

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



ADVANCED PROGRAMMING

Report

Functional Programming in Python

Lecturer(s): Dr. Truong Tuan Anh
Student: Dai Ngoc Quoc Trung 2053537

HO CHI MINH CITY, MAY 2024

Contents

1	Programming Language Python	1
1.1	Introduction	1
1.2	Installation Setup	3
1.3	Execution Modes	4
1.4	Python Fundamentals	4
1.4.1	Tokens	4
1.4.1.1	Keywords	4
1.4.1.2	Identifiers	4
1.4.1.3	Literals	5
1.4.1.4	Delimiters	5
1.4.2	Operators	5
1.4.2.1	Arithmetic Operators	5
1.4.2.2	Comparison Operators	6
1.4.2.3	Logical Operators	6
1.4.2.4	Bitwise Operators	6
1.4.2.5	Assignment Operators	6
1.4.2.6	Membership Operators	7
1.4.2.7	Identity Operators	7
1.4.2.8	Operator Precedence	7
1.4.3	Data Types	8
1.4.4	Objects and Variables	9
1.4.5	Input/Output	9
1.4.6	Control Flow	9
1.4.6.1	Conditional Statements	9
1.4.6.2	Loops	10
1.4.6.3	Control Flow Mechanisms	10
1.4.6.4	Nested Loops and Conditional Statements	11
1.5	Functional Programming	11
1.5.0.1	Concepts of functional programming	11

2	How to create a functional program in Python	13
2.1	First Class functions	13
2.1.1	Functions are objects	13
2.1.2	Functions can be passed as arguments	13
2.1.3	Functions can be returned as a value	14
2.2	Pure functions	14
2.3	Evaluation	14
2.4	Data structures comprehension	15
2.4.1	List	15
2.4.2	Set	15
2.4.3	Dictionary	16
2.5	Zip function	16
2.6	Curried function	16
2.7	Recursive functions	17
2.8	Function composition	18
2.9	Anonymous function	18
2.10	Higher-order function	18
2.10.1	Apply-to-all	18
2.10.2	Filter	19
2.10.3	Reduce	19
2.10.4	Closure function	19
2.11	Decorator functions	20

Chapter 1

Programming Language Python

1.1 Introduction

Python is a high-level, interpreted programming language renowned for its simplicity and readability, making it an excellent choice for beginners and experienced developers alike. Python is dynamically-typed which means that the type of a variable is determined at runtime rather than at compile-time. Python also supports garbage collection and helps prevent memory leaks by freeing up memory that is no longer needed, thus allowing the program to use it for new objects[1].



Figure 1.1: Python Logo

One of Python's key strengths is its versatility. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python's comprehensive standard library, often referred to as "batteries included," provides modules and packages for a wide range of tasks, from web development and data analysis to machine learning and scientific computing[1].

Guido van Rossum began working on Python in the late 1980s as a successor to the ABC language, intending to address some of ABC's shortcomings while maintaining its strengths, and first released it in 1991 as Python 0.9.0. Over the years, Python has evolved through several major versions, with Python 2.0 introduced in 2000, bringing significant new features like list comprehensions and garbage collection. Python 3.0, released in

2008, aimed to rectify fundamental design flaws, making it the current standard despite breaking backward compatibility with Python 2. Python 2.7.18, released in 2020, was the last release of Python 2 [2]. The transition from Python 2 to Python 3 was significant due to the lack of backward compatibility. This required many developers to update their codebases, but the enhancements and new features in Python 3 justified the change. Python 3 has been designed to improve language consistency, performance, and support for modern computing needs.



Figure 1.2: Guido van Rossum - the father of Python

Advantages:

The key advantages of Python are as follows:

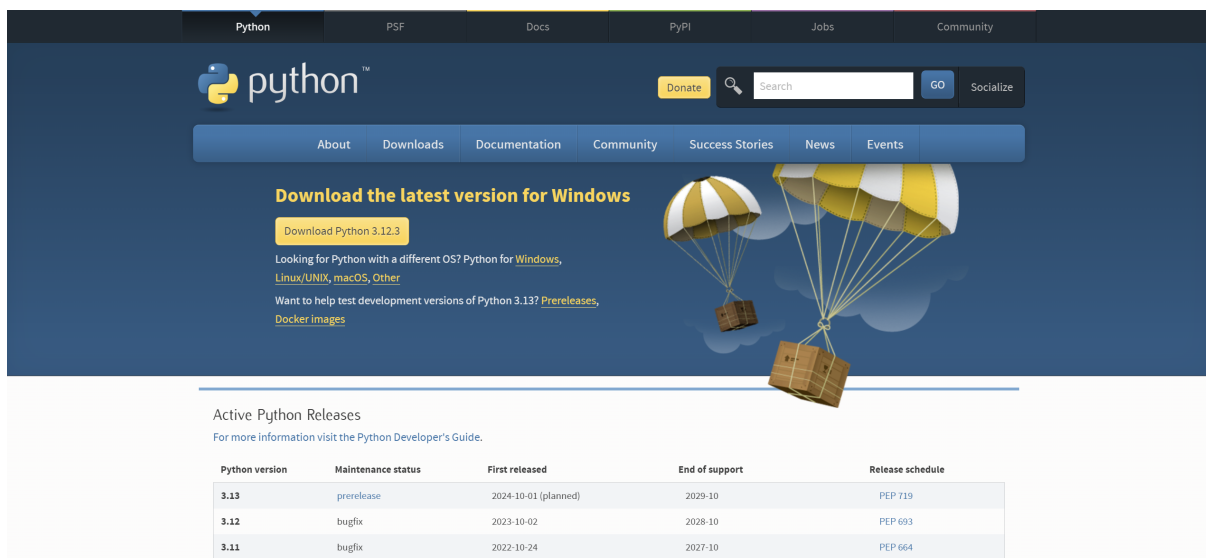
- **Ease of Learning and Use:** Python's simple and readable syntax resembles natural language, making it accessible to beginners.
- **Extensive Libraries and Frameworks:** Python boasts a vast standard library and numerous third-party packages for tasks ranging from web development (Django, Flask) to data analysis (Pandas, NumPy) and machine learning (TensorFlow, Scikit-learn).
- **Community Support:** Python has a large, active community that contributes to its development, provides support, and creates a wealth of learning resources.
- **Portability:** Python code can run on various platforms without modification, including Windows, macOS, and Linux.
- **Integration Capabilities:** Python integrates well with other languages and tools, allowing for smooth interoperability and use in multi-language projects.
- **Rapid Prototyping:** Python's simplicity and the availability of numerous libraries enable quick development and testing of ideas.

Disadvantages:

- **Performance:** As an interpreted language, Python is generally slower than compiled languages like C or Java.
- **Global Interpreter Lock (GIL):** The GIL in CPython can be a performance bottleneck for multi-threaded applications.
- **Memory Consumption:** Python's dynamic typing and high-level data structures can result in higher memory usage.
- **Weak in Mobile Computing:** Python is not typically used for mobile app development due to performance and integration issues compared to native mobile languages like Swift (iOS) or Java (Android).
- **Runtime Errors:** Python's dynamic typing can lead to runtime errors that might be harder to detect compared to compile-time errors in statically typed languages.
- **Lesser Support for Low-Level Programming:** Python is not well-suited for low-level programming tasks, such as operating system development or embedded programming.

1.2 Installation Setup

To download the latest version of Python, visit [Python Installation](#).



The screenshot shows the Python.org website. The main navigation bar includes links for Python, PSF, Docs, PyPI, Jobs, and Community. Below the navigation bar, there's a search bar and a 'Donate' button. The main content area features a large banner with the text 'Download the latest version for Windows' and a button to 'Download Python 3.12.3'. Below this, there are links for 'Linux/UNIX, macOS, Other' and 'Prereleases, Docker images'. To the right of the banner is an illustration of two parachutes with boxes hanging from them. Below the banner, there's a section titled 'Active Python Releases' with a link to the 'Python Developer's Guide'. This section contains a table with the following data:

Python version	Maintenance status	First released	End of support	Release schedule
3.13	prerelease	2024-10-01 (planned)	2029-10	PEP 719
3.12	bugfix	2023-10-02	2028-10	PEP 693
3.11	bugfix	2022-10-24	2027-10	PEP 664

Figure 1.3: Python installation guide

1.3 Execution Modes

After installing the latest version of the Python interpreter, we can now write and execute some basic Python codes. There are two ways to execute a Python program:

1. **Interactive Mode:** Python commands can be directly put into the interpreter prompt (usually denoted by `>>>`). To start interactive mode, simply put `'python'` (or `'python3'`) in the terminal without specifying a script file.
2. **Script Mode:** Involves writing Python code in a script file (usually with a `.py` extension) and executing it using the Python interpreter.
3. **Jupyter Notebook:** An interactive computing environment where to create and share documents containing live code, equations, visualizations, and explanatory text. Notebooks are organized into cells, which can contain code (Python or other languages), Markdown text, or raw content.

1.4 Python Fundamentals

1.4.1 Tokens

Python tokens are the smallest individual units in a program. They include keywords, identifiers, literals, delimiters, and operators.

1.4.1.1 Keywords

Keywords are reserved words that have special meanings in Python. Examples include `if`, `else`, `while`, `for`, `def`, and `class`.

```
# Example of using keywords
if x > 0:
    print("Positive number")
else:
    print("Non-positive number")
```

1.4.1.2 Identifiers

Identifiers are names used to identify variables, functions, classes, modules, and other objects. They must start with a letter (A-Z or a-z) or an underscore (`_`), followed by letters, digits, or underscores.

```
# Examples of identifiers
variable_name = 10
function_name = lambda x: x * 2
```

```
class MyClass:  
    pass
```

1.4.1.3 Literals

Literals are fixed values assigned to variables. Python supports several types of literals including string, integer, float, and boolean.

```
# Examples of literals  
string_literal = "Hello, World!"  
integer_literal = 42  
float_literal = 3.14  
boolean_literal = True
```

1.4.1.4 Delimiters

Delimiters are symbols that have special meaning in Python, such as parentheses "()", brackets "[]", braces "{}", comma ",", colon ":", and others.

```
# Examples of delimiters  
list_example = [1, 2, 3, 4]  
tuple_example = (1, 2, 3, 4)  
dict_example = {'key1': 'value1', 'key2': 'value2'}
```

1.4.2 Operators

Operators are symbols that perform operations on variables and values. Python supports several types of operators:

1.4.2.1 Arithmetic Operators

Arithmetic operators perform mathematical operations such as addition, subtraction, multiplication, and division.

```
# Arithmetic Operators  
x = 10  
y = 5  
  
addition = x + y          # Addition: 15  
subtraction = x - y       # Subtraction: 5  
multiplication = x * y    # Multiplication: 50  
division = x / y          # Division: 2.0  
floor_division = x // y   # Floor Division: 2  
modulus = x % y           # Modulus: 0  
exponentiation = x ** y   # Exponentiation: 100000
```


1.4.2.2 Comparison Operators

Comparison operators compare two values and return a Boolean result (**True** or **False**).

```
# Comparison Operators
x = 10
y = 5

equal = (x == y)      # Equal to: False
not_equal = (x != y)   # Not equal to: True
greater_than = (x > y) # Greater than: True
less_than = (x < y)    # Less than: False
greater_equal = (x >= y) # Greater than or equal to: True
less_equal = (x <= y)  # Less than or equal to: False
```

1.4.2.3 Logical Operators

Logical operators combine conditional statements and return a Boolean result.

```
# Logical Operators
a = True
b = False

logical_and = a and b # Logical AND: False
logical_or = a or b    # Logical OR: True
logical_not = not a    # Logical NOT: False
```

1.4.2.4 Bitwise Operators

Bitwise operators perform operations on binary representations of integers.

```
# Bitwise Operators
x = 10 # Binary: 1010
y = 4  # Binary: 0100

bitwise_and = x & y    # AND: 0000 -> 0
bitwise_or = x | y     # OR: 1110 -> 14
bitwise_xor = x ^ y    # XOR: 1110 -> 14
bitwise_not = ~x       # NOT: -(x+1) -> -11
left_shift = x << 2    # Left shift: 101000 -> 40
right_shift = x >> 2   # Right shift: 0010 -> 2
```

1.4.2.5 Assignment Operators

Assignment operators are used to assign values to variables. They include the basic assignment operator ("=") and compound operators that combine assignment with another operation.

```
# Assignment Operators
x = 10
x += 5 # Equivalent to x = x + 5 -> 15
x -= 5 # Equivalent to x = x - 5 -> 10
x *= 5 # Equivalent to x = x * 5 -> 50
x /= 5 # Equivalent to x = x / 5 -> 10.0
x //= 5 # Equivalent to x = x // 5 -> 2
x %= 5 # Equivalent to x = x % 5 -> 0
x **= 5 # Equivalent to x = x ** 5 -> 0
```

1.4.2.6 Membership Operators

Membership operators test for membership in a sequence, such as strings, lists, or tuples.

```
# Membership Operators
x = [1, 2, 3, 4, 5]

in_operator = 3 in x # True
not_in_operator = 6 not in x # True
```

1.4.2.7 Identity Operators

Identity operators compare the memory locations of two objects.

```
# Identity Operators
a = 10
b = 10
c = a

# a, b, c now point to the same memory location
is_operator = (a is b) # True
is_not_operator = (a is not c) # False
```

1.4.2.8 Operator Precedence

Operator precedence determines the order in which operators are evaluated in expressions. Operators with higher precedence are evaluated before operators with lower precedence. The following table shows the order of precedence for Python operators from highest to lowest:

Operator	Description
()	Parentheses
**	Exponentiation
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor Division, Modulus
+, -	Addition, Subtraction
«, »	Bitwise Shift Operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparison Operators
not	Logical NOT
and	Logical AND
or	Logical OR

Table 1.4.1: Python Operator Precedence

Examples of Operator Precedence

```

# Example 1: Exponentiation before Multiplication
result = 2 ** 3 * 4
# Equivalent to (2 ** 3) * 4
# result = 32

# Example 2: Multiplication before Addition
result = 2 + 3 * 4
# Equivalent to 2 + (3 * 4)
# result = 14

# Example 3: Parentheses alter precedence
result = (2 + 3) * 4
# result = 20

# Example 4: Logical operators with precedence
result = True or False and False
# Equivalent to True or (False and False)
# result = True

result = (True or False) and False
# result = False

```

1.4.3 Data Types

Python has various built-in data types, including numbers, strings, lists, tuples, sets, and dictionaries.

```
# Examples of data types
integer_number = 10      # Integer
floating_point_number = 10.5 # Float
string_text = "Hello"    # String
list_example = [1, 2, 3] # List
tuple_example = (1, 2, 3) # Tuple
set_example = {1, 2, 3}  # Set
dict_example = {'one': 1, 'two': 2} # Dictionary
```

1.4.4 Objects and Variables

In Python, everything is an object, and variables are references to these objects.

```
# Example of objects and variables
x = 10 # x is a variable referencing an integer object with value 10
y = x  # y now references the same object as x
x = 20 # x references a new object with value 20
```

1.4.5 Input/Output

Python provides built-in functions for reading input from the user and displaying output.

```
# Example of input and output
user_input = input("Enter your name: ") # Reading input
print(f"Hello, {user_input}!")          # Printing output
```

1.4.6 Control Flow

Control flow in Python allows to dictate the order in which statements are executed in a program. Python provides several control flow tools:

1.4.6.1 Conditional Statements

Conditional statements allow to execution of different blocks of code based on certain conditions.

```
# Example of if, elif, and else statements
x = 10

if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

1.4.6.2 Loops

Loops allow to execution of a block of code multiple times. Python provides two types of loops: **for** loops and **while** loops.

For Loop

A **for** loop iterates over a sequence (such as a list, tuple, or string).

```
# Example of a for loop
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

While Loop

A **while** loop repeats as long as a condition is true.

```
# Example of a while loop
count = 0
while count < 5:
    print(count)
    count += 1
```

1.4.6.3 Control Flow Mechanisms

Python provides several statements to control the flow of loops:

Break Statement

The **break** statement exits the loop immediately.

```
# Example of break statement
for i in range(10):
    if i == 5:
        break
    print(i)
```

Continue Statement

The **continue** statement skips the rest of the code inside the loop for the current iteration and jumps to the next iteration.

```
# Example of continue statement
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Pass Statement

The **pass** statement does nothing and is used as a placeholder in situations where a statement is syntactically required but doesn't want to execute any code.

```
# Example of pass statement
for i in range(10):
    if i % 2 == 0:
        pass
    else:
        print(i)
```

1.4.6.4 Nested Loops and Conditional Statements

Nest loops and conditional statements within each other to create complex control flows.

```
# Example of nested loops and conditional statements
for i in range(3):
    for j in range(3):
        if i == j:
            print(f"i and j are both {i}")
        else:
            print(f"i is {i}, j is {j}")
```

1.5 Functional Programming

Functional programming is a programming paradigm that emphasizes binding everything in a pure mathematical functions style. It is a declarative type of programming, focusing on "what to solve" rather than "how to solve." Functional programming uses expressions instead of statements[3]. While Python is not purely a functional programming language, it does offer many functional programming features that provide benefits similar to those of functional programming.

1.5.0.1 Concepts of functional programming

A functional programming language is expected to adhere to the following concepts:

- **Pure Functions:** These functions always produce the same output for the same arguments, regardless of any other factors. Additionally, they do not have any side effects, meaning they do not modify any arguments, or global variables, or produce any output.
- **Recursion:** Functional languages do not use "for" or "while" loops for iteration. Instead, iteration is implemented through recursion.

- **Functions are First-Class and can be Higher-Order:** First-class functions are treated as first-class variables. They can be passed to functions as parameters, returned from functions, or stored in data structures.
- **Variables are Immutable:** In functional programming, variables cannot be modified after they have been initialized. New variables can be created, but existing variables cannot be modified.

Chapter 2

How to create a functional program in Python

2.1 First Class functions

In Python, functions are first-class objects and can be manipulated like all other objects.

Properties of first class functions:

2.1.1 Functions are objects

This example below demonstrate an assignment of a predefined function to a variable. The variable does not call the function, instead it takes the function referenced and create a second pointing name for it.

```
def hello():  
    return "Hello World"  
  
print(hello())           # Output: Hello World  
  
world = hello            # As a reference for hello  
  
print(world())           # Output: Hello World
```

2.1.2 Functions can be passed as arguments

Functions that can be passed as arguments allow flexibility and reusable code. This practice is common in functional programming style, such as callbacks, event handlers and higher-order functions.


```
def foo(f, x):  
    return f(x)  
  
def pow2(x):  
    return x**2  
  
print(foo(pow2, 4))
```

In this example, function **'foo'** is a higher-order function that takes 2 parameters - a function and a variable. The function **'pow2'** is then passed to function **foo** and produce the output.

2.1.3 Functions can be returned as a value

```
def f(x):  
    def g(y):  
        return x * y  
    return g  
  
expFuncVal = f(3)  
print(expFuncVal(4))
```

2.2 Pure functions

A function can be called a pure function if it always return the same result for same argument values and has no side effects such as manipulating, printing, etc. The only result that its returning value.

```
def add1(x, y):  
    return x + y  
  
def add2(x, y):  
    print(x+y)
```

In this example, both **'add1'** and **'add2'** functions take 2 arguments that demonstrate the addition operation. However only the function **'add1'** returning the result of the operation and thus considered as a pure function. Meanwhile **'add2'** do the computation and print it out which conflicts the definition of a pure function.

2.3 Evaluation

Python supports some evaluation strategies, typically strict and non-strict evaluation.

- **Strict:** Python operators are generally strict and evaluate all sub-expressions from left to right. This means that the evaluation happens immediately, and the result is stored for future use. Strict evaluation is also known as Eager evaluation.

```
def division(x, y):  
    return x / y  
print(division(8, 4))    # Output: OK  
print(division(3, 6))    # Output: OK  
print(division(1, 0))    # Output: Failure,  
                        # since 1 can not be divided by 0.  
  
print(len([1+0, 1*0, 1/0, 1-0]))  
# Output: Failure, since it must evaluate the expression "1/0"  
# before it could return the value of higher-order function.
```

- **Non-strict:** In Python, the logical expression operators **and**, **or**, and **ifelse** are all non strict evaluation. They are also known as **short-circuit** operators because they don't need to evaluate all arguments to determine the value.

```
>>> 0 and print("string")  
0  
>>> True or print("right")  
True
```

2.4 Data structures comprehension

2.4.1 List

In Python, we can define a list comprehension by using this illustrations:

$$[< \text{expression} > \text{ for } < \text{item} > \text{ in } < \text{iterable} > (\text{if } < \text{condition} >)]$$

```
listComprehension = [x for x in range(5)]  
print(listComprehension)
```

2.4.2 Set

We can do the same of list with set.

$$\{< \text{expression} > \text{ for } < \text{item} > \text{ in } < \text{iterable} > (\text{if } < \text{condition} >)\}$$

```
setComprehension = {x for x in range(5)}  
print(setComprehension)
```

2.4.3 Dictionary

{< key_expression >:< value_expression > for < item > in < iterable > (if < condition >)}

```
dictComprehension = {x: x**2 for x in range(10)}  
print(dictComprehension)
```

2.5 Zip function

Zip function takes multiple iterables as arguments and returns an iterator of tuples where i-th tuple contains the i-th element from each of the input iterables. If the input iterables are of different lengths, the resulting iterator will have the length of the shorter iterable.

```
list1 = [1, 2, 3]  
list2 = [4, 5, 6]  
zipList = zip(list1, list2)  
print(list(zipList))
```

2.6 Curried function

Currying function is a technique of functional programming where a function with multiple arguments is transformed into a sequence of functions, each taking an argument. This allows for partial application of functions, where some arguments are fixed, producing a new function that takes the remaining arguments.

```
def curry_add(x):  
    def curried(y)  
        return x + y  
    return curried  
  
print(curry_add(1)(2)) # Output: 3
```

In this example, 'curry_add' is a function that takes an argument **x** and returns a new function 'curried' that takes another argument **y** and returns the sum of **x** and **y**.

We can also use the 'functools' module which provides a partial function that can be used to fix some arguments of a function and create a new function.

```
from functools import partial

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

add_5 = partial(add, 5)
print(add_5(10)) # Output: 15
```

Why curried function is useful in functional programming style?

- It helps to create a higher-order function. [4]
- It reduces the chance of errors in our function by dividing it into multiple smaller functions that can handle only one responsibility. [4]
- It promotes reusability and modularity. [4]
- It makes code more readable.

2.7 Recursive functions

In functional programming, recursion is often preferred over iterative constructs like loops for processing sequences and data structures. This is because functional programming emphasizes immutability and the use of pure functions, both of which are naturally supported by recursion.

```
# Recursion
def factorial(n):
    if n == 0:
        return 1
    elif n > 0:
        return n * factorial(n - 1)

print(factorial(4)) # Output: 24

def add_recursion(a, b):
    if a == 0:
        return b
    else:
        return add_recursion(a - 1, b + 1)

print(add_recursion(4, 5)) # Output: 9
```

2.8 Function composition

Function composition is a function that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second function.

```
def composition(f, g):  
    return lambda x: f(g(x))  
  
def f(x):  
    return x + 2  
  
def g(x):  
    return x + 3  
  
h = composition(f, g)  
print(h(4))
```

2.9 Anonymous function

An anonymous function is often called as a Lambda function. This function is nameless and similar with pure functions.

```
list1 = [1,2,3,4]  
list2 = [5,6,7]  
  
x1 = map(lambda x: x + y, list1, list2)
```

2.10 Higher-order function

Higher-order function is a function that contains other functions as parameters or returns a function as an output.

2.10.1 Apply-to-all

A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters. For example,

$$h(x) = x * x \longrightarrow \alpha h : (1, 2, 3) \text{ yields } (1, 4, 9)$$

In Python, it refers to 'map' function.

```
list1 = [1, 2, 3, 4, 5]  
list2 = [5, 6, 7, 8, 9]
```

```
def mult(x, y):  
    return x * y  
  
print(list(map(mult, list1, list2))) # Output: [5, 12, 21, 32, 45]
```

2.10.2 Filter

The **'filter'** function in Python is used to filter elements from an iterable (such as a list) based on a specified condition. It returns an iterator containing the elements for which the condition evaluates to True.

```
listFilter = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
def is_even(x):  
    return x % 2 == 0  
  
result = list(filter(is_even, listFilter))  
print(result) # Output: [2, 4, 6, 8, 10]
```

2.10.3 Reduce

The **'reduce'** function in Python is used to apply a rolling computation to sequential pairs of elements in an iterable. Its functionality is the same as **'foldl'** function in Haskell programming language.

```
listA = [1, 2, 3, 4]  
  
def fold_left(func, initial, lst):  
    return reduce(func, lst, initial)  
  
result = fold_left(lambda x, y: x + y, 0, listA)
```

The function **'fold_left'** generally takes 3 argument: a function, an initial number, and an iterator which is a list in this case. Then it applies the function on sequential pairs of elements in the list and we obtain the result.

2.10.4 Closure function

A closure in Python refers to a function that retains access to variables from the outer (enclosing) scope even after the outer function has finished executing [5]

```
def make_multiplier(factor):  
    def multiplier(x):  
        return x * factor  
    return multiplier
```

```
# Create a closure function
double = make_multiplier(2)

# Call the closure function
print(double(5))  # Output: 10
```

2.11 Decorator functions

A decorator is a special type of function that is used to modify or extend the behavior of another function or method. Decorators allow to add functionality to existing code without modifying the code itself, promoting code reusability, modularity, and maintainability.

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        print(f"Arguments: {args} {kwargs}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} returned {result}")
        return result

    return wrapper

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

@log_function_call
def multiply(a, b):
    return a * b

result1 = add(3, 5)
result2 = multiply(4, 7)
```

In this example, the 'log_function_call' function helps to display some information that function 'add' and 'multiply' would take, while maintaining the purely functionality that these two functions have. And the output is:

```
Calling function: add
Arguments: (3, 5) {}
Function add returned 8

Calling function: multiply
Arguments: (4, 7) {}
Function multiply returned 28
```

References

- [1] *Python (programming language)*. [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))
- [2] Peterson, Benjamin (20 April 2020). *Python 2.7.18, the last release of Python 2*. <https://pythoninsider.blogspot.com/2020/04/python-2718-last-release-of-python-2.html>
- [3] *Functional Programming in Python*. <https://www.geeksforgeeks.org/functional-programming-in-python/>
- [4] *Curried functions*. <https://www.geeksforgeeks.org/what-is-currying-function-in-javascript/>
- [5] *Closure functions*. <https://www.linkedin.com/pulse/understanding-closures-python-guide-real-world-examples-h-s-karthik/>