Python Foundations for Data Science

Module 1.2

Sunit Bhattacharya

July 2025

How to write a program

- Let's try writing code for a toy problem.
- Algorithm = Set of instructions
- Program = Implementation of the algorithm.

- Pre-python:
  - How to think of a program using natural language?
  - How to write a program in a make-believe language?

Becoming a Data Scientist: An **Algorithm**

- Step 1: Be born.
- Step 2: Stay alive and pursue education.
- Step 3: Pass important school exams (10th, 12th, etc.).
- Step 4: Decide your career path.
- Step 5: If the chosen path is Data Science, keep learning relevant skills.
- Step 6: Apply for data science jobs when ready.
- Step 7: Work, sleep, and repeat.
- Step 8: Die

This is a high-level algorithm outlining the steps in becoming a data scientist.

Becoming a Data Scientist: A **Program**

```
// Pseudocode implementation of the algorithm
let alive = true
while (alive) {
    if (pass_exams([10,12,15,17]) &&
        decide_career() == "Data Science"
        && check_interest()) {
        let job = apply_for_job()
        while (job && alive) {
            work()
            sleep()
            alive = check_alive()
        }
    } else {
        alive = check_alive()
    }
}
```

How to Think Like a Programmer

- **Algorithm**: A step-by-step way to solve a problem.
- **Program**: Tells the computer how to follow the algorithm.
- **Programming Language**: A structured way to write instructions for a machine.

Example: Calculate Average Inflation Over 5 Years

- Step 1: Collect inflation rates
- Step 2: Add all values
- Step 3: Divide by count

Python Program: Average Inflation

```
inflation = [4.2, 5.1, 6.3, 3.9, 4.7]
average = sum(inflation) / len(inflation)
print("Average inflation:", average)
```

How to Run Python Code

- **Options:**
  - Free online compilers
  - Google Colab
  - VS Code
  - Python shell (for quick tests)
- **Recommended:** Use Colab for this course (free and easy)

What is a Virtual Environment in Python?

Virtual Environment:

- A *virtual environment* is an isolated Python environment that allows you to manage separate package installations for different projects without conflicts.
- It ensures that each project can have its own dependencies (libraries and versions) independent of others.
- Avoids version clashes and preserves system Python integrity.

Why use Virtual Environments?

- Different projects may require different versions of the same package.
- Simplifies dependency management and reproducibility.
- Facilitates sharing and deployment of projects.

What is a Virtual Environment in Python?

Basic Commands with `venv` (built-in tool):

```
# Create virtual environment
python3 -m venv env_name

# Activate (Linux/macOS)
source env_name/bin/activate

# Activate (Windows)
.\env_name\Scripts\activate

# Deactivate
deactivate
```

How to Install and Use Virtual Environments

… **With Anaconda:**

```
# Create a new environment
conda create -n env_name python=3.11

# Activate the environment
conda activate env_name

# Deactivate
conda deactivate
```

Python Basics: Variables and Types

```python
x = 10
country = "India"
gdp_values = [2.9, 3.1, 3.4]  # in trillion USD
```

- Types: integer, string, list
- Python uses dynamic typing

Classroom Exercise: Algorithm Design

- Task: Design an algorithm to **calculate the average temperature** over a week.
- Write down the **step-by-step instructions** (in plain English or pseudocode) for this task.
- Think about:
    - How to represent the temperatures (list, variables, etc.)
    - How to sum the values
    - How to calculate the average
- **Example:**
    - Step 1: Collect temperatures for 7 days.
    - Step 2: Add all the collected temperatures.
    - Step 3: Divide the total by 7.
    - Step 4: Output the result.

Classroom Exercise: Implementing the Program

```
# Python program to calculate average temperature over a week

temperatures = [23.4, 22.1, 21.8, 24.0, 23.9, 22.5, 21.7]
total = sum(temperatures)
count = len(temperatures)
average_temp = total / count
print("Average temperature over the week:", average_temp)
```

Try writing this code yourself based on your algorithm!

Control Flow: Conditions and Loops

Why use Conditionals?

- Conditionals allow programs to **make decisions** based on data.
- They enable different **actions** depending on whether certain conditions are true or false.
- This helps in creating **flexible and dynamic** programs that can respond to varying inputs.

Example: Identifying High GDP Values

```
for gdp in gdp_values:
    if gdp > 3.0:
        print("High GDP")
```

The 'if' statement checks the GDP value and only prints a message when the condition '$gdp > 3.0$' is `True`.

Classroom Exercise: Conditional Statements

Task: Categorize temperature levels based on value

Write a Python program to classify the day's temperature (in °C) as:

- **Cold**: $temperature < 10$
- **Mild**: $10 \leq temperature \leq 25$
- **Hot**: $temperature > 25$

Classroom Exercise: Conditional Statements

Task: Categorize temperature levels based on value Example

Algorithm:

- Input temperature
- If temperature < 10, print "Cold"
- Else if temperature between 10 and 25 inclusive, print "Mild"
- Else, print "Hot"

Sample Python Code:

```
temperature = float(input("Enter temperature (°C): "))

if temperature < 10:
    print("Cold")
elif 10 <= temperature <= 25:
    print("Mild")
else:
    print("Hot")
```

Functions in Python: Concept and Example

What is a Function?

- A **function** is a reusable block of code that performs a specific task.
- Functions help to **organize** code, **avoid repetition**, and make programs easier to understand and maintain.
- Functions can take **inputs** (parameters) and return an **output** (result).

Example: Calculate GDP Growth Rate

```
def growth_rate(gdp_now, gdp_before):
    return (gdp_now - gdp_before) / gdp_before

print(growth_rate(3.1, 2.9))  # 6.9% growth
```

The function takes two GDP values and returns the relative growth between them.

Classroom Exercise: Writing Your Own Function

Task: Write a Python function `average(lst)` **to calculate the average of a list of numbers.**

Steps to consider:

- The function should accept a `list` of numeric values as input.
- Calculate the sum of all items in the list.
- Divide by the number of items to get the average.
- Return the average value.

Example usage:

```
print(average([10, 20, 30]))  # Output should be 20.0
```

Try writing the function yourself and test it with different lists!

Why Visualize Data?

- Visualization helps us **see patterns**, trends, and anomalies in our data.
- It makes complex datasets **intuitive and accessible**.
- Common use cases: understanding distributions, spotting outliers, and comparing groups.

Matplotlib is Python's most popular library for creating static, interactive, and animated graphics.

Simple Visualization with Matplotlib

Let's plot the distribution of `random` data:

```
import matplotlib.pyplot as plt
import numpy as np
import random # You need to import the 'random' module
# Generate 1,000 random data points
data_uniform = [random.random() for _ in range(1000)]
plt.hist(data_uniform, bins=20) # You were using 'data' instead of
plt.title("Histogram of Random Data")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

This code generates a histogram of random numbers.

Why Do We Need Specialized Numeric Libraries?

Example Problem: Sum of Two Large Arrays

- Task: Add two arrays, each with 10 million numbers, element-wise.
- With a **Python list** and a `for` loop, this operation can take several **seconds or more** due to Python's slow loop execution.
- With **NumPy arrays**, the same operation is performed in a tiny fraction of a second!

Why?

- NumPy uses efficient, compiled code underneath.
- Operations are **vectorized**—no Python loops needed!
- This efficiency is essential for data science, AI, statistics, and scientific computing.

Numeric Computing: Why Do We Need Specialized Libraries?

- Numeric computing involves **processing large numerical datasets** efficiently.
- Python lists are flexible but **not optimized for numerical operations**:
  - Operations on lists are often **slow** and **verbose**.
  - Lack of **vectorized operations** means looping explicitly for calculations.
  - Memory inefficiency: lists are collections of pointers.
- **NumPy** provides:
  - **Multidimensional arrays** (ndarrays) optimized for fast numerical computations.
  - **Vectorized operations** allowing element-wise math without explicit Python loops.
  - Efficient memory use and integration with C/C++ backends for speed.
- Essential for data science, machine learning, scientific computing, and more!

Introduction to NumPy Arrays

- NumPy's primary data structure: **ndarray** (N-dimensional array)
- Unlike Python lists, NumPy arrays:
  - Hold elements of the **same data type** (homogeneous)
  - Support efficient, **element-wise operations**
  - Can be **multi-dimensional** (1D, 2D, 3D, ...)
- Example: Creating a 1D NumPy array of GDP values

```
import numpy as np
gdp = np.array([2.9, 3.1, 3.4])
print(gdp)
```

Computing with NumPy: Basic Array Operations

- NumPy makes numerical computing fast and simple:
  - Element-wise arithmetic (+, −, ∗, /)
  - Aggregations (`np.mean()`, `np.std()`, etc.)
- Operations are vectorized—no manual loops necessary!

Example: Analyze synthetic data

```
mean = np.mean(data)
std = np.std(data)
print(f"Mean: {mean}, Std: {std}")
```

NumPy vs. Pure Python: Simplicity

- **Adding two lists element-wise (Pure Python):**

```
a = [1, 2, 3, 4, 5]
b = [6, 7, 8, 9, 10]
c = [a[i] + b[i] for i in range(len(a))]
print(c)  # [7, 9, 11, 13, 15]
```

- Requires manual looping or a list comprehension.
- Can get cumbersome and error-prone for complex operations.

With NumPy: Simple, Intuitive Syntax

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
b = np.array([6, 7, 8, 9, 10])
c = a + b
print(c)  # [ 7  9 11 13 15]
```

- One line does the entire calculation!

NumPy vs. Pure Python: Speed for Large Problems

- **Pure Python:** Element-wise operations on large lists are slow—Python loops have overhead.
- **NumPy:** Uses optimized, compiled C code under the hood; vectorized operations run orders of magnitude faster.
- **Example:** Adding two arrays of one million elements:

Pure Python

```
# Slow, loop-based approach
result = [a[i] + b[i] for i in range(1_000_000)]
```

NumPy

```
# Fast, vectorized approach
result = a + b
```

- **Result:** NumPy performs the operation much faster and with less code!

Generating Synthetic Data with NumPy

- **NumPy** enables the creation of large arrays of random numbers for simulations, testing, and experiments.
- Synthetic data is crucial for experimenting with algorithms without real-world datasets.
- NumPy offers tools like `np.random` for generating arrays from various distributions (uniform, normal, etc.).

Example: Generate 1000 random values

```
import numpy as np

data = np.random.rand(1000) # Uniform [0, 1)
print(data[:10])
```

Create Structured Data: Simulate Multivariate Records

- You can simulate structured datasets: e.g., "fake persons" with age, height, and weight.
- Use different NumPy random functions for each field.

Example: Synthetic demographic data

```
n = 1000
ages = np.random.randint(18, 65, n)
heights = np.random.normal(170, 10, n)
weights = np.random.normal(65, 15, n)
```

- This creates 1,000 fake people with plausible age, height, and weight.

Summary: Why Use Synthetic Data?

- Rapidly prototype algorithms and analysis.
- Test statistical methods on controlled data.
- Practice machine learning workflows—even when real datasets aren't available.
- Sharing code is easier since everyone can generate the same fake data.

NumPy's random module is your friend for quick, flexible data generation!

Introducing Pandas: Python's Data Analysis Library

- **Pandas** is a powerful, open-source Python library dedicated to data analysis and manipulation.
- Designed for working with "tabular" (spreadsheet-like) and time-series data.
- Developed by Wes McKinney, Pandas offers functions to load, clean, transform, and analyze data efficiently.
- It is integral to the data science ecosystem and widely used for everything from exploratory analysis to data engineering.
- Pandas is built on top of **NumPy** and extends its capabilities to more complex data structures and operations.

Pandas and NumPy: Working Together

- **NumPy** provides fast, efficient operations on numerical arrays.
- **Pandas** leverages NumPy "under the hood" for its performance.
- Every Pandas `Series` or `DataFrame` stores its data as one or more underlying NumPy arrays.
- You get both: the speed of NumPy and the ease of use and labeling capabilities of Pandas (row and column labels).
- Many Pandas operations automatically broadcast or vectorize using NumPy's machinery.
- Useful workflow: Load and manipulate data with Pandas, perform heavy numerical computations with NumPy if needed.

DataFrame: The Core Data Structure of Pandas

- The **DataFrame** is a 2-dimensional labeled data structure, similar to a spreadsheet or SQL table.
- Columns can contain different data types (numbers, text, dates, etc.).
- DataFrames support powerful filtering, aggregation, grouping, merging, and reshaping.
- You can create a DataFrame from lists, NumPy arrays, dictionaries, or direct imports (CSV, Excel, SQL).
- Think of a DataFrame as a “supercharged table” for all your data analysis needs!

Simple DataFrame Example

```
import pandas as pd
data = {'Country': ['India', 'UK'],
        'Population': [1400, 68]}
df = pd.DataFrame(data)
print(df)
```