VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

ಜಞ

ADVANCE PROGRAMING

ASSIGNMENT REPORT

FUNCTIONAL PROGRAMING IN PYTHON

CC02 — SEMESTER 232

LECTURER: TRƯƠNG TUẤN ANH

PHAN DUY HÃN

Student	ID
Nguyễn Viết Hùng	2252272

HO CHI MINH CITY – 2024

I/The programming language Python

Introduction to Python

Python is a high-level, interpreted programming language renowned for its clarity and readability, making it highly appealing to both novice and experienced developers. Created by Guido van Rossum and first released in 1991, it was designed to be an accessible yet powerful language, supporting software development across diverse applications.

Historical Background

Python's development, initiated in the late 1980s, was aimed at succeeding the ABC language, emphasizing simplicity and exception handling.

Design Philosophy

The language is built around the philosophy of readability and simplicity, reflected in its motto "Beautiful is better than ugly, and readability counts." This philosophy reduces maintenance costs and supports rapid development and large system management alike.

Core Features

Python supports multiple paradigms including procedural, object-oriented, and functional programming, and features like dynamic typing and automatic memory management.

Advantages and Disadvantages

Python's comprehensive standard library and a rich ecosystem of third-party packages allow developers to perform a wide array of tasks, from web development with frameworks like Django and Flask to complex data analysis and machine learning with libraries like NumPy and TensorFlow. Despite these strengths, its simplicity can lead to drawbacks such as slower execution speeds compared to compiled languages like C++ and Java, and potential runtime errors due to its dynamic nature.

Development and Evolution

Python's evolution is driven by Python Enhancement Proposals (PEPs), which guide the introduction of new features and involve the community in decision-making processes. Although Python 3 introduced significant changes that were not backward-compatible with Python 2, it marked a significant evolution in the language's capabilities, including better support for Unicode and improved garbage collection.

Ecosystem and Community

The ecosystem around Python extends to vast libraries and frameworks, enhancing its applications in web development, data science, and beyond. Python's ease of use and versatility also make it popular in academic and educational settings, introducing programming concepts to students and non-programmers alike. The strong global

community around Python, with numerous conferences and meetups, underscores its role as a continually evolving language that adapts to new trends and maintains its relevance in the software development world.

How to create a functional program in Python

Creating a functional program in Python involves leveraging functional programming principles such as immutability, pure functions, higher-order functions, and using functions as first-class citizens. Here's how you can apply these principles to create a functional program in Python:

1. Use Pure Functions

A pure function is one that, given the same inputs, always returns the same output and does not have any observable side effects. This means it doesn't alter any external state (like global variables) or depend on any hidden state.

This function is pure because it always produces the same output for the same `a` and `b`, and it does not modify or depend on external state.

2. Employ Immutability

Immutability means that once an object is created, it cannot be changed. In Python, you can achieve this by using tuples instead of lists for data that should not be modified, or by only modifying data structures by creating copies of them rather than changing the original objects.

This function does not modify the original list but returns a new list with the item added.

3. Utilize Higher-Order Functions

Higher-order functions are functions that take other functions as arguments or return them as results. Python's built-in functions like `map()`, `filter()`, and `reduce()` are common

examples.

4. Function Composition

Function composition involves creating new functions by combining existing functions, which can help in building complex operations from simpler ones.

```python
def double(x):
    return x * 2

def increment(x):
    return x + 1

def double_then_increment(x):
    return increment(double(x))
```

5. Leverage List Comprehensions and Generators

List comprehensions and generators provide a functional approach to creating lists and iterators, which can be used instead of loops and mutable accumulators.

```python
squares = [x*x for x in range(10)]
```

6. Avoid Side Effects
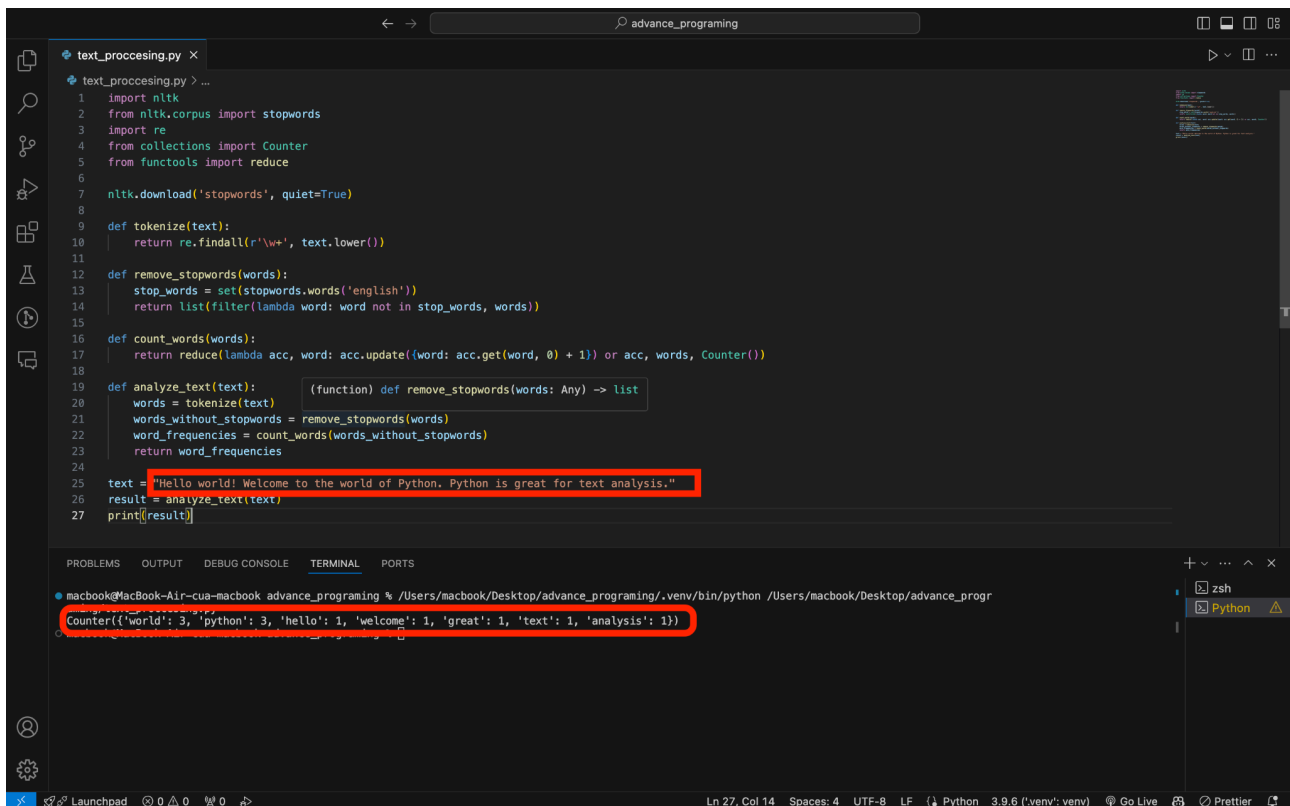
Avoid functions that change external states or depend on them. This approach helps in keeping your functions pure and your program predictable.

By combining these strategies, you can develop programs in Python that adhere to functional programming principles, leading to cleaner, more maintainable code that's easier to debug and test.

III/DEMO

[Link GitHub](#)

General description:

This Python program is designed for basic text processing and analysis, specifically focusing on word frequency analysis in given text data. It utilizes the principles of functional programming to ensure that each function is pure and side-effect free, promoting readability and maintainability. The program is comprised of several distinct functions, each responsible for a specific aspect of text processing:

Explain each function

1. `tokenize(text)`: This function uses a regular expression to split the provided text into words, converting all characters to lowercase to standardize the results. It returns a list of words found in the text, which are the basic units for further analysis.

2. `remove_stopwords(words)`: After tokenizing the text, this function filters out common English stopwords (frequently occurring words like "and", "the", etc., which often do not contribute meaningfully to text analysis). It uses a set of stopwords

from the NLTK library and filters the list of words, returning only those not present in the stopwords list.

3. `count_words(words)`: This function counts the frequency of each word in the list returned by `remove_stopwords`. It employs the `reduce` function to accumulate results in a `Counter` dictionary, where each key is a word and each value is the count of that word in the text. The use of `reduce` here illustrates how accumulative operations can be performed functionally.

4. `analyze_text(text)`: Acting as the main function of the script, this combines all the above functions into a pipeline. It first tokenizes the text, then removes stopwords, and finally counts the frequency of the remaining words. The result is a dictionary of word frequencies which provides insights into the most significant words in the text.