VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED PROGRAMMING (CO2039)

Assignment (Semester: 232)

Functional Programming in Python

Advisor: Trương Tuấn Anh, CSE-HCMUT
Students: Trần Đình Đăng Khoa - 2211649.

HO CHI MINH CITY, MAY 2024

Contents

1 Introduction

Functional programming (FP) is a programming paradigm that has garnered significant attention in both academic and professional realms due to its potent abstraction capabilities and its emphasis on immutability and first-class functions. Unlike imperative programming, which revolves around changing state and iterating over instructions, functional programming conceptualizes computation as the evaluation of mathematical functions, eschewing changing state and mutable data.

Python, although traditionally recognized as an imperative and object-oriented language, has embraced several functional programming features, rendering it suitable for this paradigm. Python supports first-class functions, enabling functions to be treated as any other object, passed as arguments, and returned from other functions. Higher-order functions, which accept other functions as parameters or return them as results, are also integral to Python's functionality. Furthermore, Python's lambda expressions facilitate the creation of anonymous functions, allowing for succinct and expressive code.

The purpose of this report is to explore the application of functional programming principles in Python. We will delve into key functional programming concepts supported by Python, including first-class functions, higher-order functions, and lambda expressions. Additionally, we will demonstrate how to create a functional program in Python through a practical example that processes a list of numbers using the map, filter, and reduce functions.

2 Python as a Functional Programming Language

Python, while primarily an imperative and object-oriented language, provides several features that facilitate functional programming. These features include first-class functions, higher-order functions, anonymous functions (lambdas), and a rich standard library that supports functional constructs.

2.1 First-Class Functions

In Python, functions are first-class citizens. This designation means that functions can be assigned to variables, passed as arguments to other functions, and returned as values from other functions.

```
    def add(x, y):
        return x + y

    f = add
    print(f(2, 3))  # Output: 5
```

Listing 1: First-Class Functions

First-class functions are fundamental to functional programming as they allow for higher-order functions and functional composition. By assigning the function ‘add‘ to the variable ‘f‘, we demonstrate that functions in Python can be manipulated like any other object. This capability allows for flexible and dynamic function usage, promoting code reuse and modularity.

2.2 Higher-Order Functions

Higher-order functions are functions that take other functions as arguments or return functions as their result. They are essential for functional programming as they enable the creation of more abstract and reusable code.

```
    def apply_function(f, x, y):
        return f(x, y)

    result = apply_function(add, 5, 10)
    print(result)  # Output: 15
```

Listing 2: Higher-Order Functions

In this example, `apply_function` is a higher-order function because it takes another function `f` as an argument. This allows us to pass different functions to `apply_function`, making it highly versatile. Higher-order functions are powerful tools for building abstractions and composing behavior in a clean and readable way.

2.3 Anonymous Functions (Lambdas)

Python supports anonymous functions through the `lambda` keyword. These functions are useful for short, throwaway functions that are not reused elsewhere.

```python
square = lambda x: x * x
print(square(4))  # Output: 16
```

Listing 3: Anonymous Functions (Lambdas)

Lambda functions provide a concise way to define simple functions inline. They are particularly useful for functional programming constructs like `map`, `filter`, and `reduce`, where small functions are often passed as arguments. Despite their brevity, lambda functions can be used effectively to create clear and expressive code.

3 Creating a Functional Program in Python

We will create a functional program that processes a list of numbers. The program will filter out even numbers, square the remaining numbers, and then sum them up. This will demonstrate the use of `map`, `filter`, and `reduce` functions.

3.1 Implementation

```
from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filter out even numbers
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))

# Square the remaining numbers
squared_numbers = list(map(lambda x: x * x, odd_numbers))

# Sum the squared numbers
sum_of_squares = reduce(lambda x, y: x + y, squared_numbers)

print(f"The sum of squares of odd numbers is: {sum_of_squares}")
```

Listing 4: Functional Program Demo

3.2 Explanation

This program showcases the core functional programming concepts in Python:

1. Filtering with `filter`:

```
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
```

- The `filter` function constructs an iterator from elements of the iterable (in this case, the list `numbers`) for which the function `lambda x: x % 2 != 0` returns `True`.
- This lambda function checks if a number is odd by computing the remainder of the division by 2.
- The `list` constructor is used to convert the resulting iterator back into a list, containing only the odd numbers.

2. Mapping with `map`:

```
squared_numbers = list(map(lambda x: x * x, odd_numbers))
```

- The `map` function applies the function `lambda x: x * x` to each item of the iterable (the list `odd_numbers`).
- This lambda function squares its input.
- Again, the `list` constructor is used to convert the map object into a list of squared numbers.

3. Reducing with `reduce`:

```
sum_of_squares = reduce(lambda x, y: x + y, squared_numbers)
```

- The `reduce` function from the `functools` module applies the function `lambda x, y: x + y` cumulatively to the items of the iterable (the list `squared_numbers`), from left to right, so as to reduce the iterable to a single value.
- This lambda function sums two numbers.
- The result is the sum of all squared numbers in the list.

These functional constructs — `filter`, `map`, and `reduce` — are powerful tools for data transformation and aggregation, allowing for clear and expressive code. The use of lambda functions in these constructs further enhances the readability and conciseness of the code, making it easy to understand the operations being performed on the data.

4 Advanced Functional Programming Techniques in Python

While the foundational concepts of functional programming in Python are powerful on their own, more advanced techniques can further enhance the expressiveness and efficiency of Python code. This section will explore currying, memoization, and functional data structures, showcasing Python's versatility in supporting sophisticated functional programming methodologies.

4.1 Currying

Currying is a technique where a function is transformed into a sequence of functions, each taking a single argument. This allows for partial application of functions, which can lead to more flexible and reusable code.

```
    def curry(f):
        def curried_function(*args):
            if len(args) == f.__code__.co_argcount:
                return f(*args)
            return lambda *more_args: curried_function(*(args + more_args))
        return curried_function

    @curry
    def add_three_numbers(a, b, c):
        return a + b + c

    add_five = add_three_numbers(2, 3)
    result = add_five(4)  # Output: 9
    print(result)
```

Listing 5: Currying Example

In this example, the `curry` decorator transforms `add_three_numbers` into a curried function. The curried function can be partially applied, enabling flexible and modular code composition.

4.2 Memoization

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again. This is particularly useful in recursive functions and dynamic programming.

```
    def memoize(f):
        cache = {}
        def memoized_function(*args):
            if args not in cache:
                cache[args] = f(*args)
            return cache[args]
        return memoized_function

    @memoize
    def fibonacci(n):
        if n <= 1:
            return n
        return fibonacci(n-1) + fibonacci(n-2)

        print(fibonacci(10))  # Output: 55
```

Listing 6: Memoization Example

In this example, the `memoize` decorator caches the results of the `fibonacci` function. By doing so, it prevents redundant calculations and significantly enhances performance, particularly for functions with overlapping subproblems like the Fibonacci sequence.

4.3 Functional Data Structures

Functional data structures are immutable data structures that can be used in a functional programming style. Python's standard library includes several data structures that support immutability, such as tuples and frozensets. Additionally, third-party libraries like 'pyrsistent' provide more advanced immutable data structures.

```
    from pyrsistent import pvector

    # Create an immutable vector
    vec = pvector([1, 2, 3])
    print(vec)  # Output: pvector([1, 2, 3])

    # Add an element
    new_vec = vec.append(4)
    print(new_vec)  # Output: pvector([1, 2, 3, 4])

    # Original vector remains unchanged
    print(vec)  # Output: pvector([1, 2, 3])
```

Listing 7: Functional Data Structures Example

In this example, 'pyrsistent''s 'pvector' is used to create an immutable vector. Operations on the vector, such as appending an element, return a new vector without modifying the original one, thereby preserving immutability.

5 Advanced Example: Functional Data Processing Pipeline

To illustrate the power of combining these advanced functional programming techniques in Python, we will create a functional data processing pipeline. This pipeline will read data, transform it, and then aggregate the results.

5.1 Pipeline Implementation

```
from functools import partial

# Sample data: list of dictionaries
data = [
    {'name': 'Alice', 'age': 30, 'salary': 70000},
    {'name': 'Bob', 'age': 24, 'salary': 48000},
    {'name': 'Charlie', 'age': 36, 'salary': 120000},
    {'name': 'Diana', 'age': 28, 'salary': 60000},
]

# Function to filter employees by age
def filter_by_age(min_age, person):
    return person['age'] >= min_age

# Function to transform data to extract salaries
def extract_salary(person):
    return person['salary']

# Function to calculate average
def average(salaries):
    return sum(salaries) / len(salaries) if salaries else 0

# Partial application of filter function
filter_adults = partial(filter_by_age, 30)

# Functional pipeline
adult_salaries = map(extract_salary, filter(filter_adults, data))
average_salary = average(list(adult_salaries))

print(f"The average salary of employees aged 30 and above is: {average_salary}
")
```

Listing 8: Functional Data Processing Pipeline

5.2 Explanation

This pipeline demonstrates several key functional programming concepts:

1. **Filtering with Partially Applied Functions**:

```
filter_adults = partial(filter_by_age, 30)
```

- Here, the ‘filter_by_age‘ function is partially applied with the minimum age set to 30 using ‘partial‘ from the ‘functools‘ module. This creates a new function, ‘filter_adults‘, that can be used to filter out individuals below the age of 30.

2. **Mapping with Extracted Functions**:

```
adult_salaries = map(extract_salary, filter(filter_adults, data))
```

- The ‘map‘ function is used to apply ‘extract_salary‘ to each item in the filtered list, resulting in an iterator of salaries for individuals aged 30 and above.

3. **Aggregating with a Custom Average Function**:

```
average_salary = average(list(adult_salaries))
```

- The ‘average‘ function computes the mean salary from the list of salaries obtained from the ‘map‘ function. Converting the iterator to a list ensures all items are processed before calculating the average.

This example showcases the power of functional programming in creating modular, reusable, and readable data processing pipelines in Python.

6 Conclusion

Functional programming is an essential paradigm that brings numerous benefits to software development, including improved modularity, easier reasoning about code, and a higher level of abstraction. Although Python is predominantly an imperative and object-oriented language, it effectively incorporates functional programming constructs, making it a versatile tool for developers.

By leveraging first-class functions, higher-order functions, and anonymous functions, Python enables the creation of concise and expressive functional programs. This report demonstrated how these features can be utilized to process a list of numbers in a functional style, showcasing the elegance and power of functional programming in Python.

The examples provided illustrate the functional programming principles of immutability and function composition, highlighting Python's capability to support these concepts. Additionally, advanced techniques such as currying, memoization, and functional data structures further enhance Python's functional programming capabilities. As the software industry continues to evolve, the ability to blend multiple programming paradigms within a single language becomes increasingly valuable. Python's support for functional programming enhances its flexibility and applicability across various domains and problem sets.

Future work could explore more advanced functional programming techniques in Python, such as currying, memoization, and functional data structures. Additionally, integrating functional programming with Python's rich ecosystem of libraries can further enhance its utility in real-world applications. Furthermore, the study of performance implications and optimization strategies for functional programs in Python could provide valuable insights for developers aiming to build efficient and scalable systems.

7 References

- Python Functional Programming HOWTO
- Real Python - Functional Programming in Python
- Wikipedia - Functional Programming
- Pyrsistent Documentation
- Learn Python - Functional Programming