (where both are often used interchangeably) and JSON.

Example to Illustrate:

- Correct JSON:

```
json
{"name": "Alice", "age": 25}
```

- Incorrect JSON (common mistake):

```
json
{'name': 'Alice', 'age': 25}
```

Why It Matters:

- If you attempt to parse or validate JSON data using single quotes in tools, libraries, or services expecting proper JSON format, it will result in errors or invalid data processing. Thus, when dealing with JSON, always use double quotes.
- In Python dictionaries, both are permissible, and Python's `json` module can serialize dictionaries using double quotes to ensure the output is valid JSON:

```
python
import json
user_info = {'name': 'Alice', 'age': 25}
json_string = json.dumps(user_info)
print(json_string) # Outputs: {"name": "Alice", "age": 25}
```

The strict use of double quotes in JSON ensures compatibility across diverse systems and programming languages, adhering to a standardized format that simplifies data interchange.

Dictionaries vs. JSON in Python

Key Differences

- **Syntax:**
 - Dictionaries use Python syntax (`{'key': value}`), while JSON uses a syntax similar to JavaScript (`{"key": "value"}`).
- **Data Types:**
 - JSON supports basic data types (strings, numbers, arrays, booleans, and null), while dictionaries can hold more complex data types like Python objects.
- **Parsing and Serialization:**
 - JSON data must be parsed to be used in Python as dictionaries, and dictionaries often need to be serialized into JSON format for data interchange.
- **Example:**

```
python Copy code  
  
import json  
json_string = '{"name": "John", "age": 30}'  
user_dict = json.loads(json_string) # Convert JSON string to dictionary  
print(user_dict)  
back_to_json = json.dumps(user_dict) # Convert dictionary back to JSON str
```

Dictionaries vs. JSON in Python

Example Code

```
python Copy code

# Python Dictionary
user_info = {'name': 'Alice', 'age': 25, 'member': True}

# Convert to JSON
import json
user_json = json.dumps(user_info)
print("JSON String:", user_json)

# Convert back to Dictionary
user_from_json = json.loads(user_json)
print("Dictionary from JSON:", user_from_json)
```

Key Points

- Use **dictionaries** for internal data manipulation within Python programs.
- Use **JSON** for sending or receiving data to/from a web client/server or for configuration files.

Comparing Python Dictionaries and JSON

Dictionaries vs. JSON: Key Differences

Feature	Python Dictionary	JSON
Type	Native data structure in Python	Text-based data interchange format
Mutability	Mutable (can be changed)	Immutable (represented as a string)
Key Types	Supports any immutable type (strings, numbers, tuples)	Only string keys allowed
Data Types Supported	Can contain various Python types including objects	Limited to strings, numbers, arrays, booleans, null
Use in Applications	Internal data handling within Python programs	Data interchange between systems, web APIs
Syntax	Python syntax: <code>{'key': value}</code>	JSON syntax: <code>{"key": "value"}</code>
Parsing/Serialization	Not required for internal use	Must be parsed to/from string for use
Typical Use Cases	Data manipulation, configuration settings, caching	Web data transmission, configuration files

Comparing Python Dictionaries and JSON

Understanding Through Example

```
python Copy code

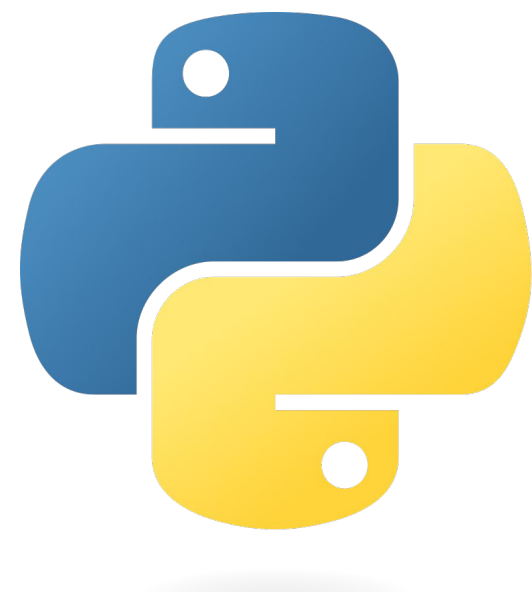
# Python Dictionary Example
user_info = {'name': 'Alice', 'age': 25, 'member': True}

# Converting Dictionary to JSON
import json
user_json = json.dumps(user_info)
print("JSON String:", user_json) # Output: '{"name": "Alice", "age": 25, "member": true}'

# Converting JSON to Dictionary
user_from_json = json.loads(user_json)
print("Dictionary from JSON:", user_from_json) # Output: {'name': 'Alice', 'age': 25, 'member': True}
```

Key Points

- **Python dictionaries** are ideal for flexible and dynamic data handling directly within Python.
- **JSON** is essential for structured data exchange across different programming environments, particularly in web contexts.



Set in Python

Prof. Anis Koubaa

- Introduction to Python's functional programming paradigm.
- Understanding the difference between functional and other paradigms (imperative, object-oriented).
- Motivation for using anonymous functions, higher-order functions, and Python's built-in data structures.



SPECIALIZATION

CS316 INTRODUCTION TO AI AND DATA SCIENCE

CHAPTER 2 Python for Data Science

LECTURE 1 *Introduction to Python Programming*

Introduction to Sets in Python

Exploring the Core Concepts of Sets

- Sets are one of the built-in data structures in Python, designed to store unique elements in an **unordered** and **unindexed** collection.

Characteristics of Sets

- **Unordered:** The elements in a set do not have a defined order; they may appear in a different order every time you use them.
- **Unindexed:** You cannot access items in a set by referring to an index or a key.
- **No Duplicates:** Sets automatically remove duplicate elements, storing only unique items.
- **Mutable:** You can add or remove items from a set, but you cannot change the items already in the set.

Efficiency of Sets

- Sets are highly efficient for membership testing, making them ideal for checking whether an item exists in a set.

python

 Copy code

```
# Define a set of fruits
fruit_set = {'apple', 'banana', 'cherry'}
print("Original Set:", fruit_set)

# Add an element to the set
fruit_set.add('date')
print("After Adding:", fruit_set)

# Remove an element from the set
fruit_set.remove('banana')
print("After Removal:", fruit_set)
```

Methods of Creating Sets in Python

Method	Description	Example
Curly Braces	Directly using curly braces with elements separated by commas.	<pre>my_set = {1, 2, 3}</pre>
<code>set()</code> Constructor	Using the <code>set()</code> constructor, often to create an empty set or convert other iterables into a set.	<pre>my_set = set([1, 2, 3])</pre>
Set Comprehension	Similar to list comprehensions, but for sets, useful for creating sets with conditional logic.	<pre>my_set = {x ** 2 for x in range(5)}</pre>
<code>fromkeys()</code>	Typically used with dictionaries, but can be adapted to create sets by using the keys from a dictionary.	<pre>my_set = set(dict.fromkeys([1, 2, 3]))</pre>

Methods of Creating Sets in Python

python

Copy code

```
# Creating a set using Curly Braces
fruit_set = {'apple', 'banana', 'cherry'}
print("Set created with Curly Braces:", fruit_set)

# Creating a set using the set() Constructor
number_set = set([1, 2, 3, 4, 5])
print("Set created with set() Constructor from a list:", number_set)

# Creating a set using Set Comprehension
square_set = {x ** 2 for x in range(5)}
print("Set created with Set Comprehension:", square_set)

# Creating a set from a dictionary's keys using fromkeys()
# Normally used to create dictionaries but adapted for sets here
temp_dict = dict.fromkeys(['hello', 'world', 'hello']) # Duplicate 'hello' to demonstrate uniqueness
unique_set_from_keys = set(temp_dict)
print("Set created from dictionary keys using fromkeys():", unique_set_from_keys)

# Additional demonstration of creating a set from a string to highlight uniqueness
string_set = set('hello')
print("Set created from a string (notice the uniqueness and unordered nature):", string_set)
```

```
Set created with Curly Braces: {'cherry', 'apple', 'banana'}
Set created with set() Constructor from a list: {1, 2, 3, 4, 5}
Set created with Set Comprehension: {0, 1, 4, 9, 16}
Set created from dictionary keys using fromkeys(): {'world', 'hello'}
Set created from a string (notice the uniqueness and unordered nature): {'h', 'l', 'e', 'o'}
```

Accessing Sets in Python

Accessing Set Elements

Unlike lists, sets in Python do not maintain any order, so elements cannot be accessed by an index. However, there are specific ways to work with sets effectively:

Characteristics of Sets

- **Unordered:** No index is associated with any element.
- **Unique:** Each element in a set is distinct.

Iterating Over a Set

- **Looping:** The primary method to go through elements in a set.
- Example:

```
python Copy code

my_set = {'apple', 'banana', 'cherry'}
for fruit in my_set:
    print(fruit) # Order of output is not predictable
```

Membership Testing

- **Checking Existence:** Use the `in` keyword to test if an element is part of the set. This operation is very efficient in sets.

- Example:

```
python Copy code

if 'banana' in my_set:
    print('banana is in the set')
```

Key Takeaways

- Sets do not support direct element access using indices or slicing because they are inherently unordered.
- The best way to access elements in a set is by iterating over the set or by testing for membership.

Set Methods in Python

1. Adding Elements

- `.add(element)`
 - Adds a single element to the set.
 - Does nothing if the element is already in the set.
 - **Example:**

python

 Copy code

```
my_set = {'apple', 'banana'}  
my_set.add('cherry')  
print("After adding:", my_set) # Output: {'apple', 'banana', 'cherry'}
```

Set Methods in Python

2. Removing Elements

- `.remove(element)`
 - Removes an element from the set.
 - Raises a `KeyError` if the element is not found.
 - **Example:**

python

 Copy code

```
my_set.remove('banana')
print("After removal:", my_set) # Output: {'apple', 'cherry'}
```

- `.discard(element)`
 - Similar to `.remove()`, but does not raise an error if the element is not present.
 - **Example:**

python

 Copy code

```
my_set.discard('banana') # No output, no error
```

Set Methods in Python

3. Clearing All Elements

- `.clear()`
 - Removes all elements from the set, resulting in an empty set.
 - Example:

python

 Copy code

```
my_set.clear()  
print("Set after clear:", my_set)  # Output: set()
```

Set Methods in Python

4. Checking Membership

- Not a method, but using `in` keyword
 - Efficiently checks if an element is in the set.
- Example:

python

 Copy code

```
if 'apple' in my_set:  
    print('apple is in the set')
```

Set Methods in Python

5. Copying Sets

- `.copy()`
 - Returns a shallow copy of the set.
 - Example:

python

 Copy code

```
new_set = my_set.copy()  
print("Copied Set:", new_set)
```

Set Methods in Python

Advanced Operations

- Set Algebra:
 - `.union()`, `.intersection()`, `.difference()`, `.symmetric_difference()`
 - Perform mathematical set operations.
- Example:

```
python Copy code  
  
set1 = {'apple', 'orange'}  
set2 = {'banana', 'orange'}  
print("Union:", set1.union(set2)) # Output: {'apple', 'banana', 'orange'}
```

List Operations – union

Overview of Set Operations

- Sets in Python support several operations that allow you to perform mathematical set operations efficiently. These operations are useful for comparing sets and deriving new sets based on existing ones.

- Union

- Combines all elements from two sets, excluding any duplicates.
- Operator: `|`
- Method: `.union()`
- Example:

python

 Copy code

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2 # Alternatively: set1.union(set2)
print("Union:", union_set) # Output: {1, 2, 3, 4, 5}
```

List Operations – intersection

- Intersection
 - Returns only the elements that are present in both sets.
 - Operator: `&`
 - Method: `.intersection()`
 - Example:

python

 Copy code

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1 & set2 # Alternatively: set1.intersection(set2)
print("Intersection:", intersection_set) # Output: {3}
```

List Operations – difference

- Difference
 - Returns the elements that are in the first set but not in the second.
 - Operator: `-`
 - Method: `.difference()`
 - Example:

python

 Copy code

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1 - set2 # Alternatively: set1.difference(set2)
print("Difference:", difference_set) # Output: {1, 2}
```

List Operations – symmetric difference

Symmetric Difference

- Returns elements that are in either of the sets but not in both.
- Operator: `^`
- Method: `.symmetric_difference()`
- Example:

```
python Copy code  
  
set1 = {1, 2, 3}  
set2 = {3, 4, 5}  
symmetric_difference_set = set1 ^ set2 # Alternatively: set1.symmetric_dif  
print("Symmetric Difference:", symmetric_difference_set) # Output: {1, 2,
```

Practical Uses of Sets in Python

1. Removing Duplicates

- Sets automatically eliminate any duplicate entries, making them ideal for cleaning data.
- Example:

python

 Copy code

```
data = [1, 2, 2, 3, 4, 4, 4, 5]
unique_data = list(set(data))
print("Unique Data:", unique_data) # Output: [1, 2, 3, 4, 5]
```

Leveraging Sets for Efficient Data Handling

- Sets in Python are powerful tools for managing data due to their unique properties of storing only unique elements and providing fast membership tests.

Practical Uses of Sets in Python

2. Membership Testing

- Sets offer a highly efficient way to test whether an element exists within them, thanks to their hashing mechanism.
- Example:

python

 Copy code

```
permissions = {'read', 'write', 'delete'}
if 'edit' in permissions:
    print("Edit permission granted.")
else:
    print("Edit permission not granted.") # This will be printed.
```

Practical Uses of Sets in Python

3. Set Operations for Data Analysis

- Perform mathematical set operations like union, intersection, and difference to analyze and compare datasets.
- Example:

python

 Copy code

```
employees = {'John', 'Jane', 'Mary', 'Bob'}  
admin_users = {'Jane', 'Mary', 'Sam'}  
regular_users = employees.difference(admin_users)  
print("Regular Users:", regular_users) # Output: {'John', 'Bob'}
```

Practical Uses of Sets in Python

4. Filtering Data

- Use set operations to filter out unwanted data or to apply conditions across large datasets.
- Example:

python

 Copy code

```
ages = {23, 45, 31, 55, 31, 23}
adult_ages = {age for age in ages if age >= 18}
print("Adult Ages:", adult_ages) # Output: {45, 55, 23, 31}
```

Practical Uses of Sets in Python

5. Creating Subsets for Tests and Experiments

- Quickly generate subsets of data for testing hypotheses or running experiments without repetitions.
- Example:

python

 Copy code

```
full_dataset = set(range(100)) # A set of numbers from 0 to 99
experimental_group = set(range(0, 100, 2)) # Even numbers as a subset
control_group = full_dataset.difference(experimental_group)
print("Control Group:", control_group) # Output: {1, 3, 5, ..., 99}
```

END OF LECTURE



Python Basics

Python Cheat Sheet for Beginners

Learn Python online at www.DataCamp.com

> How to use this cheat sheet

Python is the most popular programming language in data science. It is easy to learn and comes with a wide array of powerful libraries for data analysis. This cheat sheet provides beginners and intermediate users a guide to starting using python. Use it to jump-start your journey with python. If you want more detailed Python cheat sheets, check out the following cheat sheets below:



> Accessing help and getting object types

```
1 + 1 # Everything after the hash symbol is ignored by Python
help(max) # Display the documentation for the max function
type('a') # Get the type of an object - this returns str
```

> Importing packages

Python packages are a collection of useful tools developed by the open-source community. They extend the capabilities of the python language. To install a new package (for example, pandas), you can go to your command prompt and type in `pip install pandas`. Once a package is installed, you can import it as follows.

```
import pandas # Import a package without an alias
import pandas as pd # Import a package with an alias
from pandas import DataFrame # Import an object from a package
```

> The working directory

The working directory is the default file path that python reads or saves files into. An example of the working directory is `"C:/file/path"`. The `os` library is needed to set and get the working directory.

```
import os # Import the operating system package
os.getcwd() # Get the current directory
os.chdir("new/working/directory") # Set the working directory to a new file path
```

> Operators

Arithmetic operators

<code>102 + 37</code> # Add two numbers with +	<code>22 // 7</code> # Integer divide a number with //
<code>102 - 37</code> # Subtract a number with -	<code>3 ** 4</code> # Raise to the power with **
<code>4 * 6</code> # Multiply two numbers with *	<code>22 % 7</code> # Returns 1 # Get the remainder after division with %
<code>22 / 7</code> # Divide a number by another with /	

Assignment operators

```
a = 5 # Assign a value to a
x[0] = 1 # Change the value of an item in a list
```

Numeric comparison operators

<code>3 == 3</code> # Test for equality with ==	<code>3 >= 3</code> # Test greater than or equal to with >=
<code>3 != 3</code> # Test for inequality with !=	<code>3 < 4</code> # Test less than with <
<code>3 > 1</code> # Test greater than with >	<code>3 <= 4</code> # Test less than or equal to with <=

Logical operators

<code>~(2 == 2)</code> # Logical NOT with ~	<code>(1 >= 1) (1 < 1)</code> # Logical OR with
<code>(1 != 1) & (1 < 1)</code> # Logical AND with &	<code>(1 != 1) ^ (1 < 1)</code> # Logical XOR with ^

> Getting started with lists

A list is an ordered and changeable sequence of elements. It can hold integers, characters, floats, strings, and even objects.

Creating lists

```
# Create lists with [], elements separated by commas
x = [1, 3, 2]
```

List functions and methods

```
x.sorted(x) # Return a sorted copy of the list e.g., [1,2,3]
x.sort() # Sorts the list in-place (replaces x)
reversed(x) # Reverse the order of elements in x e.g., [2,3,1]
x.reversed() # Reverse the list in-place
x.count(2) # Count the number of element 2 in the list
```

Selecting list elements

Python lists are zero-indexed (the first element has index 0). For ranges, the first element is included but the last is not.

```
# Define the list
x = ['a', 'b', 'c', 'd', 'e']
x[0] # Select the 0th element in the list
x[-1] # Select the last element in the list
x[1:3] # Select 1st (inclusive) to 3rd (exclusive)
x[2:] # Select the 2nd to the end
x[:3] # Select 0th to 3rd (exclusive)
```

Concatenating lists

```
# Define the x and y lists
x = [1, 3, 6]
y = [10, 15, 21]
x + y # Returns [1, 3, 6, 10, 15, 21]
3 * x # Returns [1, 3, 6, 1, 3, 6, 1, 3, 6]
```

> Getting started with dictionaries

A dictionary stores data values in key-value pairs. That is, unlike lists which are indexed by position, dictionaries are indexed by their keys, the names of which must be unique.

Creating dictionaries

```
# Create a dictionary with {}
{'a': 1, 'b': 4, 'c': 9}
```

Dictionary functions and methods

```
x = {'a': 1, 'b': 2, 'c': 3} # Define the x dictionary
x.keys() # Get the keys of a dictionary, returns dict_keys(['a', 'b', 'c'])
x.values() # Get the values of a dictionary, returns dict_values([1, 2, 3])
```

Selecting dictionary elements

```
x['a'] # 1 # Get a value from a dictionary by specifying the key
```

> NumPy arrays

NumPy is a python package for scientific computing. It provides multidimensional array objects and efficient operations on them. To import NumPy, you can run this Python code `import numpy as np`

Creating arrays

```
# Convert a python list to a NumPy array
np.array([1, 2, 3]) # Returns array([1, 2, 3])
# Return a sequence from start (inclusive) to end (exclusive)
np.arange(1,5) # Returns array([1, 2, 3, 4])
# Return a stepped sequence from start (inclusive) to end (exclusive)
np.arange(1,5,2) # Returns array([1, 3])
# Repeat values n times
np.repeat([1, 3, 6], 3) # Returns array([1, 1, 1, 3, 3, 3, 6, 6, 6])
# Repeat values n times
np.tile([1, 3, 6], 3) # Returns array([1, 3, 6, 1, 3, 6, 1, 3, 6])
```

> Math functions and methods

All functions take an array as the input.

```
np.log(x) # Calculate logarithm
np.exp(x) # Calculate exponential
np.max(x) # Get maximum value
np.min(x) # Get minimum value
np.sum(x) # Calculate sum
np.mean(x) # Calculate mean
```

```
np.quantile(x, q) # Calculate q-th quantile
np.round(x, n) # Round to n decimal places
np.var(x) # Calculate variance
np.std(x) # Calculate standard deviation
```

> Getting started with characters and strings

```
# Create a string with double or single quotes
"DataCamp"

# Embed a quote in string with the escape character \
"He said, \"DataCamp\""

# Create multi-line strings with triple quotes
"""
A Frame of Data
Tidy, Mine, Analyze It
Now You Have Meaning
Citation: https://mdsr-book.github.io/haikus.html
"""

str[0] # Get the character at a specific position
str[0:2] # Get a substring from starting to ending index (exclusive)
```

Combining and splitting strings

```
"Data" + "Framed" # Concatenate strings with +, this returns 'DataFramed'
3 * "data " # Repeat strings with *, this returns 'data data data '
"beekeepers".split("e") # Split a string on a delimiter, returns ['b', '', 'k', '', 'p', 'rs']
```

Mutate strings

```
str = "Jack and Jill" # Define str
str.upper() # Convert a string to uppercase, returns 'JACK AND JILL'
str.lower() # Convert a string to lowercase, returns 'jack and jill'
str.title() # Convert a string to title case, returns 'Jack And Jill'
str.replace("J", "P") # Replaces matches of a substring with another, returns 'Pack and Pill'
```

> Getting started with DataFrames

Pandas is a fast and powerful package for data analysis and manipulation in python. To import the package, you can use `import pandas as pd`. A pandas DataFrame is a structure that contains two-dimensional data stored as rows and columns. A pandas series is a structure that contains one-dimensional data.

Creating DataFrames

<pre># Create a dataframe from a dictionary pd.DataFrame({ 'a': [1, 2, 3], 'b': np.array([4, 4, 6]), 'c': ['x', 'x', 'y'] })</pre>	<pre># Create a dataframe from a list of dictionaries pd.DataFrame([{'a': 1, 'b': 4, 'c': 'x'}, {'a': 1, 'b': 4, 'c': 'x'}, {'a': 3, 'b': 6, 'c': 'y'}])</pre>
--	--

Selecting DataFrame Elements

Select a row, column or element from a dataframe. Remember: all positions are counted from zero, not one.

```
# Select the 3rd row
df.iloc[3]
# Select one column by name
df['col']
# Select multiple columns by names
df[['col1', 'col2']]
# Select 2nd column
df.iloc[:, 2]
# Select the element in the 3rd row, 2nd column
df.iloc[3, 2]
```

Manipulating DataFrames

<pre># Concatenate DataFrames vertically pd.concat([df, df]) # Concatenate DataFrames horizontally pd.concat([df,df],axis="columns") # Get rows matching a condition df.query('logical_condition') # Drop columns by name df.drop(columns=['col_name']) # Rename columns df.rename(columns={'oldname': 'newname'}) # Add a new column df.assign(temp_f=9 / 5 * df['temp_c'] + 32)</pre>	<pre># Calculate the mean of each column df.mean() # Get summary statistics by column df.agg(aggregation_function) # Get unique rows df.drop_duplicates() # Sort by values in a column df.sort_values(by='col_name') # Get rows with largest values in a column df.nlargest(n, 'col_name')</pre>
---	--

> Accessing help and getting object types

```
1 + 1 # Everything after the hash symbol is ignored by Python
help(max) # Display the documentation for the max function
type('a') # Get the type of an object – this returns str
```

> Importing packages

Python packages are a collection of useful tools developed by the open-source community. They extend the capabilities of the python language. To install a new package (for example, pandas), you can go to your command prompt and type in `pip install pandas`. Once a package is installed, you can import it as follows.

```
import pandas # Import a package without an alias
import pandas as pd # Import a package with an alias
from pandas import DataFrame # Import an object from a package
```

> The working directory

The working directory is the default file path that python reads or saves files into. An example of the working directory is "C://file/path". The os library is needed to set and get the working directory.

```
import os # Import the operating system package
os.getcwd() # Get the current directory
os.chdir("new/working/directory") # Set the working directory to a new file path
```

> Operators

Arithmetic operators

```
102 + 37 # Add two numbers with +
102 - 37 # Subtract a number with -
4 * 6 # Multiply two numbers with *
22 / 7 # Divide a number by another with /
22 // 7 # Integer divide a number with //
3 ** 4 # Raise to the power with **
22 % 7 # Returns 1 # Get the remainder after
division with %
```

Assignment operators

```
a = 5 # Assign a value to a
x[0] = 1 # Change the value of an item in a list
```

Numeric comparison operators

```
3 == 3 # Test for equality with ==
3 != 3 # Test for inequality with !=
3 > 1 # Test greater than with >
3 >= 3 # Test greater than or equal to with >=
3 < 4 # Test less than with <
3 <= 4 # Test less than or equal to with <=
```

Logical operators

```
~(2 == 2) # Logical NOT with ~
(1 != 1) & (1 < 1) # Logical AND with &
(1 >= 1) | (1 < 1) # Logical OR with |
(1 != 1) ^ (1 < 1) # Logical XOR with ^
```



Math functions and methods

All functions take an array as the input.

```
np.log(x) # Calculate logarithm
np.exp(x) # Calculate exponential
np.max(x) # Get maximum value
np.min(x) # Get minimum value
np.sum(x) # Calculate sum
np.mean(x) # Calculate mean
```

```
np.quantile(x, q) # Calculate q-th quantile
np.round(x, n) # Round to n decimal places
np.var(x) # Calculate variance
np.std(x) # Calculate standard deviation
```

Flow Control Method

- **if-else (Conditional Statement)**

```
if price >= 700:  
    print("Buy.")  
else:  
    print("Don't buy.")
```

- **For loop (Iterative Loop Statement)**

```
a = "New Text"  
count = 0  
for i in a:  
    if i == 'e':  
        count = count + 1  
print(count)
```

- **While loop (Conditional Loop Statement)**

```
a = 0  
i = 1  
while i < 10:  
    a = a * 2  
    i = i + 1  
print(a)
```

- **Loop Control: Break, Pass and continue**

Operators

Numeric Operator: (Say, a holds 5, b holds 10)

- $a + b = 15$
- $a - b = -5$
- $a * b = 50$
- $7.0 // 2.0 = 3.0, -11 // 3 = -4$
- $b / a = 2$
- $b \% a = 0$
- $a ** b = 9765625$

Comparison Operator:

- $(a == b)$: not true
- $(a != b)$: true
- $(a > b)$: not true
- $(a < b)$: not true
- $(a > b)$: not true
- $(a <= b)$ is true

Boolean Operator:

- $a \text{ and } b$
- $a \text{ or } b$
- $\text{not } a$

PYTHON FOR DATA SCIENCE CHEAT SHEET

Python Basics

Datatypes

- Numbers: a=2(Integer), b=2.0(Float), c=1+2j(Complex)
- String: a="New String"
- List: a=[1,2,3,'Word']
- Sets: a= {2,3,4,5}
- Tuple: a=(1,2,4)
- Dictionary: x= {'a': [1,2], 'b': [4,6]}

Operators

Numeric Operator: (Say, a holds 5, b holds 10)

- a + b = 15
- a - b = -5
- a * b = 50
- 7.0//2.0 = 3.0, -11//3 = -4
- b/a = 2
- b % a = 0
- a**b = 9765625

Comparison Operator:

- (a == b): not true
- (a != b): true
- (a > b): not true
- (a > b): not true
- (a < b): true

Boolean Operator:

- a and b
- a or b
- not a

Operations

List Operations

- List=[]: Defines an empty list
- list[i]=a: Stores a at the ith position
- list[i]: Retrieves the character at the ith position
- list[i:j]: Retrieves characters in the range i to j
- list.append(val): Adds item at the end
- list.pop([i]): Removes and returns item at index i

String Operations

- String[i]: Retrieves the character at the ith position
- String[i:j]: Retrieves characters in the range i to j

Dictionary Operations

- dict={} : Defines an empty dictionary
- dict[i]=a: stores "a" to the key "i"
- dict[i]: Retrieves the item with the key "i"
- dict.key: Gives all the key items
- dict.values: Gives all the values

OOPS

Inheritance:

A process of using details from a new class without modifying existing class.

Polymorphism:

A concept of using common operation in different ways for different data input.

Encapsulation:

Hiding the private details of a class from other objects.

Class/object

Class: class Pen:
pass

Object: obj=Pen()

Flow Control Method

- if-else (Conditional Statement)**
if price >= 700:
 print("Buy.")
else:
 print("Don't buy.")
- For loop (Iterative Loop Statement)**
a="New Text"
count=0
for i in a:
 if i=='e':
 count=count+1
print(count)
- While loop (Conditional Loop Statement)**
a=0
i=1
while i < 10:
 a=a*2
 i=i+1
print(a)
- Loop Control: Break, Pass and continue**

Functions

```
def new_function():  
    print("Hello World")  
  
new_function()
```

Lambda Function

```
lambda a,b: a+b  
  
lambda a,b: a*b
```

Comments

```
# Single Line Comment  
#####  
  
Multi-line comment  
#####
```

Generic Operations

- range(5): 0,1,2,3,4
- S=input("Enter:")
- Len(a): Gives item count in a
- min(a): Gives minimum value in a
- max(a): Gives maximum value in a
- sum(a): Adds up items of an iterable and returns sum
- sorted(a): Sorted list copy of a
- importing modules: import random

File Operations

f= open("File Name", "openingmode")

(Opening modes: r: read, w: write, a: append, r+: both read and write)

Try & Except Block

```
try:  
    [Statement body block]  
    raise Exception()  
except Exception as e:  
    [Error processing block]
```



FURTHERMORE:
Python for Data Science Certification Training Course

Python For Data Science Cheat Sheet

Python Basics

Learn More Python for Data Science [Interactively at www.datacamp.com](https://www.datacamp.com)



Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
5
```

Calculations With Variables

>>> x+2 7	Sum of two variables
>>> x-2 3	Subtraction of two variables
>>> x*2 10	Multiplication of two variables
>>> x**2 25	Exponentiation of a variable
>>> x%2 1	Remainder of a variable
>>> x/float(2) 2.5	Division of a variable

Types and Type Conversion

str()	'5', '3.45', 'True'	Variables to strings
int()	5, 3, 1	Variables to integers
float()	5.0, 1.0	Variables to floats
bool()	True, True, True	Variables to booleans

Asking For Help

```
>>> help(str)
```

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset

```
>>> my_list[1]
>>> my_list[-3]
```

Select item at index 1
Select 3rd last item

Slice

```
>>> my_list[1:3]
>>> my_list[1:]
>>> my_list[:3]
>>> my_list[:]
```

Select items at index 1 and 2
Select items after index 0
Select items before index 3
Copy my_list

Subset Lists of Lists

```
>>> my_list2[1][0]
>>> my_list2[1][:2]
```

my_list[list][itemOfList]

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

>>> my_list.index(a)	Get the index of an item
>>> my_list.count(a)	Count an item
>>> my_list.append('!')	Append an item at a time
>>> my_list.remove('!')	Remove an item
>>> del(my_list[0:1])	Remove an item
>>> my_list.reverse()	Reverse the list
>>> my_list.extend('!')	Append an item
>>> my_list.pop(-1)	Remove an item
>>> my_list.insert(0, '!')	Insert an item
>>> my_list.sort()	Sort the list

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

>>> my_string.upper()	String to uppercase
>>> my_string.lower()	String to lowercase
>>> my_string.count('w')	Count String elements
>>> my_string.replace('e', 'i')	Replace String elements
>>> my_string.strip()	Strip whitespaces

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
Selective import
>>> from math import pi
```

pandas Data analysis	Machine learning
NumPy Scientific computing	matplotlib 2D plotting

Install Python

ANACONDA
Leading open data science platform
powered by Python

spyder
Free IDE that is included
with Anaconda

jupyter
Create and share
documents with live code,
visualizations, text, ...

NumPy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3], [4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset

```
>>> my_array[1]
2
```

Select item at index 1

Slice

```
>>> my_array[0:2]
array([1, 2])
```

Select items at index 0 and 1

Subset 2D Numpy arrays

```
>>> my_2darray[:,0]
array([1, 4])
```

my_2darray[rows, columns]

NumPy Array Operations

```
>>> my_array > 3
array([False, False, False,  True], dtype=bool)
>>> my_array * 2
array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
array([6, 8, 10, 12])
```

NumPy Array Functions

>>> my_array.shape	Get the dimensions of the array
>>> np.append(other_array)	Append items to an array
>>> np.insert(my_array, 1, 5)	Insert items in an array
>>> np.delete(my_array, [1])	Delete items in an array
>>> np.mean(my_array)	Mean of the array
>>> np.median(my_array)	Median of the array
>>> my_array.corrcoef()	Correlation coefficient
>>> np.std(my_array)	Standard deviation

DataCamp

Learn Python for Data Science [Interactively](https://www.datacamp.com)



Python For Data Science Cheat Sheet

Jupyter Notebook

Learn More Python for Data Science Interactively at www.DataCamp.com



Saving/Loading Notebooks

Create new notebook

Make a copy of the current notebook

Save current notebook and record checkpoint

Preview of the printed notebook

Close notebook & stop running any scripts

Open an existing notebook

Rename notebook

Revert notebook to a previous checkpoint

Download notebook as

- IPython notebook
- Python
- HTML
- Markdown
- reST
- LaTeX
- PDF

Writing Code And Text

Code and text are encapsulated by 3 basic cell types: markdown cells, code cells, and raw NBConvert cells.

Edit Cells

Cut currently selected cells to clipboard

Paste cells from clipboard to current cursor position

Paste cells from clipboard above current cell

Paste cells from clipboard below current cell

Paste cells from clipboard on top of current cell

Revert "Delete Cells" invocation

Merge current cell with the one above

Move current cell up

Adjust metadata underlying the current notebook

Remove cell attachments

Paste attachments of current cell

Copy cells from clipboard to current cursor position

Paste cells from clipboard below current cell

Delete current cells

Split up a cell from current cursor position

Merge current cell with the one below

Move current cell down

Find and replace in selected cells

Copy attachments of current cell

Insert image in selected cells

Insert Cells

Add new cell above the current one

Add new cell below the current one

Working with Different Programming Languages

Kernels provide computation and communication with front-end interfaces like the notebooks. There are three main kernels:

IP[y]:
IPython

R
IRkernel

IJ[.jl]
Julia

Installing Jupyter Notebook will automatically install the IPython kernel.

Restart kernel

Restart kernel & run all cells

Restart kernel & run all cells

Interrupt kernel

Interrupt kernel & clear all output

Connect back to a remote notebook

Run other installed kernels

Command Mode:

jupyter MyJupyterNotebook Last Checkpoint: a few seconds ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3 13 14

In []:

Edit Mode:

In []:

Executing Cells

Run selected cell(s)

Run current cells down and create a new one above

Run all cells above the current cell

Change the cell type of current cell

toggle, toggle scrolling and clear all output

Run all cells

Run all cells below the current cell

toggle, toggle scrolling and clear current outputs

View Cells

Toggle display of Jupyter logo and filename

Toggle line numbers in cells

Toggle display of toolbar

Toggle display of cell action icons:

- None
- Edit metadata
- Raw cell format
- Slideshow
- Attachments
- Tags

Widgets

Notebook widgets provide the ability to visualize and control changes in your data, often as a control like a slider, textbox, etc.

You can use them to build interactive GUIs for your notebooks or to synchronize stateful and stateless information between Python and JavaScript.

Download serialized state of all widget models in use

Save notebook with interactive widgets

Embed current widgets

1. Save and checkpoint
2. Insert cell below
3. Cut cell
4. Copy cell(s)
5. Paste cell(s) below
6. Move cell up
7. Move cell down
8. Run current cell

9. Interrupt kernel
10. Restart kernel
11. Display characteristics
12. Open command palette
13. Current kernel
14. Kernel status
15. Log out from notebook server

Asking For Help

Walk through a UI tour

Edit the built-in keyboard shortcuts

Description of markdown available in notebook

Python help topics

NumPy help topics

Matplotlib help topics

Pandas help topics

List of built-in keyboard shortcuts

Notebook help topics

Information on unofficial Jupyter Notebook extensions

IPython help topics

SciPy help topics

SymPy help topics

About Jupyter Notebook

