🧠 Python for AI Engineering

Generative AI Bootcamp – Week 1, Day 1, Session 1

November 17, 2025

Learning Objectives

- Refresh core Python concepts for AI systems
- Understand clean, maintainable code structure
- Explore type hints, decorators, and idiomatic patterns
- Prepare foundation for async and service-oriented labs

Python in the AI Stack

- Language of choice for data science and AI engineering
- Integrates seamlessly with model APIs and orchestration tools
- Strong ecosystem: `numpy`, `pandas`, `torch`, `fastapi`, `pydantic`, etc.
- Enables prototyping → production with minimal rewrites

Clean Python Code Principles

1. **Explicit is better than implicit** (Zen of Python)
2. Use **type hints** and **docstrings**
3. Follow **PEP8** conventions
4. Write **modular** code (no logic duplication)
5. Avoid side effects (particularily, global variables)
6. Prefer **composition** over inheritance for modular AI pipelines

(Parenthesis I: from the Zen of Python)

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Readability counts.
> There should be one – and preferably only one – obvious way to do it.

The Importance of Type Hints

- Help catch bugs early by enabling static analysis tools like MyPy
- Make code self-documenting, providing immediate clarity on function parameters and return values without requiring detailed docstrings
- Enhance developer productivity through better IDE support, including autocomplete, error highlighting, code navigation, and safer refactoring
- Support advanced use cases such as runtime validation (e.g., Pydantic, typeguard), dependency injection (e.g., Lagom), and automatic API documentation generation (e.g., drf-spectacular for OpenAPI specs)

(Parenthesis II: from PEP8)

```
# Correct:
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Wrong:
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

(Parenthesis II: from PEP8)

```python
# Correct:
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

foo = long_function_name(
    var_one, var_two,
    var_three, var_four)

# Wrong:
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

(Parenthesis II: from PEP8)

```
# Correct:
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)

# Wrong:
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

(Parenthesis II: from PEP8)

```
# Correct:
spam(ham[1], {eggs: 2})
spam(1)
dct['key'] = lst[index]
foo = (0,)
if x == 4: print(x, y); x, y = y, x

# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
spam (1)
dct ['key'] = lst [index]
bar = (0, )
if x == 4 : print(x , y) ; x , y = y , x
```

(Parenthesis II: from PEP8)

🧠✏️ Homework: take a close look at PEP8 🙏

Helpful Tools

- Coding style: formatters like `black`
- Static analysis I: linters like `ruff` (unused variables, incorrect syntax, excessive function complexity, or misuse of APIs)
- Static analysis II: type checkers like `mypy`

Key Syntax & Features

- Comprehensions, enumerate, zip, map/filter
- `typing` for clarity and maintainability
- Context managers (`with` statements)
- `dataclasses` for structured configs
- functions are first-class objects (decorators, DI)
- asynchronous programming

Example: Data Preprocessing Utility

```python
from dataclasses import dataclass

@dataclass
class TextCleaner:
    stopwords: list[str]

    def clean(self, text: str) -> str:
        return " ".join(
            w.lower() for w in text.split() if w.lower() not in self.stopwords
        )

cleaner = TextCleaner(stopwords=['the', 'is', 'a'])
cleaner.clean('This is a simple Example')
```

Decorators in Practice

- Attach reusable logic to functions (e.g., timing, logging, validation)

```python
import time

def timed(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"Execution: {time.time() - start:.2f}s")
        return result
    return wrapper

@timed
def slow_function():
    ...
```

Decorators in Practice

You can also have "decorator factories" 🤯

```python
def retry(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            k = 0
            while k < n + 1:
                k += 1
                if k > 1: print('retrying...')
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(e)
            raise Exception('reached max retries')
        return wrapper
    return decorator
```

Takeaways

- Write **clear, testable** code – avoid hidden side effects
- Leverage **typing**, **dataclasses**, and **decorators**
- Prepare for **modular AI service code** in next sessions