LECTURER: TAI LE QUY

OBJECT ORIENTED AND FUNCTIONAL PROGRAMMING WITH PYTHON

Thanks Prof. Dr. Max Pumperla for his contribution

TOPIC OUTLINE

Object oriented programming	1 + 2
Functional programming	3
Projects and testing in Python	4
Working with Database in Python	5
Documenting a project	6

UNIT 4

PROJECTS AND TESTING IN PYTHON

STUDY GOALS

- Structuring a project
- Unit testing in Python using pytest

- Directory structure:
  - Root directory (usually the project name)
    - *main.py* entry point or other scripts
  - */src* or */your_project_name* for main code.
  - */tests* for testing related files.
  - */docs* for documentation.
  - *setup.py* or *pyproject.toml* for packaging and distribution.
- Importance of an __init__.py
  - Makes a directory a package.
  - Can contain initialization code for a package.
- Use of virtual environments
  - Isolate project dependencies.
  - Tools: *virtualenv, venv, conda* etc.

DESIGN CONSIDERATIONS

- Modularity
  - Files and packages for clarity
- Code Readability counts
- Decoupling Components
  - Keep different functionalities separate.
  - Make use of classes, functions, and modules.
- Error Handling
  - Use of *try, except, finally*.
  - Anticipate and handle potential issues gracefully.

FOCUS ON TESTING FROM THE START

- Importance of Testing
  - Easy refactoring
  - Catching bugs early
  - Ensuring code quality and reliability
- Types of Tests
  - Unit tests: Test individual components
  - Integration tests: Test interactions between components.
- Using *pytest*
  - pytest ./tests
  - Files should have test_ prefix
  - Test code (usually functions) also have a test_ prefix
  - Lots of useful extensions (e.g. code coverage)

- A simplistic habit class
  - Not a full solution
  - Four attributes
  - Save all “check-off” dates explicitly
  - Simple “check” function
  - Save this as “habit.py”
  - Test file will be called “test_habit.py”

```python
from datetime import datetime, timedelta

class Habit:

    def __init__(self, name, desc):
        self.name = name
        self.desc = desc
        self.streak = 0
        self.check_dates = []

    def check(self, date):
        self.check_dates.append(date)
        self.compute_streak()
```

STREAK COMPUTATION

```python
def compute_streak(self):
    # Sorting check_dates
    sorted_dates = sorted(self.check_dates)

    # If no dates are checked, streak is zero
    if not sorted_dates:
        self.streak = 0
        return

    longest_streak = 1
    current_streak = 1
    for i in range(1, len(sorted_dates)):
        if sorted_dates[i] - sorted_dates[i - 1] == timedelta(days=1):
            current_streak += 1
            longest_streak = max(longest_streak, current_streak)
        else:
            current_streak = 1

    self.streak = longest_streak
```

PYTEST TEST CLASS

- TestHabit class
- setup_method
  - Set up data etc.
- Test methods

```python
import pytest

class TestHabit:

    def setup_method(self):
        # Setting up a Habit instance for testing
        self.habit = Habit("Exercise", "Do exercises for 30 minutes")

    def test_single_date(self):
        self.habit.check(datetime(2023, 8, 19))
        assert self.habit.streak == 1

    def test_multiple_dates(self):
        # Checking consecutive dates
        self.habit.check(datetime(2023, 8, 18))
        self.habit.check(datetime(2023, 8, 19))
        self.habit.check(datetime(2023, 8, 20))
        assert self.habit.streak == 3

        # Checking non-consecutive dates
        self.habit.check(datetime(2023, 8, 22))
        assert self.habit.streak == 3  # Streak should still be 3
```

- teardown_method
  - clean up
- Run with “pytest .”

```python
    def test_random_order_dates(self):
        # Checking dates in random order
        self.habit.check(datetime(2023, 8, 19))
        self.habit.check(datetime(2023, 8, 17))
        self.habit.check(datetime(2023, 8, 18))
        assert self.habit.streak == 3

    def teardown_method(self):
        # Clean up the habit instance
        del self.habit

# Run the tests
pytest.main()
```