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

ADVANCED PROGRAMMING (CO2039)

Assignment report

Functional Programming in Python

Advisor: PhD. Truong Tuan Anh
Student: Pham Le Huu Hiep - ID: 2252223

HO CHI MINH CITY, MAY 2024

Contents

1 Introduction

Advanced Programming is a comprehensive course designed for students in the Faculty of Computer Science and Engineering. This course aims to deepen students' understanding of various advanced programming concepts and techniques. It covers an extensive range of topics, providing a solid foundation in both Functional Programming and Object-Oriented Programming.

The curriculum is structured to offer a blend of theoretical knowledge and practical skills. Students will explore the principles and paradigms of Functional Programming, learning how to write clean, efficient, and maintainable code using functions as the primary building blocks. This includes studying key concepts such as first-class functions, higher-order functions, immutability, and recursion.

In addition to Functional Programming, the course delves into Object-Oriented Programming (OOP), emphasizing the design and implementation of software using objects and classes. Students will gain proficiency in creating modular and reusable code, understanding inheritance, polymorphism, encapsulation, and abstraction.

The course also emphasizes practical application, encouraging students to implement Functional Programming principles in various programming languages. Through hands-on projects and assignments, students will practice writing functional code, solve complex problems, and develop a functional programming mindset.

In addition to technical skills, the course fosters critical thinking and problem-solving abilities. Students will be challenged to approach problems from a functional perspective, designing solutions that leverage the power of immutability and pure functions to create robust and scalable software.

By the end of the course, students will have a solid grasp of Functional Programming, equipped with the knowledge and skills to apply this paradigm effectively in their future careers. **Advanced Programming** not only enhances their programming proficiency but also prepares them to tackle sophisticated programming challenges with confidence and creativity.

2 Functional Programming in Python

We'd better start with the hardest question: "What is functional programming (FP), anyway?"

One answer would be to say that functional programming is what you do when you program in languages like Lisp, Scheme, Clojure, Scala, Haskell, ML, OCAML, Erlang, or a few others. That is a safe answer, but not one that clarifies very much. Unfortunately, it is hard to get a consistent opinion on just what functional programming is, even from functional programmers themselves. A story about elephants and blind men seems apropos here. It is also safe to contrast functional programming with "imperative programming" (what you do in languages like C, Pascal, C++, Java, Perl, Awk, TCL, and most others, at least for the most part). Functional programming is also not object-oriented programming (OOP), although some languages are both. And it is not Logic Programming (e.g.,Prolog), but again some languages are multi-paradigm.

Functional programming is characterized by several key features. Programming languages, like Python designed for this paradigm prioritize these features and may make other approaches, like frequently changing variables, difficult or even impossible.

- Functions are first class (objects). That is, everything you can do with "data" can be done with functions themselves (such as passing a function to another function).
- Recursion is used as a primary control structure. In some languages, no other "loop" construct exists.
- There is a focus on list processing (for example, it is the source of the name Lisp). Lists are often used with recursion on sublists as a substitute for loops.
- "Pure" functional languages eschew side effects. This excludes the almost ubiquitous pattern in imperative languages of assigning first one, then another value to the same variable to track the program state.
- Functional programming either discourages or outright disallows statements, and instead works with the evaluation of expressions (in other words, functions plus arguments). In the pure case, one program is one expression (plus supporting definitions).
- Functional programming worries about what is to be computed rather than how it is to be computed.
- Much functional programming utilizes "higher order" functions (in other words, functions that operate on functions that operate on functions).

Advocates of functional programming argue that all these characteristics make for more rapidly developed, shorter, and less bug-prone code. Moreover, high theorists of computer science, logic, and math find it a lot easier to prove formal properties of functional languages and programs than of imperative languages and programs. One crucial concept in functional programming is that of a "pure function"—one that always returns the same result given the same arguments—which is more closely akin to the meaning of "function" in mathematics than that in imperative programming.

Python cannot be classified as a "pure functional programming language" due to the prevalence of side effects in most Python programs. This means that variables are frequently reassigned, mutable data collections often undergo changes, and input/output operations are commonly intertwined with computation. Additionally, Python does not strictly adhere to the principles of functional programming overall. However, Python is a multi-paradigm language that facilitates functional programming when needed and allows for seamless integration with other programming styles.

2.1 Introducing The Essential Functional Concepts

2.1.1 First-Class Functions

Functional programming is often succinct and expressive. One way to achieve it is by providing functions as arguments and return values for other functions. We'll look at numerous examples of manipulating functions.

For this to work, functions must be first-class objects in the runtime environment. In programming languages such as C, a function is not a runtime object. In Python, however, functions are objects that are created (usually) by the def statements and can be manipulated by other Python functions. We can also create a function as a callable object or by assigning lambda to a variable.

Here's how a function definition creates an object with attributes:

```python
def square(x):
return x * x
# Using function as an argument
def apply_func(func, arg):
    return func(arg)
# Using function as a return value
def get_square_function():
    return square
# Using lambda expression
multiply_by_two = lambda x: x * 2
# Example of first-class functions in Python
result1 = apply_func(square, 5)  # Passes square function as an argument
result2 = get_square_function()(5)  # Returns square function as a result
result3 = multiply_by_two(5)  # Uses lambda expression as a function
```

In this example, square is defined as a regular function and is then utilized as an argument for another function **(apply_func)** and as a return value for yet another function **(get_square_function)**. Additionally, a lambda expression **(multiply_by_two)** demonstrates another way to create a function object in Python.

2.1.2 Pure functions

To achieve expressiveness, a function should be devoid of the confusion caused by side effects. Employing pure functions allows for certain optimizations by altering the evaluation order. The major advantage, however, is that pure functions are conceptually simpler and significantly easier to test.

To write a pure function in Python, we must ensure that the code is strictly local, avoiding **global** statements. Careful attention should be given to any use of **nonlocal**, which, although a form of side effect, is restricted to nested function definitions. This standard is straightforward to meet. Pure functions are commonly found in Python programs.

A Python **lambda** is an example of a pure function. Although using **lambda** objects in this way is not highly recommended, it is indeed possible to create pure functions using **lambda** expressions.

Here is how to use the pure functions created with lambda expressions in Python:

1. **Checking if a number is even:**

```python
is_even = lambda x: x % 2 == 0
result = is_even(10)
print(result)  # Output: True
```

2. **Calculating the squares:**

```python
square = lambda x: x ** 2
result = square(27)
print(result)  # Output: 729
```

3. **Concatenating two strings:**

```python
concatenate = lambda s1, s2: s1 + s2
result = concatenate("Hello, ", "world!")
print(result)  # Output: Hello, world!
```

2.1.3 Higher-order functions

Concise and powerful programs can be written by using higher-order functions, which are functions that take other functions as arguments or return functions as results. Through higher-order functions, complex operations can be constructed by combining simpler functions.

Take the Python **sorted()** function, for instance. By passing a function as an argument, the behavior of the **sorted()** can be altered

Here is some examples we might want to consider. Let's assume we have a list of tuples where each tuple contains a student's name and their score on an exam:

```
students_scores = [("Hiep", 99), ("Bob", 75), ("Wang", 93), ("Lee", 85),
↪  ("Mary", 92)]
```

This list can be sorted based on the scores (the second element of each tuple) using the **sorted()** function and a lambda expression since **sorted()** is a higher-order functions. In this case, the lambda is consider as a parameter of the **sorted()** function:

```
sorted_students = sorted(students_scores, key=lambda student: student[1])
print(sorted_students)
# Output in ascending order
[('Bob', 75), ('Lee', 85), ('Mary', 92), ('Wang', 93), ('Hiep', 99)]
```

The list can be sorted in descending order by using **reverse=True** parameter:

```
sorted_students = sorted(students_scores, key=lambda student:
↪  student[1],reverse=True)
print(sorted_students)
# Output in descending order
[ ('Hiep', 99), ('Wang', 93), ('Mary', 92), ('Lee', 85), ('Bob', 75)]
```

2.1.4 Immutable data

Since we are not using variables to maintain the state of a computation, we should focus on using immutable objects. We can extensively use **tuples** and **namedtuples** to create more complex, immutable data structures.

The concept of immutable objects is familiar in Python. Using immutable **tuples** instead of more complex mutable objects can have performance benefits. In some cases, these benefits arise from rethinking the algorithm to avoid the overhead associated with object mutation.

- Tuples:
  Tuples are a simple and common example of immutable data structures in Python.

```python
point = (3, 4)
# Trying to modify the tuple will result in an error
# point[0] = 5  # This will raise a TypeError
```

- Namedtuples:
  Namedtuples provide a way to define immutable, self-documenting tuples.

```python
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(3, 4)
# Trying to modify the namedtuple will result in an error
# p.x = 5  # This will raise an AttributeError
```

- Frozen sets:
  A frozenset is an immutable version of a set.

```python
frozen = frozenset([1, 2, 3])
# Trying to modify the frozenset will result in an error
# frozen.add(4)  # This will raise an AttributeError
```

- Integers, floats, and other simple types:
  Basic data types like integers and floats are also immutable.

```python
num = 42
# You cannot modify the value directly, but you can reassign the
↪  variable
num = 43  # This is a new object, not a modification of the original
```

2.1.5 Strict and Non-strict evaluation

Strict and non-strict evaluation are concepts related to how and when expressions in a programming language are evaluated. Understanding these concepts helps in optimizing code and managing performance and resource usage.

1. Strict Evaluation (Eager Evaluation)
In strict evaluation, also known as eager evaluation, expressions are evaluated as soon as they are bound to a variable. This is the default behavior in Python.

Example:

```
def strict_function(x):
    return x * 2
result = strict_function(5)  # The expression 5 is evaluated immediately.
print(result)  # Output: 10
```

In the example above, the argument **5** is evaluated before the function **strict_function** is called. This is an example of strict evaluation.

2. Non-Strict Evaluation (Lazy Evaluation)

In non-strict evaluation, also known as lazy evaluation, expressions are not evaluated until their values are actually needed. Python does not support lazy evaluation by default, but it can be achieved using certain constructs like generators and the lambda function.

Example using a Generator:

```
def lazy_generator():
    for i in range(5):
        yield i * 2
gen = lazy_generator()
for value in gen:
    print(value)  # Values are computed and printed one by one, only when
    ↪  needed.
```

Values are generated and evaluated one at a time, only when requested by the **for** loop. This is an example of non-strict (lazy) evaluation.

Example using a Lambda Function:

```
lazy_lambda = lambda x: x * 2
def use_lambda(l):
    print("Lambda function is about to be called.")
    return l(5)  # The lambda function is evaluated only when this line is
    ↪  executed.
print(use_lambda(lazy_lambda))  # Output: Lambda function is about to be
↪  called. 10
```

The lambda function is not evaluated when it is defined, but only when it is called within the **use_lambda** function.

3. Practical Applications

- Strict Evaluation:
  - Used in most of Python's built-in functions and standard code execution.
  - Suitable for scenarios where immediate computation is required.
- Non-Strict Evaluation:
  - Useful for improving performance and reducing memory usage, especially with large datasets or infinite sequences.
  - Examples include using generators for reading large files or streams, and the **itertools** module for creating efficient iterators.

```python
import itertools
# Infinite sequence generator
counter = itertools.count(start=1, step=1)

# Take the first 5 elements from the infinite sequence
limited_counter = itertools.islice(counter, 5)

for value in limited_counter:
    print(value)  # Outputs: 1 2 3 4 5
```

In this example, **itertools .count** creates an infinite sequence, but **itertools . islice** lazily evaluates and limits the output to the first 5 elements.

2.1.6 Functional type systems

Functional type systems in Python refer to using type hints and type checking to ensure code correctness, maintainability, and readability in a functional programming style. Python, being a dynamically typed language, allows for flexibility in writing code, but it also supports static type checking through type hints introduced in PEP 484 and tools like **mypy**

1. Type Hints

Type hints allow developers to indicate the expected types of variables, function parameters, and return values. This can improve code readability and help with static analysis.

Basic Example:

```python
def add(x: int, y: int) -> int:
    return x + y
result: int = add(3, 5)
print(result)  # Output: 8
```

In this case, the function add has type hints indicating that it takes two integers and returns an integer.

2. Advanced Type Hints with Functional Programming

In functional programming, functions often take other functions as arguments and return functions. Type hints can be used to specify these higher-order functions.

Higher-Order Function Example:

```python
from typing import Callable

def apply_twice(func: Callable[[int], int], value: int) -> int:
    return func(func(value))

def increment(x: int) -> int:
    return x + 1

result: int = apply_twice(increment, 5)
print(result)  # Output: 7
```

In this application, **apply_twice** takes a function **func** that takes an integer and returns an integer, and an integer value. The type hint **Callable** [[**int**], **int**] is used to specify the type of the function argument.

2.1.7 Some advanced concepts

These concepts are integral to the implementation of purely functional languages. Since Python is not purely functional, a hybrid approach will not require an in-depth focus on these topics. These concepts will be outlined initially for the benefit of those familiar with functional languages like Haskell who are transitioning to Python. While these concerns are universal in programming, they will be addressed differently in Python. Often, imperative programming will be used instead of strictly adhering to functional programming principles.

- **Referential transparency:** In languages with compiled code and optimizations, considering different ways to achieve the same result can be crucial. This is because the compiler can analyze these options and choose the most efficient one. However, in Python, a language interpreted at runtime, this concept is not as significant as there are no significant optimizations happening during compilation.
- **Currying:** Currying is a technique in functional programming that transforms a function with multiple arguments into a sequence of functions that take a single argument each.

While Python is not a purely functional language, it allows you to achieve currying using nested functions.

- **Monads:** These are purely functional constructs that allow us to structure a sequential pipeline of processing in a flexible way. While Python is not a strictly functional language, it can incorporate functional programming concepts like monads. Monads are a design pattern that provide a way to manage side effects (like I/O or mutable state changes) within pure functions.

2.2 Functions, Generators, and Collections

The essence of functional programming lies in using pure functions to transform input values into output values. Pure functions, which have no side effects, are relatively straightforward to implement in Python. Avoiding side effects minimizes reliance on variable assignments to track the state of computations. Although assignment statements cannot be eliminated from Python, dependence on stateful objects can be reduced by selecting Python's built-in functions and data structures that don't require stateful operations.

This section will explore various Python features from a functional programming perspective, including:

- Pure functions with no side effects
- Treating functions as objects that can be passed as arguments or returned as results
- Utilizing Python strings with both object-oriented suffix notation and prefix notation
- Using collections as the primary tool for functional programming design

Generator expressions will be written to accomplish tasks such as:

- Conversions
- Restructuring
- Complex calculations

There will be a brief overview of various built-in Python collections and how to use them within a functional programming framework. This may alter the approach to working with lists, dictionaries, and sets. Emphasizing functional Python encourages a focus on tuples and immutable collections.

2.2.1 Writing pure functions

A function without side effects aligns with the pure mathematical concept of a function, meaning it does not alter any global variables. By avoiding the use of the **global** statement, this goal is nearly achieved. To be truly pure, a function must also refrain from modifying mutable objects' states. The below snippet illustrates an original function with a global variable:

```
global_offset = 5

def calculate_total(x: int, y: int) -> int:
    return x + y + global_offset

```

From this style, it can be refactored to be pure function:

```
def calculate_total(x: int, y: int, offset: int) -> int:
    return x + y + offset
```

In this refactored version, **global_offset** is passed as an parameter, making the function pure by eliminating the dependency on the global state.

Python has numerous internal stateful objects, such as instances of the file class and other file-like objects. These objects are commonly used and often act as context managers. When a stateful object does not fully implement the context manager interface, the **close()** method can be used. For such cases, the **contextlib.closing()** function ensures proper context management.

Completely eliminating stateful Python objects is not feasible. Therefore, it is crucial to balance state management with the benefits of functional programming. To achieve this, the with statement should always be used to encapsulate stateful file objects within a defined scope.

It is advisable to always use file objects within a **with** context. Global file objects, global database connections, and related stateful object issues should be avoided. The global file object pattern, commonly used for handling open files, is problematic. Consider the following example:

```
def open(iname: str, oname: str)
    global ifile,ofile
    ifile = open(iname, "r")
    ofile = open(oname, "w")
```

In this context, other functions can use the **ifile** and **ofile** variables, assuming they correctly reference the global files, which remain open for the application to use.

This design is not very functional and should be avoided. Instead, files should be passed as parameters to functions, and open files should be managed within a **with** statement to ensure their state is properly handled. This refactoring is crucial to move from global variables to formal parameters, making file operations more transparent and controlled.

This design pattern is also relevant to databases. A database connection object should typically be passed as a formal argument to an application's functions. This approach contrasts with the practices of some popular web frameworks that use a global database connection to make the database a seamless part of the application. However, this transparency can obscure the dependency between a web operation and the database, complicating unit testing. Moreover, a multi-threaded web server might not benefit from a single shared database connection; using a connection pool is often more effective. Therefore, a hybrid approach that incorporates functional design with a few isolated stateful features can be advantageous.

2.2.2 Using strings

Python strings are immutable, making them ideal for functional programming as they produce new strings without side effects. Methods of Python's string objects are pure functions.

String method syntax in Python is post-fix, unlike the prefix notation of most functions, which can be confusing when used together. For instance, in **len(variable.title ())**, **title ()** uses post-fix notation, while **len()** uses prefix notation. For web scraping, a function may be needed to clean data, involving a mix of prefix and post-fix notations to process the string and convert it to a **Decimal** object. An example of such a function is:

```python
from decimal import *
from typing import Text, Optional

def clean_decimal(text: Text) -> Optional[Text]:
    if text is None: return None
    return Decimal(
        text.replace("$", "").replace(",", "")
    )
```

This function removes **$** and , from the string and then uses the cleaned string as input for the **Decimal** class. If the input is **None**, it returns **None**, maintaining the type hint. To make the syntax more uniform, one could define custom prefix functions for string methods:

```python
def replace(str: Text, a: Text, b: Text) -> Text:
    return str.replace(a, b)
```

Using this function allows for more consistent-looking syntax although it is debatable if this kind of consistency is genuinely beneficial, as it might be an example of unnecessary uniformity:

```python
def replace(str: Text, a: Text, b: Text) -> Text:
    return str.replace(a, b)

text = "$1,234.56"

# Using the replace function to clean the text and then convert to Decimal
cleaned_decimal = Decimal(replace(replace(text, "$", ""), ",", ""))
print(cleaned_decimal)
```

2.2.3 Using tuples and named tuples

A more effective approach might be to create a more meaningful prefix function to remove punctuation, as shown in the following command snippet:

```python
def remove(str: Text, chars: Text) -> Text:
    if chars:
        return remove(
            str.replace(chars[0], ""),
            chars[1:]
        )
    return str
```

This function recursively removes each character listed in the chars variable from the input string. It can be used like the way below, which clarifies the intention of the string cleanup process:

```python
text = "$1,234.56"

# Using the remove function to clean the text and then convert to Decimal
cleaned_decimal = Decimal(remove(text, "$,"))
print(cleaned_decimal)
```

2.2.4 Using generator expressions

Generator expressions are a powerful feature in Python that can be very useful in functional programming for their efficiency and simplicity. They are particularly useful in functional programming for creating iterators on-the-fly, allowing for efficient memory usage and lazy evaluation.

A generator expression is similar to a list comprehension, but instead of creating a list, it returns a generator object. This generator object can be iterated over, and it produces items one at a time and only when required. This makes generator expressions more memory efficient than list comprehensions, especially when dealing with large datasets.

The syntax of a generator expression is similar to that of a list comprehension, but it uses parentheses `()` instead of square brackets `[]`.

```
(expression for item in iterable if condition)
```

- **expression**: The value to be yielded by the generator.
- **item**: The variable representing each element in the iterable.
- **iterable**: The collection or sequence to iterate over.
- **condition** (optional): A filtering condition.

Here is a simple example of a generator expression that produces squares of numbers:

```
squares = (x**2 for x in range(10))
print(squares)  # Output: <generator object <genexpr> at 0x...>

# To get the values, we can iterate over the generator
for square in squares:
    print(square)
```

In functional programming, generator expressions can be combined with several functions which accept an iterable as an argument.

```
# Sum of squares of numbers from 0 to 9
sum_of_squares = sum(x**2 for x in range(10))
print(sum_of_squares)  # Output: 285

# Maximum square less than 100
max_square = max(x**2 for x in range(10) if x**2 < 100)
print(max_square)  # Output: 81

# Checking if any square is greater than 50
```

```python
any_square_gt_50 = any(x**2 > 50 for x in range(10))
print(any_square_gt_50)  # Output: True
```

Further conditions can also be added to filter items before they are processed by the generator expression.

```python
# Generating squares of even numbers only
even_squares = (x**2 for x in range(10) if x % 2 == 0)
for square in even_squares:
    print(square)
# Output:
# 0
# 4
# 16
# 36
# 64
```

Generator expressions can be nested, despite the fact that this style can make the code more complex and harder to read.

```python
# Flatten a 2D list using a nested generator expression
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = (element for row in matrix for element in row)
for num in flattened:
    print(num)
# Output:
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

Generator expressions in Python offer significant advantages in terms of memory efficiency, lazy evaluation, and conciseness.

- **Memory Efficiency:** Generator expressions do not generate the entire sequence at once, which makes them memory efficient.
- **Lazy Evaluation:** Values are generated only when needed, which can improve performance in certain scenarios.
- **Conciseness:** They provide a succinct way to create generators.

2.2.5 Using lists, dictionaries, and sets

In the context of Python and functional programming, "collection" generally refers to data structures that can hold multiple items. These structures are essential in functional programming as they often support operations like mapping, filtering, and reducing. Here are the main types of collections used in Python: lists, dictionaries, sets, tuples.

In Python, lists, dictionaries, and sets are essential data structures that are frequently used in functional programming. Understanding how to use these data structures effectively within this paradigm is crucial.

1. Lists

Lists are ordered, mutable collections that can hold a variety of objects. They support various operations such as iteration, filtering, and mapping, making them highly versatile in functional programming.

The snippet below uses list comprehensions and the **map** and **filter** functions for functional programming:

```python
numbers = [1, 2, 3, 4, 5]

# Using map to square each number
squares = list(map(lambda x: x**2, numbers))
print(squares)  # Output: [1, 4, 9, 16, 25]

# Using filter to get even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4]
```

2. Dictionaries

Dictionaries are unordered, mutable collections that store data in key-value pairs. They are particularly useful for mapping relationships between data and are also iterable, allowing functional programming techniques to be applied.

Let's consider the example below which uses dictionary comprehensions and the **map** function to transform dictionaries:

```
# Using dictionary comprehension to create a new dictionary with squared
↪  values
numbers_dict = {'a': 1, 'b': 2, 'c': 3}
squares_dict = {k: v**2 for k, v in numbers_dict.items()}
print(squares_dict)  # Output: {'a': 1, 'b': 4, 'c': 9}

# Using map with dictionaries (map returns an iterator)
keys = numbers_dict.keys()
values_squared = map(lambda x: x**2, numbers_dict.values())
squares_dict_map = dict(zip(keys, values_squared))
print(squares_dict_map)  # Output: {'a': 1, 'b': 4, 'c': 9}
```

3. Sets

Sets are unordered collections of unique elements. They are useful for operations involving membership testing, de-duplication, and mathematical operations like union, intersection, and difference.

The example of using set comprehensions and functional operations on sets is shown below:

```
# Using set comprehension to create a new set with squared values
numbers_set = {1, 2, 3, 4, 5}
squares_set = {x**2 for x in numbers_set}
print(squares_set)  # Output: {1, 4, 9, 16, 25}

# Using filter to create a subset
evens_set = set(filter(lambda x: x % 2 == 0, numbers_set))
print(evens_set)  # Output: {2, 4}

# Set operations: union, intersection, and difference
set_a = {1, 2, 3}
set_b = {2, 3, 4}
union_set = set_a | set_b
intersection_set = set_a & set_b
difference_set = set_a - set_b
print(union_set)        # Output: {1, 2, 3, 4}
print(intersection_set) # Output: {2, 3}
print(difference_set)    # Output: {1}
```

2.3 Immutable Data Structures

In functional programming, immutability is a core concept that ensures data cannot be modified after it is created. This leads to more predictable and reliable code, as it eliminates side effects and makes functions easier to reason about and test. Python provides several immutable data structures and libraries that support immutability, which are crucial for writing functional programs.

Python has several built-in immutable data types as shown in the below:

1. Tuples:

- Tuples are ordered collections of items that cannot be changed after creation.
- They are defined using parentheses `()`.

```
my_tuple = (1, 2, 3)
```

- Trying to alter a tuple will raise a TypeError.

```
my_tuple[0] = 10  # Raises TypeError
```

2. Strings:

- Strings in Python are immutable sequences of characters.
- Any operation that modifies a string returns a new string rather than modifying the original.

```
my_string = "hello"
new_string = my_string.upper()  # Returns 'HELLO', my_string remains
↪  'hello'
```

3. Frozensets:

- Frozensets are immutable versions of sets.

- They are defined using the **frozenset()** function.

```
my_set = frozenset([1, 2, 3])
```

- Like sets, frozensets support membership tests and other set operations, but they cannot be modified.

```
my_set.add(4)  # Raises AttributeError
```

Immutability has several benefits in Functional Programming that consists of:

- Predictability: Immutable objects do not change state, so they are predictable and easier to understand.
- Thread Safety: Immutable objects can be shared between threads without concern for concurrent modifications.
- Caching and Memoization: Since immutable objects do not change, they can be safely cached and reused.

In addition to built-in types, Python offers libraries that provide immutable collections. One notable library is **pyrsistent**. It is a library that provides persistent (immutable) data structures. These structures are designed to be modified without changing the original, allowing for safe and efficient state transformations.

- Persistent List (PList): it is similar to Python's list but immutable.

```
from pyrsistent import plist
my_plist = plist([1, 2, 3])
new_plist = my_plist.append(4)  # Returns a new plist, original
↪ remains unchanged
```

- Persistent Dictionary (PMap): it is similar to Python's dictionary but immutable.

```
from pyrsistent import pmap
my_pmap = pmap({'a': 1, 'b': 2})
new_pmap = my_pmap.set('c', 3)  # Returns a new pmap, original
↪ remains unchanged
```

- Persistent Set (PSet): it is similar to Python's set but immutable.

```
from pyrsistent import pset
my_pset = pset([1, 2, 3])
new_pset = my_pset.add(4)  # Returns a new pset, original remains
↪  unchanged
```

For instance, here is a simple example that demonstrates the use of immutable data structures in a functional style of Python.

```
from pyrsistent import pmap

def update_inventory(inventory, item, quantity):
    # Return a new inventory with the updated quantity
    return inventory.set(item, inventory.get(item, 0) + quantity)

# Initial inventory
inventory = pmap({'apples': 10, 'oranges': 5})

# Update inventory
new_inventory = update_inventory(inventory, 'apples', 5)

print(inventory)      # pmap({'apples': 10, 'oranges': 5})
print(new_inventory)  # pmap({'apples': 15, 'oranges': 5})
```

In this example, **inventory** remains unchanged after the update operation, demonstrating the power of immutability in maintaining predictable and reliable state transformations.

By leveraging immutable data structures, functional programs in Python is written more robust and the source code is maintainable.

2.4 Lambda Functions

Lambda functions, also known as anonymous functions, are a key feature of functional programming in Python. They allow you to create small, throwaway functions without the need for formally defining them using def. This makes your code more concise and can be particularly useful when a function is needed only once or for a short period.

2.4.1 Defining Lambda Functions

Lambda functions are defined using the lambda keyword, followed by a list of parameters, a colon, and an expression. The syntax is:

```
lambda parameters: expression
```

Based on this syntax, an example is shown below:

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

This defines a lambda function that takes two arguments, x and y, and returns their sum.

2.4.2 Use Cases for Lambda Functions

Lambda functions are commonly used with higher-order functions such as **map()**, **filter** (), and **reduce()**.

- Using **map()** with Lambda Functions:

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16, 25]
```

- Using **filter** () with Lambda Functions:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4]
```

- Using **reduce()** with Lambda Functions:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 120
```

2.4.3 Pros and Cons

Lambda functions themselves have advantages associative with limitations. These aspects are consider below:

- Advantages
  - Conciseness: Lambda functions can make code more concise, especially when the function logic is simple.
  - Readability: They can enhance readability by keeping the function definition close to its usage.
  - No Need for Naming: Lambda functions do not require a name, which is useful for short-lived operations.
- Limitations
  - Limited Functionality: Lambda functions are limited to a single expression and cannot include statements or annotations.
  - Readability: For complex operations, lambda functions can decrease readability as they lack the descriptive power of named functions.

2.4.4 Application

In fact, using Lambda functions helps developers handle several issues by applying them beside some additional features:

- Sorting with **sorted()**: Lambda functions are often used as the key function in sorting.

```python
words = ["apple", "banana", "cherry", "date"]
sorted_words = sorted(words, key=lambda word: len(word))
print(sorted_words)  # Output: ['date', 'apple', 'banana', 'cherry']
```

- Event Handling: Lambda functions are useful in GUI applications for handling events.

```python
import tkinter as tk

root = tk.Tk()
button = tk.Button(root, text="Click me", command=lambda:
↪  print("Button clicked!"))
button.pack()
root.mainloop()
```

- Using with **apply()** in Pandas: Lambda functions are handy when applying functions to **DataFrame** columns or rows in Pandas.

```
import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3], 'B': [10, 20, 30]})
df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
print(df)
# Output:
#    A   B   C
# 0  1  10  11
# 1  2  20  22
# 2  3  30  33
```

2.4.5 Summary

In conclusion, lambda functions in Python are a powerful tool for functional programming, allowing for the creation of small, anonymous functions on the fly. They are particularly useful in conjunction with higher-order functions and can significantly enhance the expressiveness and conciseness of your code. However, they should be used judiciously, as overuse or use in complex situations can lead to reduced readability.

2.5 Higher-order functions

A key feature of functional programming is the use of higher-order functions, which are functions that take other functions as arguments or return functions as results. Python provides several types of higher-order functions. These can be categorized as:

- Functions that accept one or more functions as arguments
- Functions that return another function
- Functions that both accept a function and return a function, combining the previous two types

Python includes numerous built-in higher-order functions of the first type, which will be discussed in this section. The concept of a function that generates other functions may seem unusual. However, when considering the **Callable** class, defining a class effectively creates a function that produces **Callable** objects when evaluated. This serves as an example of a function that generates another function.

Higher-order functions that both accept and return functions can include complex callable classes and function decorators. Decorators will be introduced in this section, with a more detailed examination covered in the section "Decorator Design Techniques."

Additionally, there is a desire for higher-order versions of the collection functions from the previous section. This chapter will illustrate the **reduce(extract())** design pattern to perform reductions on specific fields extracted from a larger tuple. It will also define custom versions of common collection-processing functions.

In this section, the following functions will be consider:

- **max()** and **min()**
- **map()**
- **filter ()**
- **iter ()**
- **sorted()**

The **max()** and **min()** functions are reductions; they create a single value from a collection. The other functions are mappings. They do not reduce the input to a single value. The **max()**, **min()**, and **sorted()** functions have both a default behavior as well as a higher-order function behavior. A function can be provided via the **key=** argument. The **map()** and **filter ()** functions take the function as the first positional argument.

2.5.1 Finding the Maximum and Minimum values using map()

The function **map()** applies a function to every item in an iterable (like a list) and returns a map object (which can be converted to a list or another iterable).

Suppose you have a list of numbers and you want to find the maximum and minimum of the list after applying some transformation:

```python
numbers = [1, 2, 3, 4, 5]

# Find max and min of the squares of the numbers
max_value = max(map(lambda x: x**2, numbers))
min_value = min(map(lambda x: x**2, numbers))

print("Max of squares:", max_value) # Max of squares: 25
print("Min of squares:", min_value) # Min of squares: 1
```

2.5.2 Finding the Maximum and Minimum values using **filter ()**

The **filter ()** creates a new iterable by including only the items for which the provided function returns **True**.

Suppose you have a list of numbers and you want to find the maximum and minimum values among the even numbers only.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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

max_value = max(even_numbers)
min_value = min(even_numbers)

print("Max of even numbers:", max_value) # Max of even numbers:10
print("Min of even numbers:", min_value) # Min of even numbers:2
```

2.5.3 Finding the Maximum and Minimum values using reduce()

The function **reduce()** from the **functools** module applies a function cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value.

To find the maximum and minimum using **reduce()**, it is necessary to import it from the **functools** module. The function **reduce()** should take two arguments and return one, applying it cumulatively to the items of the iterable.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

# Use reduce to find the maximum value
max_value = reduce(lambda a, b: a if a > b else b, numbers)
# Use reduce to find the minimum value
min_value = reduce(lambda a, b: a if a < b else b, numbers)

print("Max value:", max_value) # Max value:5
```

```python
print("Min value:", min_value) # Min value:1
```

2.5.4 Data management problem that can be applied sorted() and iter ()

Suppose you have a list of student records where each record is a dictionary containing the student's name and their score. You want to find the top **N** students with the highest scores and list their names in alphabetical order.

The behavior of the two functions that can be useful in solving this kind of problem:

- Using **iter** (): The **iter** () function is used to get an iterator from an iterable. We can use it in scenarios where we need to manually control iteration or convert iterables to iterators.
- Using **sorted**(): The **sorted**() function returns a new sorted list from the elements of any iterable. It can take a key function to customize the sorting order.

Now, let's consider the below algorithm:

1. Define the list of student records.
2. Sort the records by score in descending order using **sorted**() with a key function.
3. Extract the top **N** students from the sorted list.
4. Sort the names of the top **N** students in alphabetical order using **sorted**().
5. Use **iter** () to create an iterator and manually iterate over the top **N** sorted names.

From these steps, the implementation can be provided as below:

```python
# List of student records
students = [
    {'name': 'Alice', 'score': 85},
    {'name': 'Bob', 'score': 92},
    {'name': 'Charlie', 'score': 88},
    {'name': 'David', 'score': 91},
    {'name': 'Eve', 'score': 79}
]

# Define the number of top students to find
N = 3

# Step 1: Sort the records by score in descending order
```

```
sorted_by_score = sorted(students, key=lambda x: x['score'], reverse=True)

# Step 2: Extract the top N students
top_students = sorted_by_score[:N]

# Step 3: Extract names of the top N students and sort them alphabetically
top_names_sorted = sorted(student['name'] for student in top_students)

# Step 4: Use iter() to create an iterator
top_names_iterator = iter(top_names_sorted)

# Manually iterate over the iterator and print each name
print("Top N students in alphabetical order:")
for name in top_names_iterator:
    print(name)
"""
Output:
Top N students in alphabetical order:
Bob
Charlie
David
"""
```

2.6 Functional Programming Modules in Python

Python provides several modules that support functional programming paradigms, allowing developers to write cleaner, more efficient, and more expressive code. Two of the most notable modules are **functools** and **itertools** .

1. **functools** Module

The **functools** module provides higher-order functions and operations on callable objects, enhancing Python's functional programming capabilities.

- **reduce()** is used to apply a rolling computation to a sequence of elements. It takes a function and a sequence and returns a single value.

```
from functools import reduce
```

```python
numbers = [1, 2, 3, 4, 5]
result = reduce(lambda x, y: x + y, numbers)
print(result)  # Output: 15
```

- **partial ()** allows you to fix a certain number of arguments of a function and generate a new function.

```python
from functools import partial

def multiply(x, y):
    return x * y

double = partial(multiply, 2)
print(double(5))  # Output: 10
```

- **lru_cache ()** is a decorator that provides a Least Recently Used (LRU) cache for function results, improving performance by storing results of expensive function calls.

```python
from functools import lru_cache

@lru_cache(maxsize=32)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

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

2. **itertools** Module

The **itertools** module offers a suite of fast, memory-efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

- **count()** generates consecutive integers, starting from a specified number.

```python
from itertools import count

for i in count(10):
    if i > 15:
        break
    print(i)
# Output: 10, 11, 12, 13, 14, 15
```

- **cycle()** cycles through an iterable indefinitely.

```
from itertools import cycle

counter = 0
for item in cycle(['A', 'B', 'C']):
    if counter > 6:
        break
    print(item)
    counter += 1
# Output: A, B, C, A, B, C, A
```

- **repeat()** repeats an object, either indefinitely or a specified number of times.

```
from itertools import repeat

for item in repeat('Hello', 3):
    print(item)
# Output: Hello, Hello, Hello
```

- **chain()** combines multiple iterables into a single iterator.

```
from itertools import chain

combined = chain([1, 2, 3], ['a', 'b', 'c'])
print(list(combined))
# Output: [1, 2, 3, 'a', 'b', 'c']
```

- **islice ()** provides slicing functionality for iterators.

```
from itertools import islice

```

```python
items = range(10)
sliced = islice(items, 2, 8, 2)
print(list(sliced))
# Output: [2, 4, 6]
```

- **combinations()** generates all possible combinations of a specified length from the input iterable, while **permutations()** generates all possible permutations of a specified length from the input iterable.

```python
from itertools import combinations, permutations

items = ['A', 'B', 'C']
comb = list(combinations(items, 2))
perm = list(permutations(items, 2))

print(comb)  # Output: [('A', 'B'), ('A', 'C'), ('B', 'C')]
print(perm)  # Output: [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B',
↪  'C'), ('C', 'A'), ('C', 'B')]
```

By leveraging these modules, Python developers can write more efficient and expressive functional code, harnessing the power of iterators, higher-order functions, and functional programming paradigms.

2.7 Recursion

Recursion is a fundamental concept in functional programming where a function calls itself to solve a smaller instance of the same problem. Python supports recursion, and it can be a powerful tool for tasks that can be broken down into smaller, repetitive sub-tasks. Here, let's consider a few examples of how to use recursion effectively in Python, often in the context of functional programming and some limited problems of these:

2.7.1 Factorial Function

Calculate the product of all positive integers up to a given number **n**. This can be done recursively by recognizing that $n! = n \times (n-1)!$.

1. Recursion definition

- Base Case: $0! = 1$

- Recursive Case: $n! = n \times (n-1)!$

2. Implementation and Testing

```python
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:       # Recursive case
        return n * factorial(n - 1)

# Test
print(factorial(5))  # Output: 120
```

2.7.2 Fibonacci Sequence

Generate the $n-th$ Fibonacci number where each number is the sum of the two preceding ones, starting from 0 and 1.

1. Recursion definition

- Base Cases: $fib(0) = 0, fib(1) = 1$
- Recursive Case: For $n > 1, fib(n) = fib(n-1) + fib(n-2)$

2. Implementation and Testing

```python
def fibonacci(n):
    if n <= 0:  # Base case 1
        return 0
    elif n == 1:  # Base case 2
        return 1
    else:  # Recursive case
        return fibonacci(n - 1) + fibonacci(n - 2)

# Test
print(fibonacci(5))  # Output: 5 (0, 1, 1, 2, 3, 5)
```

2.7.3 Sum of a List

Calculate the sum of all elements in a list by adding the first element to the sum of the remaining elements.

1. Recursion definition

- Base Cases: The sum of an empty list is 0.
- Recursive Case: For a non-empty list, the sum is the first element plus the sum of the rest of the list.

2. Implementation and Testing

```python
def sum_list(lst):
    if not lst:  # Base case
        return 0
    else:  # Recursive case
        return lst[0] + sum_list(lst[1:])
# Test
print(sum_list([1, 2, 3, 4, 5]))  # Output: 15
```

2.7.4 Depth-First Search (DFS) in a Binary Tree

Traverse a binary tree in a depth-first manner, visiting nodes as deeply as possible along each branch before backtracking.

1. Recursion definition

- Base Cases: If the node is **None**, return an empty list.
- Recursive Case: Visit the current node, then recursively visit the left sub-tree, followed by the right sub-tree.

2. Implementation and Testing

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

def dfs(root):
    if root is None:  # Base case
        return []
    else:  # Recursive case
        return [root.value] + dfs(root.left) + dfs(root.right)

# Test
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

print(dfs(root))  # Output: [1, 2, 4, 5, 3]
```

2.7.5 Handling Recursion Limits

Python has a recursion limit to prevent infinite recursion from crashing the program. You can check and set the recursion limit using the **sys** module:

```
import sys

# Check the current recursion limit
print(sys.getrecursionlimit())

# Set a new recursion limit
sys.setrecursionlimit(2000)
```

2.7.6 Tail Recursion Optimization

Python does not optimize tail recursion (a special case where the recursive call is the last operation in the function), so deeply nested recursion can still lead to a stack overflow. For problems where recursion depth might become an issue, consider using an iterative approach or refactor the recursive solution.

2.7.7 Tail-Recursive Factorial

To mimic tail recursion optimization, you can use a helper function with an accumulator:

```python
def factorial_tail_recursive(n, accumulator=1):
    if n == 0:
        return accumulator
    else:
        return factorial_tail_recursive(n - 1, n * accumulator)

# Test the function
print(factorial_tail_recursive(5))  # Output: 120
```

By using an accumulator, this function effectively keeps the state in each recursive call, reducing the problem size each time until it reaches the base case.

2.7.8 Summary

To conclude, Recursion is a powerful tool in functional programming and can simplify the implementation of many algorithms and data structure manipulations. However, be mindful of Python's recursion limits and the potential for stack overflow with deep recursion. When necessary, consider iterative solutions or optimizations like tail recursion to ensure efficiency and avoid runtime errors.

2.8 Lazy Evaluation

Lazy evaluation is a programming technique where an expression is not evaluated until its value is actually needed. This approach can improve performance by avoiding unnecessary calculations and can help manage memory usage more efficiently. Python supports lazy evaluation primarily through **itertools** module.

The benefits included:

- Performance Improvement: Avoids the overhead of computing intermediate values that may never be used.
- Memory Efficiency: Reduces memory footprint by generating values only when needed.
- Handling Infinite Sequences: Enables the representation and manipulation of infinite sequences, which would be impossible with eager evaluation.

```
# count(): Generates an infinite sequence of numbers, starting from the
↪  specified number.
from itertools import count
counter = count(start=10, step=2)
for _ in range(5):
    print(next(counter), end=' ')  # Output: 10 12 14 16 18

# cycle(): Cycles through an iterable indefinitely.
from itertools import cycle
colors = cycle(['red', 'green', 'blue'])
for _ in range(6):
    print(next(colors), end=' ')  # Output: red green blue red green blue

# islice(): Slices an iterator, returning selected elements.
from itertools import islice
naturals = count(1)
sliced = islice(naturals, 10, 20)  # Take values from 10th to 19th
print(list(sliced))  # Output: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

# chain(): Combines multiple iterables into a single iterator.
from itertools import chain
combined = chain([1, 2, 3], ['a', 'b', 'c'])
print(list(combined))  # Output: [1, 2, 3, 'a', 'b', 'c']

#takewhile(): Takes elements from an iterator as long as a condition is true.
from itertools import takewhile
naturals = count(1)
taken = takewhile(lambda x: x <= 5, naturals)
print(list(taken))  # Output: [1, 2, 3, 4, 5]
```

2.9 Comprehensions

Comprehensions are a concise way to create sequences such as lists, dictionaries, and sets in Python. They provide a syntactic construct to generate these collections from existing sequences in an efficient and readable manner. In functional programming, comprehensions can be seen as a way to express mapping and filtering operations, which are fundamental concepts.

2.9.1 List Comprehensions

List comprehensions provide a succinct way to create lists. The basic syntax is:

```
[expression for item in iterable if condition]
```

- **expression** is the value to be included in the list.
- **item** is the variable that takes each value from the iterable.
- **iterable** is the collection being iterated over.
- **condition** is an optional filter that determines if the expression should be included.

```
# List of squares of even numbers from 0 to 9
squares = [x**2 for x in range(10) if x % 2 == 0]
print(squares)
# Output: [0, 4, 16, 36, 64]
```

In this example, the list comprehension iterates over the range of numbers from 0 to 9, squares each number, and includes it in the resulting list only if it is even.

2.9.2 Dictionary Comprehensions

Dictionary comprehensions follow a similar syntax but are used to create dictionaries:

```
{key_expression: value_expression for item in iterable if condition}
```

The snippet code below shows the example

```
# Dictionary of numbers and their squares for even numbers from 0 to 9
squares_dict = {x: x**2 for x in range(10) if x % 2 == 0}
print(squares_dict)
# Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Here, the dictionary comprehension creates a dictionary where each key-value pair consists of a number and its square, but only for even numbers.

2.9.3 Set Comprehensions

Set comprehensions are used to create sets and have a syntax similar to list comprehensions, but use curly braces:

```
{expression for item in iterable if condition}
```

Let's consider the example of using set comprehension

```python
# Set of squares of even numbers from 0 to 9
squares_set = {x**2 for x in range(10) if x % 2 == 0}
print(squares_set)
# Output: {0, 64, 4, 36, 16}
```

This example demonstrates the creation of a set containing the squares of even numbers from 0 to 9.

2.9.4 Nested Comprehensions

Finally, Comprehensions can be nested to handle more complex data structures. Nested comprehensions are useful when dealing with matrices or nested lists.

```python
# Matrix transpose using nested list comprehensions
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

transpose = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
print(transpose)
# Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

In this example, the outer list comprehension iterates over the column indices, and the inner list comprehension iterates over the rows, effectively transposing the matrix.

2.10 Pattern Matching (PEP 634)

Pattern matching, introduced in Python 3.10 through PEP 634, is a powerful feature that allows you to match complex data structures in a concise and readable way. Inspired by similar features in functional programming languages like Haskell and Scala, Python's pattern matching can greatly enhance the expressiveness and clarity of code that deals with structured data.

2.10.1 Basic Syntax

Pattern matching in Python uses the match statement, which is analogous to the switch statement in other languages but with more powerful matching capabilities. The basic syntax is:

```python
match subject:
    case pattern1:
        # Code to execute if pattern1 matches
    case pattern2:
        # Code to execute if pattern2 matches
    case _:
        # Code to execute if no other pattern matches (optional)
```

- **subject** is the value being matched against the patterns.
- **pattern1, pattern2, etc .,** are the patterns being matched.
- '_' is a wildcard pattern that matches anything, serving as a default case.

2.10.2 Matching Literals and Variables

Specific values or bind variables to parts of the data structure can be matched:

```python
def handle_status_code(code):
    match code:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case 500:
            return "Server Error"
        case _:
            return "Unknown Code"
```

The function **handle_status_code** returns a message based on the status code provided.

2.10.3 Matching Sequences

Pattern matching can be used to match and destructure sequences such as lists and tuples:

```
def process_sequence(seq):
    match seq:
        case []:
            return "Empty sequence"
        case [single]:
            return f"Single element: {single}"
        case [first, second]:
            return f"Two elements: {first}, {second}"
        case [first, *rest]:
            return f"First element: {first}, rest: {rest}"
```

Here, the function process_sequence processes sequences differently based on their length and structure.

2.10.4 Matching Data Classes and Named Tuples

Pattern matching is particularly powerful when used with data classes or named tuples, allowing you to match and extract data from structured objects:

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

def describe_point(point):
    match point:
        case Point(x, y) if x == y:
            return f"Point is on the diagonal at ({x}, {y})"
        case Point(x, y):
            return f"Point is at ({x}, {y})"
```

The **describe_point** function matches a Point object and provides different descriptions based on its coordinates.

2.10.5 Combining Patterns with Guards

Guards can be added to patterns to specify additional conditions that must be met for the pattern to match:

```
def categorize_number(num):
    match num:
        case n if n < 0:
            return "Negative number"
        case n if n == 0:
            return "Zero"
        case n if n > 0:
            return "Positive number"
```

The **categorize_number** function categorizes a number as negative, zero, or positive using guards.

2.10.6 Benefits of Pattern Matching

Pattern Matching is useful as several features:

- Expressiveness: Pattern matching allows for concise and readable code that clearly expresses the structure of the data being matched.
- Clarity: By using patterns, you can avoid deeply nested conditionals and make your code easier to understand.
- Functional Style: Pattern matching aligns well with functional programming paradigms, where matching and destructuring data is a common practice.

2.10.7 Functional Style Data Processing

Combining pattern matching with functional programming techniques can lead to elegant and powerful solutions. Here's an example of processing a list of mixed data types:

```python
def process_items(items):
    results = []
    for item in items:
        match item:
            case int():
                results.append(item * 2)
            case str() if item.isdigit():
                results.append(int(item) * 2)
            case str():
                results.append(item.upper())
            case _:
                results.append(None)
    return results

items = [1, '2', 'hello', 3.5, 'world']
print(process_items(items))
# Output: [2, 4, 'HELLO', None, 'WORLD']
```

In this example, process$_{i}temsusespatternmatchingtohandledifferenttypesofitemsinalist, applyingappropriatetran$

2.10.8 Summary

Pattern matching, as introduced in PEP 634, is a versatile and powerful feature that enhances Python's functional programming capabilities. By allowing you to match and destructure complex data structures succinctly, it enables you to write cleaner, more readable, and more expressive code.

2.11 Decorator Design Techniques

Decorator design techniques in Python are a powerful way to modify or enhance the behavior of functions or methods. Decorators are higher-order functions that take another function as an argument and extend its behavior without explicitly modifying it. In functional programming, decorators provide a clean and readable way to extend functionality, promote code reuse, and separate concerns.

2.11.1 Basic Structure of a Decorator

A decorator is essentially a higher-order function that takes a function as an argument and returns a new function with extended behavior. The basic syntax involves defining a decorator and applying it to a function using the @ symbol.

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        # Code to execute before the function call
        result = func(*args, **kwargs)
        # Code to execute after the function call
        return result
    return wrapper

@my_decorator
def my_function():
    print("Hello, World!")

my_function()
```

In this example, **my_decorator** adds behavior before and after the execution of **my_function.**

2.11.2 Common Use Cases for Decorators

1. **Logging:** Decorators can be used to log function calls, arguments, and results.

```python
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned: {result}")
        return result
    return wrapper

@log_decorator
def add(a, b):
    return a + b

add(5, 3)
```

```
# Output:
# Calling add with args: (5, 3), kwargs: {}
# add returned: 8
```

2. **Access Control:** Decorators can enforce access control and authentication.

```
def require_auth(func):
    def wrapper(user, *args, **kwargs):
        if not user.is_authenticated:
            raise PermissionError("User is not authenticated")
        return func(user, *args, **kwargs)
    return wrapper

@require_auth
def get_data(user):
    return "Sensitive data"

# Example usage
class User:
    def __init__(self, is_authenticated):
        self.is_authenticated = is_authenticated

user = User(is_authenticated=True)
print(get_data(user))  # Output: Sensitive data
```

3. **Memoization:** Decorators can be used to cache results of expensive function calls.

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
```

```python
def fibonacci(n):
    if n in {0, 1}:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(30))  # Output: 832040
```

2.11.3 Decorators with Arguments

Decorators can accept arguments to customize their behavior. This is done by defining a decorator factory that takes arguments and returns a decorator.

```python
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
# Output:
# Hello, Alice!
# Hello, Alice!
# Hello, Alice!
```

In this snippet, the repeat decorator repeats the execution of greet function **n** times.

2.11.4 Chaining Decorators

Multiple decorators can be applied to a single function, creating a chain of decorators.

```python
def uppercase(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def exclaim(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result + "!"
    return wrapper

@exclaim
@uppercase
def greet(name):
    return f"Hello, {name}"

print(greet("Alice"))
# Output: HELLO, ALICE!
```

In the above use case, **greet** is first decorated with **uppercase**, which converts the result to uppercase, and then with **exclaim**, which adds an exclamation mark.

2.11.5 Summary

Decorators in Python provide a versatile and powerful way to extend the behavior of functions and methods. By applying decorators, you can write cleaner, more maintainable code that adheres to principles such as separation of concerns and DRY (Don't Repeat Yourself). Whether used for logging, access control, caching, or other functionalities, decorators are an essential tool in a Python programmer's toolkit.

2.12 Advanced Functional Programming Libraries

While Python's standard library provides a solid foundation for functional programming, several advanced libraries can enhance and extend these capabilities. These libraries introduce concepts such as monads, functors, and higher-order functions, enabling more expressive and robust functional programming. Notable among these libraries are **PyMonads, Toolz, Pyrsistent,** and **Funcy**.

2.12.1 PyMonads

PyMonads is a library that brings the power of monads, a fundamental concept in functional programming, to Python. Monads provide a way to handle computations and manage side effects in a functional manner. **PyMonads** includes implementations of several monads such as Maybe, Either, and List.

1. Maybe Monad: Represents an optional value that can be either **Just** a value or **Nothing**.

```
from pymonad.Maybe import Maybe, Just, Nothing

def safe_divide(a, b):
    return Just(a / b) if b != 0 else Nothing

result = safe_divide(10, 2).map(lambda x: x * 2)
print(result)  # Output: Just(10.0)

result = safe_divide(10, 0).map(lambda x: x * 2)
print(result)  # Output: Nothing
```

2. Either Monad: Represents a computation that can result in either a Right (success) or Left (failure) value.

```
from pymonad.Either import Left, Right

def parse_int(s):
    try:
        return Right(int(s))
    except ValueError:
        return Left("Not a number")

result = parse_int("123").map(lambda x: x * 2)
print(result)  # Output: Right(246)

result = parse_int("abc").map(lambda x: x * 2)
print(result)  # Output: Left(Not a number)
```

2.12.2 Toolz

Toolz is a functional programming toolkit for Python that provides a set of utility functions for iterators, functions, and dictionaries. It is designed to be lightweight and fast, enabling efficient functional transformations.

1. Currying: Converting a function with multiple arguments into a series of functions that each take a single argument.

```python
from toolz import curry

@curry
def add(x, y):
    return x + y

add_five = add(5)
print(add_five(10))  # Output: 15
```

2. Composition: Creating a new function by combining multiple functions.

```python
from toolz import compose

def double(x):
    return x * 2

def increment(x):
    return x + 1

double_then_increment = compose(increment, double)
print(double_then_increment(3))  # Output: 7
```

2.12.3 Pyrsistent

Pyrsistent provides persistent (immutable) data structures that facilitate functional programming by preventing side effects and enabling safer concurrent programming.

1. Persistent List: An immutable list that returns a new list on modifications.

```python
from pyrsistent import pvector

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

2. Persistent Dictionary: An immutable dictionary that returns a new dictionary on modifications.

```python
from pyrsistent import pmap

d = pmap({'a': 1, 'b': 2})
new_d = d.set('c', 3)
print(d)        # Output: pmap({'a': 1, 'b': 2})
print(new_d)    # Output: pmap({'a': 1, 'b': 2, 'c': 3})
```

2.12.4 Funcy

Funcy is a collection of functional programming utilities for Python. It includes tools for working with sequences, functions, and dictionaries in a functional style.

1. Lazy Sequences: Delaying computation until the result is actually needed.

```python
from funcy import lmap, lfilter

squares = lmap(lambda x: x * x, range(10))
even_squares = lfilter(lambda x: x % 2 == 0, squares)
print(list(even_squares))  # Output: [0, 4, 16, 36, 64]
```

2. Partial Functions: Partially applying a function to fix a number of its arguments.

```python
from funcy import partial

def multiply(x, y):
    return x * y
```

```python
double = partial(multiply, 2)
print(double(5))  # Output: 10
```

2.12.5 Summary

Python developers, with the help of these advanced libraries, can write more modular, readable, and maintainable code that adheres to functional programming principles. These libraries extend Python's capabilities, making it easier to implement complex functional patterns and manage side effects effectively.

2.13 Error Handling in Functional Programming

Error handling in Python is a crucial aspect of writing robust and resilient applications. In functional programming, the approach to error handling often emphasizes immutability, explicit error propagation, and the use of functional constructs to manage exceptions gracefully. Here's an overview of how to effectively handle errors in Python while adhering to functional programming principles.

2.13.1 Using Exceptions in Python

Python's built-in mechanism for error handling is the try-except block. This approach can be integrated into a functional programming style, although it introduces some imperative elements.

```python
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        return f"Error: {e}"

result = divide(10, 0)  # Error: division by zero
```

While **try—except** is straightforward, it can sometimes lead to less readable code when combined with functional constructs. To maintain a more functional style, alternative methods can be used.

2.13.2 Returning Error States

Instead of using exceptions, functions can return error states, often encapsulated in a specific data type such as tuples or custom classes. This approach makes error handling explicit and fits well with functional paradigms.

```
def safe_divide(a, b):
    if b == 0:
        return (False, "division by zero")
    else:
        return (True, a / b)

success, result = safe_divide(10, 0)
if success:
    print(f"Result: {result}")
else:
    print(f"Error: {result}")
```

This method ensures that the error state is part of the function's return value, making it clear when and where errors need to be handled.

2.13.3 Using Option and Either Types

Functional programming often employs monadic types like **Option** and **Either** to manage errors. These types can be implemented in Python using classes.

1. **Option**

The **Option** type represents an optional value that can either be **Some(value)** or **None**.

```
from typing import Union, Optional

class Option:
    def __init__(self, value: Optional[Union[int, str]] = None):
        self.value = value

    def is_some(self):
        return self.value is not None

    def unwrap(self):
        if self.is_some():
            return self.value
        else:
            raise ValueError("Called unwrap on a None value")

def safe_divide(a, b) -> Option:
    if b == 0:
        return Option()
    else:
        return Option(a / b)

result = safe_divide(10, 2)
if result.is_some():
    print(f"Result: {result.unwrap()}")
else:
    print("Error: division by zero")
```

2. **Either** Type

The **Either** type represents a value of one of two possible types (a disjoint union). Typically, **Either** is used to represent values that are either a correct result or an error.

```
from typing import TypeVar, Generic

L = TypeVar('L')
R = TypeVar('R')

class Either(Generic[L, R]):
    def __init__(self, left: L = None, right: R = None):
        self.left = left
        self.right = right

    def is_left(self):
        return self.left is not None

    def is_right(self):
        return self.right is not None

    def unwrap(self):
```

```python
        if self.is_right():
            return self.right
        elif self.is_left():
            return self.left
        else:
            raise ValueError("Neither left nor right value is set")

def safe_divide(a, b) -> Either[str, float]:
    if b == 0:
        return Either(left="division by zero")
    else:
        return Either(right=a / b)

result = safe_divide(10, 0)
if result.is_right():
    print(f"Result: {result.unwrap()}")
else:
    print(f"Error: {result.unwrap()}")
```

2.13.4 Composing Error Handling

Error handling can be composed using functional combinators like **map** and **flat_map** (also known as **bind** or >>= in other languages). These allow chaining operations that might fail, propagating errors through the chain.

```python
def safe_divide(a, b) -> Either[str, float]:
    if b == 0:
        return Either(left="division by zero")
    else:
        return Either(right=a / b)

def safe_sqrt(x) -> Either[str, float]:
    if x < 0:
        return Either(left="cannot take sqrt of negative number")
    else:
        return Either(right=x ** 0.5)
```

```
result = safe_divide(10, 2).flat_map(safe_sqrt)
if result.is_right():
    print(f"Result: {result.unwrap()}")
else:
    print(f"Error: {result.unwrap()}")
```

This approach allows for clean, readable, and maintainable code, where each operation is clearly defined and errors are propagated explicitly through the use of combinators.

2.14 Concurrency with Functional Programming

Concurrency is a programming paradigm that allows multiple processes to run simultaneously, potentially improving the performance of programs by making better use of available system resources. In the context of functional programming in Python, concurrency can be leveraged to create more efficient and responsive applications. This section explores the principles and practices of concurrency in Python functional programming, highlighting key concepts, tools, and techniques.

Functional programming emphasizes immutability, statelessness, and pure functions (functions that do not produce side effects and always yield the same output for the same input). These principles naturally align with concurrency, as they reduce the complexities and potential errors associated with shared state and mutable data. By adhering to functional programming principles, concurrent programs can be more predictable and easier to reason about.

Functional Programming with Concurrency is a functional programming techniques that can be combined with concurrency tools to create robust, efficient, and maintainable code:

- Immutable Data Structures: Using immutable data structures ensures that data is not modified across concurrent tasks, preventing race conditions.
- Pure Functions: Writing pure functions that do not rely on or alter shared state makes concurrent programs more predictable and easier to debug.
- Higher-Order Functions: Utilizing higher-order functions in conjunction with concurrency tools can lead to concise and expressive code.

```
from concurrent.futures import ThreadPoolExecutor
import asyncio

def square(n):
    return n * n
```

```
# Using ThreadPoolExecutor with map for parallel computation
numbers = [1, 2, 3, 4, 5]
with ThreadPoolExecutor() as executor:
    results = list(executor.map(square, numbers))

print("Squared numbers (ThreadPool):", results)

# Using asyncio with gather for asynchronous computation
async def compute_square_async(n):
    await asyncio.sleep(1)
    return n * n

async def main():
    tasks = [compute_square_async(i) for i in numbers]
    results = await asyncio.gather(*tasks)
    print("Squared numbers (asyncio):", results)

asyncio.run(main())
```

3 Some applications of Python functional programming

3.0.1 To-do List Application

This application is designed to help you efficiently manage your tasks and stay organized. With this app, you can easily create tasks with titles, descriptions, and completion statuses. Key features include:

- Task Creation: Define tasks with optional descriptions and track their completion status.
- Task Addition: Add new tasks to your list without altering existing ones, ensuring your original list remains intact.
- Task Filtering: Filter tasks based on their completion status to focus on what's pending or review what's been completed.
- Marking Tasks: Update tasks to mark them as completed, keeping your task list current and accurate.

The intuitive interface and robust functionality make it simple to manage your daily tasks, improve productivity, and stay on top of your to-do list.

About the source code, there are some notes that should be consider:

1. Task data structure:

- The **task** function defines a task with attributes: **title** , **description** (optional with a default empty string), and completed (default is **False**).
- Returns a dictionary representing the task.

2. Adding a new task:

- The **add_task** function adds a new task to a list of tasks and returns a new list, avoiding modifications to the original list by using list concatenation.

3. Filtering tasks:

- The **filter_tasks** function filters tasks based on their completion status (completed or pending), returning a list of tasks that match the specified status.

4. Marking a task as completed:

- The **mark_completed** function updates a specific task's completed status to True based on its title, returning a new list with the updated task.

5. Usage:

Demonstrates initializing an empty task list, adding tasks, printing all tasks, filtering and printing pending tasks, and marking a task as completed, then printing the updated task list.

- **List manipulation:** Functions like add_task and mark_completed use functional techniques to create new lists without modifying the originals.
- **Higher-order functions:** filter_tasks utilizes list comprehension to filter tasks based on a condition.
- **Immutability:** Creating new data structures is prioritized rather than modifying existing ones for better data integrity.

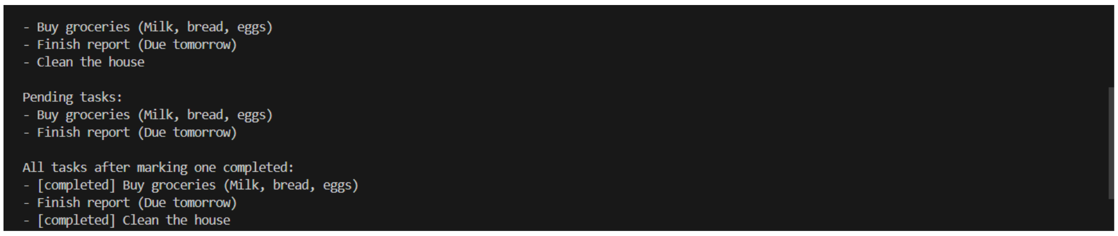
3.0.2 Directories Management

Let's explore a Python program that recursively lists all files and directories within a specified directory. The program leverages the power of recursion to traverse the file system, ensuring that all sub-directories are explored. This method is particularly useful for tasks such as generating a comprehensive list of files for indexing or backup purposes.

The program consists of a main function **list_directory** that takes a directory path as input and returns a list of all files and directories within that path. An inner function **explore** is defined to handle the recursive traversal:

- **Base Case:** If the current path is a file, it returns a list containing just that file.
- **Recursive Case:** If the path is a directory, it lists the contents and recursively explores each entry.
- **Error Handling:** A **try—except** block catches **PermissionError** exceptions to handle directories that cannot be accessed.

3.0.3 Result

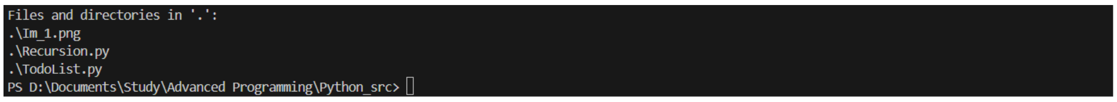


```
- Buy groceries (Milk, bread, eggs)
- Finish report (Due tomorrow)
- Clean the house

Pending tasks:
- Buy groceries (Milk, bread, eggs)
- Finish report (Due tomorrow)

All tasks after marking one completed:
- [completed] Buy groceries (Milk, bread, eggs)
- Finish report (Due tomorrow)
- [completed] Clean the house
```

Figure 1: To-do List



```
Files and directories in '.':
.\Im_1.png
.\Recursion.py
.\TodoList.py
PS D:\Documents\Study\Advanced Programming\Python_src> 
```

Figure 2: Recursion Directory

4 Conclusion

Functional programming in Python offers a powerful paradigm that emphasizes the use of pure functions, immutability, and higher-order functions to create more predictable and maintainable code. By conducting Python's functional programming features such as lambda functions, map, filter, and reduce, developers can write cleaner and more concise code. While Python is not a purely functional language, it provides sufficient tools and libraries that allow programmers to adopt and benefit from functional programming principles. Integrating these principles into Python development can lead to improved code quality, easier debugging, and enhanced scalability of software projects. As Python continues to evolve, embracing functional programming can be a valuable skill for any developer looking to enhance their programming toolkit.

References

[1] David Mertz, *Functional Programming in Python*.

[2] Steven F.Lott, *Functional Python Programming- Second Edition*.

[3] Geeksforgeeks, *Decorators in Python*. Accessed via link: https://www.geeksforgeeks.org/decorators-in-python/.

[4] Python Enhancement Proposals, *PEP 634 – Structural Pattern Matching: Specification?*. Accessed via link: https://peps.python.org/pep-0634/.

[5] Mike Schaid, *Cleaning up python code with the toolz library*. Accessed via link: https://medium.com/@mschaid86/cleaning-up-python-code-with-the-toolz-library-40803f0c8303.

[6] ArjanCodes, *Python Functors and Monads: A Practical Guide*. Accessed via link: https://arjancodes.com/blog/python-functors-and-monads/.