Advanced Python

for Neuroscience

Saeed Mohagheghi

2025 | ۱۴۰۴

ویژگی‌های دوره

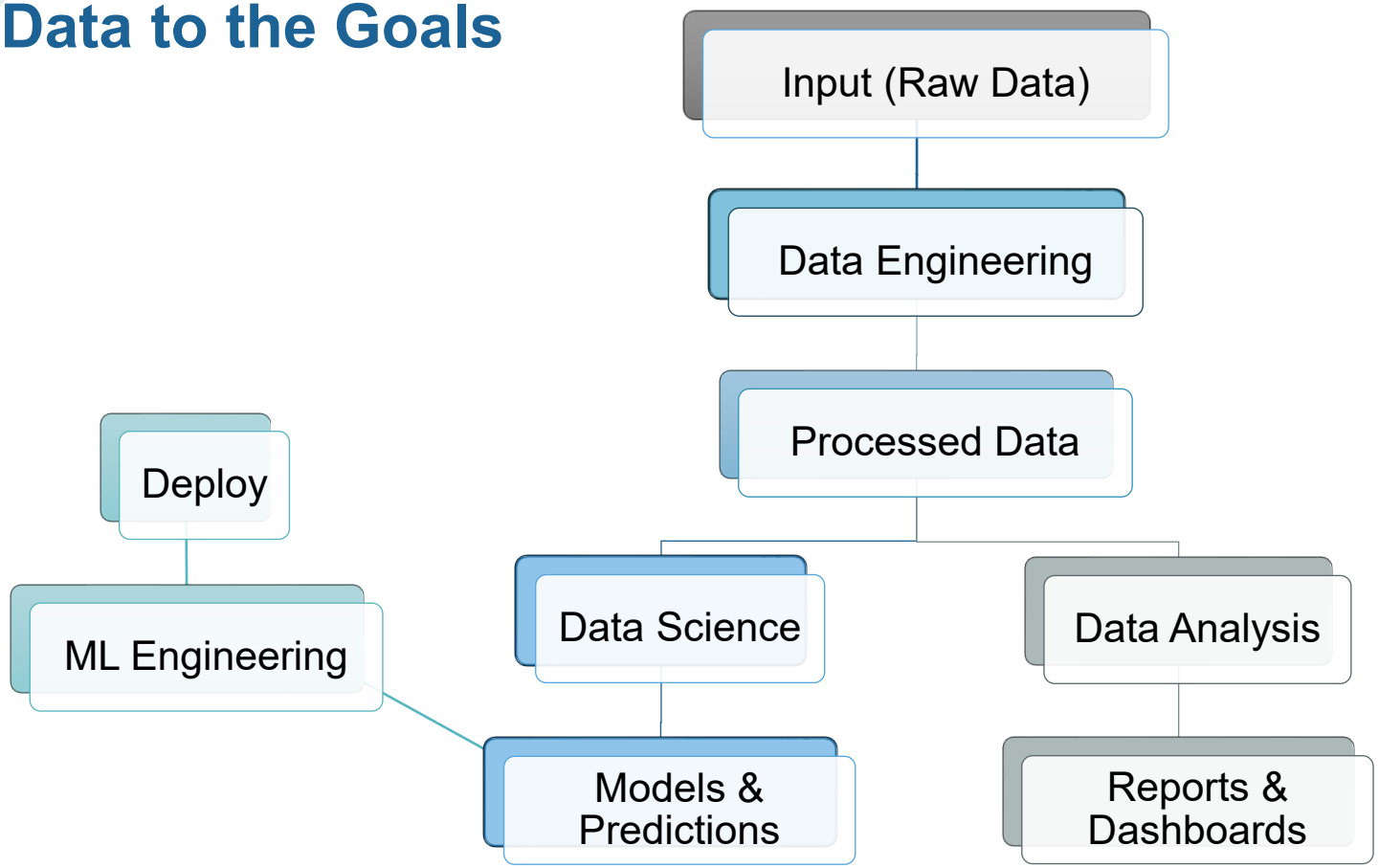
- پروژه محور
- کار با داده‌های واقعی
- مفاهیم پیشرفته پایتون
- کدنویسی زنده (Live Coding)
- کدنویسی با هوش مصنوعی (Vibe Coding)
- ...

پیش نیازها

- پایتون مقدماتی
- متغیرها / دستورات شرطی / حلقه‌ها / توابع / کلاس‌ها
- کتابخانه‌های numpy / pandas / matplotlib
- آشنایی با محیط Notebook و Google Colab
- زبان انگلیسی (در حد جستجو و استفاده از هوش مصنوعی)
- توانایی حل مساله
- ...

The Workflow

From the Data to the Goals



Data Science Pipeline

Scientist

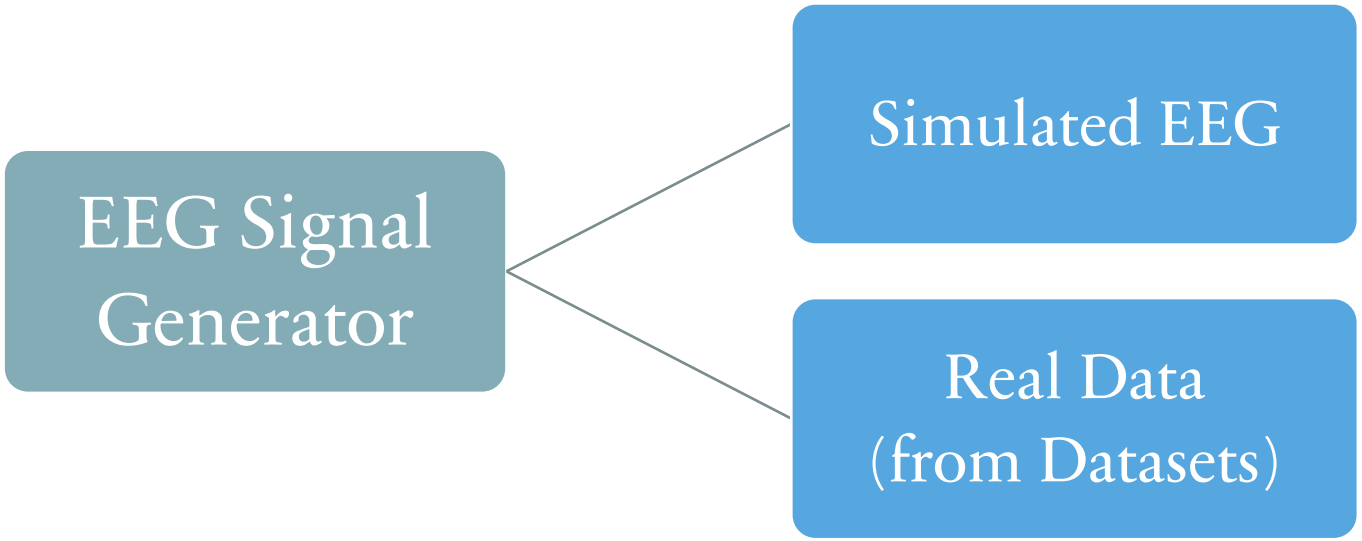
Data
- Collection
- Cleaning
- Pre-processing

Model
- Design
- Training
- Evaluation

Deploy
- Serve
- Monitor
- Maintenance

User

پروژه اول (Data)



سبک‌های کدنویسی

- **Procedural** → Top-down sequence of instructions and reusable procedures
- **Functional** → Functions as primary citizens, immutable data, and data transformation
  In Python: first-class functions, map, filter, and lambda expressions
- **Object-oriented (OOP)** → Encapsulating data and the functions (methods) into self-contained objects

programming_paradigms.ipynb

*args / **kwargs

- قابلیت تعریف تعداد نامعلوم پارامتر ورودی برای توابع

- ***args**:
  - Allows a function to accept a variable number of **positional arguments**.
  - The arguments are passed as a tuple.

- ****kwargs**:
  - Allows a function to accept a variable number of **keyword arguments**.
  - The arguments are passed as a dictionary.

مثال‌های *args و **kwargs

```python
def sum_numbers(*args):
    return sum(args)

result = sum_numbers(1, 2, 3, 4)
# result is 10
```

```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)
```

Output:
```
name: Alice
age: 30
```

```python
def mixed_function(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

mixed_function(1, 2, name="Bob", age=25)
```

Output:
```
Positional arguments: (1, 2)
Keyword arguments: {'name': 'Bob', 'age': 25}
```

https://book.pythontips.com/en/latest/args_and_kwargs.html

Generators in Python

تولید خروجی در حین اجرا (بدون نیاز به ذخیره در حافظه)

- Values are generated on-the-fly and only when requested
- Defined using a function with the **yield** statement
- Retains its state between calls,

  allowing it to resume where it left off

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

# Usage
for number in countdown(5):
    print(number)
# Output:
# 5
# 4
# 3
# 2
# 1
```

•Use `next(generator)` to manually retrieve the next value from a generator.

Generators in Python

- Advantages of Generators
  - **Memory Efficiency**: Ideal for working with large datasets or streams of data.
  - **Improved Performance**: Avoids the overhead of creating and storing an entire list.
  - **Pipelining**: Can be used to create pipelines for processing data in stages.

- Use Cases
  - Reading large files line by line.
  - Generating infinite sequences (e.g., Fibonacci numbers).
  - Processing data streams (e.g., web scraping).

generators.ipynb

https://book.pythontips.com/en/latest/generators.html

Function Decorators

- تغییر رفتار یا افزودن عملیات به توابع، بدون تغییر تابع

- A **decorator** is a special type of function that **modifies the behavior of another function**.
- Addition of functionality to existing code in a clean and readable way.

```
def decorator(func):
    def wrapper():
        print("Before calling the function.")
        func()
        print("After calling the function.")
    return wrapper
```

```
@decorator
def target_function():
    print("Hello World")
```

```
target_function():
```

```
Output:
Before calling the function.
Hello, World!
After calling the function.
```


decorators.ipynb

https://book.pythontips.com/en/latest/decorators.html

Asynchronous Programming in Python

- A method of **parallel programming** that allows tasks to run concurrently without blocking the main thread
- **asyncio**: A library to write concurrent code using the async and await syntax.

- **Advantages of Asynchronous Programming**
  - **Improved Responsiveness**: Ideal for applications with I/O-bound operations (e.g., network requests, file I/O).
  - **Concurrency**: Easily manage multiple tasks without blocking.
  - **Better Resource Utilization**: Allows for multiple tasks to run in overlapping time periods, maximizing CPU usage.

مهم‌ترین توابع asyncio

- **asyncio.Queue()**
  - A FIFO queue designed for use with coroutines.
  - Useful for task synchronization and communication between producers and consumers.

- **asyncio.create_task()**
  - Schedules the execution of a coroutine and returns a Task object.
  - Allows multiple coroutines to run concurrently.

- **asyncio.gather()**
  - Runs multiple coroutines concurrently and waits for all of them to complete.
  - Returns results in the order of the input coroutines.

- **asyncio.run()**
  - A high-level function to run the main entry point of an asynchronous program.
  - It handles the event loop and ensures proper cleanup.

async_await.ipynb

Multithreading

- **What is Multithreading?**
  - Normally, Python runs **one thing at a time** in the main thread.
  - If a task takes time (e.g., time.sleep() or waiting for I/O), the program is *blocked*.
  - **Threads** let us run multiple tasks *concurrently*.
  - Useful for I/O-bound tasks (networking, file operations).
- **Why not for CPU-bound tasks?**
  - Python has the **GIL (Global Interpreter Lock)** → only one thread executes Python bytecode at a time.
  - Use **multiprocessing** instead for CPU-bound work.

multithread.ipynb

Multithreading Example

```
import threading, time

def worker(name):
    print(f"Thread {name} starting")
    time.sleep(2)
    print(f"Thread {name} done")

threads = []
for i in range(3):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
print("All threads finished")
```

Multiprocessing

- **What is multiprocessing?**
  - Runs **separate processes**, each with its own Python interpreter & memory.
  - Bypasses the **GIL**, so true **parallel execution** for CPU-bound tasks.
- **When to use:**
  - Heavy computations (data processing, image/video processing, simulations).
  - Tasks that benefit from multiple CPU cores.

Multiprocessing Example

```
import os
from multiprocessing import Process

def worker(name):
    print(f"Process {name} (PID {os.getpid()}) running")

processes = []
for i in range(3):
    p = Process(target=worker, args=(i,))
    processes.append(p)
    p.start()

for p in processes:
    p.join()

print("All processes finished")
```

Async / Threading / Multiprocessing

Feature	Async (asyncio)	Threading	Multiprocessing
Execution type	Single thread, event loop	Threads in same process	Separate processes
GIL impact	None (single thread, cooperative multitasking)	Limited (still one thread executes Python at a time)	None (true parallelism)
Best for	I/O-bound tasks (many concurrent)	I/O-bound tasks	CPU-bound tasks
Memory overhead	Very low	Low	High
Overhead to start	Very low	Low	High
Example use cases	Thousands of network requests, async APIs	API calls, file I/O	Heavy computations, simulations
Syntax complexity	Slightly more complex (async/await)	Simple	Simple

Python Packaging

- Register in TestPypi: https://test.pypi.org/
  - Registration:
    1. Enter Name, Email, Username, Password
    2. Confirm Email (with link in the sent email)
    3. Generate Recovery codes
    4. Enable 2FA
- Use UV to initialize / add dependencies / build
- Use Twine or UV to publish

https://packaging.python.org/en/latest/

Thank You

Saeed Mohagheghi

 Daneshjoy.ir@gmail.com

 https://github.com/DaneshJoy/

 https://www.linkedin.com/in/saeed-mohagheghi