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

ADVANCED PROGRAMMING (CO2039)

Assignment - Semester 232

Functional Programming in Python

Instructor(s): Trương Tuấn Anh

Student: Trần Quốc Bảo Long - 2252453

Class: CC04

HO CHI MINH CITY, MAY 2024

Contents

1 The Python Programming Language

1.1 Introduction

Python is a high-level, general-purpose programming language, first released in 1991. Its design philosophy emphasizes code readability with the use of significant indentation.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Python is known for its simple, easy to learn syntax, which emphasizes readability and therefore reduces the cost of program maintenance.

1.2 Python Basics

1.2.1 First Program

First, let's start by installing Python into your computer. Open a text editor, then write this simple Python code:

```
print("Hello World!")
```

Save it as `hello_world.py`. Open a terminal, navigate to where you saved the file, then run the program by entering

```
python hello_world.py
```

The output should read:

```
Hello, World!
```

1.2.2 Syntax

In Python, indentations, or the spaces before the beginning of a code line, are used to indicate blocks of code, and have to be used for the program to work. For example:

```
if n < 4:
    print("The number is less than 4.")
```

1.2.3 Comments

In any programming language, comments are used to explain the code, making the code more readable and can also prevent commands from running.
In Python, comments are written by using a number sign `#` at the beginning, and Python will ignore them.

```
#This is a commnet
print("Hello, World!")

a = 10
b = 6
print(a + b) #Add two numbers
```

Multi-line comments can be made by using `#` before each line or three quotation marks at the beginning and end.

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")

"""
This is a comment
written in
more than just one line
"""
print("My name is Long")
```

1.2.4 Variables and Data Types

In Python, variables are used to store data. Python has no command for declaring a variable - it is created the moment a value is assigned to it. Python supports various data types, including integers, floats, strings, lists, tuples, dictionaries, and so on.

Numbers

- **Integer** is a whole number, positive or negative, without decimal points, of unlimited length.

```
x = 1
y = -4096
z = 333

```

- **Float**, or "floating point number", is a number, positive or negative, containing one or more decimals, or a number written in exponential notation.

```
x = 1.0
pi = 3.1415926535
z = -12.345

a = 1e9
b = -1.1111e4

```

- **Complex numbers** are written with a `j` as the imaginary part:

```
x = 3+4j
y = 5j

```

Other data types

- **String** is a sequence of characters, surrounded by either single or double quotation marks. Multi-line strings are surrounded by three quotation marks (single or double).

```
print("Hello")

a = 'Welcome to Python'
print(a)

b = """In computer programming,
a string is traditionally a sequence
of characters."""
print(b)

```

- **Boolean**: either True or False. You can evaluate any expression in Python, and get one of two answers: True or False.

  For example:

```
print(11 > 9)
print(11 == 9)

```

  The program prints out:

```
True
False
```

- **List** is used to store multiple items in a single variable, created by using square brackets. Tuples are mutable and can be changed after creation, and allow duplicate values.

```
numbers = [1, 2, 3, 4, 5]
myList = ["apple", "banana", "cherry"]

```

- **Tuple** is similar to a list, but surrounded by a round bracket. A tuple is a collection which is ordered and unchangeable. Unlike lists, tuples do not need brackets. Tuples can also be single-element, using a comma after the sole element.

```
myTuple = (2, 3, 5, 7)
coordinator = (10.0, 136.0)

another_tuple = 1, 2, 3, 4, 5

single_element = (1,)

```

- **Set** is also like lists and tuples but use curly brackets. Sets are unordered, unchangeable and unindexed. However, items can be added or removed.

```
thisset = {"A", "B", "C"}

```

- **Dictionary** is a collection of data values in key:value pairs. A dictionary is ordered, mutable but do not allow duplicates.

```
thisDict= {"A": 1, "B": 2, "C": 3}

```

1.2.5 Operators

In Python, operators are used to perform operations on variables and values. They are divided into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators

- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic

Arithmetic operators are used with numeric values to perform mathematical operations like addition, subtraction, multiplication and division.

```
x = 10
y = 10

a = x + y   #Addition
b = x - y   #Subtraction
c = x * y   #Multiplication
d = x / y   #Division
e = x % y   #Modulus
f = x ** y  #Exponentiation
g = x // y  #Floor division
```

Assignment

Assignment operators are used to assign values to variables.

```
x = 10
y = 10

x = 5
x += 3   #Same as x = x + 3
x -= 3   #Same as x = x - 3
x *= 3   #Same as x = x * 3
x /= 3   #Same as x = x / 3
x %= 3   #Same as x = x % 3
x **= 3  #Same as x = x ** 3
x //= 3  #Same as x = x // 3
```

Precedence

Operator precedence determines the order in which operations are performed in an expression. Operators with higher precedence are evaluated first.

Operator	Description
()	Parentheses
**	Exponentiation
+x, -x, ~x	Unary plus, unary minus, and bitwise NOT
*, /, //, %	Multiplication, division, floor division, and modulus
+, -	Addition and subtraction
«, »	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

Table 1. Precedence of operators

1.2.6 Control Flow

As stated above, Python supports the usual logical conditions from mathematics, like `==`, `!=`, `<` or `>`. These conditions can be used in several ways, most commonly in `if` statements and loops.

Conditional Expressions

Conditional statements, `if`, `elif` and `else`, allow you to execute different blocks of code based on certain conditions.

For example:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
elif b == a:
    print("a is equal to b")
else:
    print("a is greater than b")
```

Loops

In a programming language, a loop is used to reiterate a process multiple times. In Python, we have the `for` and `while` loops.

- **for** loop is used to iterating over a sequence (list, tuple, dictionary, etc.)

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for x in numbers:
    print(x)

```

- **while** loop is used to execute a set of statements given a condition is true.

```
i = 1
while i < 5:
    print(i)
    i += 1

```

1.3 Python Functions

Functional programming consists of, of course, functions. Before we move on to the main part of this assignment, let's have an overview of how functions work in Python.

A **function** is a block of statements that return the specific task. Instead of writing the same code again and again, we can use functions to put the commands together. It helps increase code readability and reusability.

There are two types of functions in Python:

- User-defined functions
- Built-in functions

User-defined functions are created by the user to perform a specific task. We can define a function in Python, using the `def` keyword. We can add any type of parameters to it as we require.

```
def welcome():
   print("Hello World!")

def myAge(age):
   print("I am " + age + " years old.")

myAge(20)
```

The `myAge` function is a user-defined function that takes one parameter `age` and prints the user's age.

Built-in functions are functions available within Python without the need of explicitly declaring them. For example, `print()`, `len()`, `max()`.

```
numbers = [5, 9, 3, 1, 7]
print(max(numbers))
print(len(numbers))
```

The output is:

```
9
5
```

Lambda expressions, or *anonymous functions*, are small, single-expression functions without a name. They are defined using the lambda keyword. They can have any number of parameters but can only have one expression.

```
def cube(x):
    return x*x*x

cube_v2 = lambda x : x*x*x

print(cube(7))
print(cube_v2(7))
```

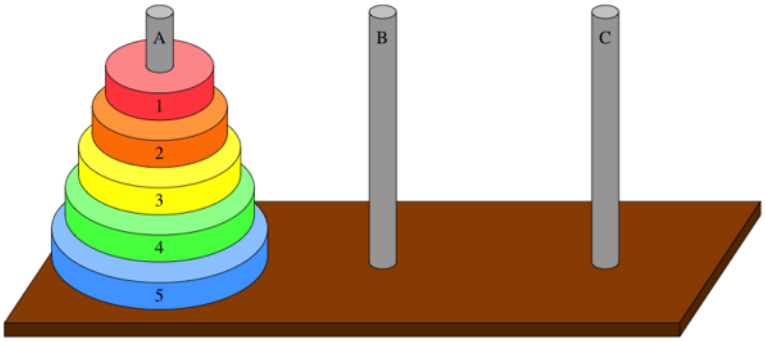
Recursive functions

A **recursive function** is a function that calls by itself. They are useful for solving problems that can be broken down into smaller problems.

For example, a function to calculate factorial of a number (product from 1 to n):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Another popular example of recursion in programming is the Tower of Hanoi problem, which tells each step to move the disks around the three rods.




```
def towerofHanoi(n, start, finish, aux):
    if n == 0:
        return
    towerofHanoi(n - 1, start, aux, finish)
    print("Move disk", n, "from rod", start, "to rod", finish)
    towerofHanoi(n - 1, aux, finish, start)

disks = 5
start = 'A'
aux = 'B'
finish = 'C'

towerofHanoi(disks, start, finish, aux)
```

2 Functional Programming in Python

And there you have it, some of the basic components in Python. Now, let's enter the world of functional programming.

Functional programming is a programming paradigm where one applies pure functions in sequence to solve complex problems. Functions take an input value and produce an output value without being affected by the program. Functional programming mainly focuses on what to solve and uses expressions instead of statements.

In Python, functional programming is supported without the support of any special features or libraries. It typically plays a fairly small role in Python code. Like other functional programming languages, it follows these basic concepts:

- **Pure Functions:** These functions have two main properties. First, they always produce the same output for the same arguments irrespective of anything else. Secondly, they have neither side-effects nor hidden outputs.
- **Recursion:** There are no "for" or "while" loop in functional languages – iterations are implemented through recursion.
- **Functions are First-Class and can be Higher-Order:** First-class functions are treated as first-class variable. The first-class variables can be passed to functions as a parameter, can be returned from functions or stored in data structures.
- **Immutability:** In functional programming, variables are immutable – we can't modify a variable after a value is assigned to it. We can create new variables, but we can't modify existing variables.

2.1 Pure Functions

In functional programming, a **pure function** always produces the same output for the same arguments input. It does not change the input variable or produce hidden outputs.

```
def pureFunction(list):
    newList = []
    for i in list:
        newList.append(i + 1)
    return newList

OriginalList = [1, 3, 5, 7]
ModifiedList = pureFunction(OriginalList)

print(OriginalList)
print(ModifiedList)
```

The output will be:

```
[1, 3, 5, 7]
[2, 4, 6, 8]
```

Above is an example of a pure function of adding 1 to every element of a list. It produces the same output for the same values.
Here is another example.

Sum of squares of two numbers:

```python
def sum_of_squares(x, y):
    return x**2 + y**2

result = sum_of_squares(3, 4)

print(result)
```

The result is:

```
25
```

2.2 Recursive Functions

Functional programming does not have `for` and `while` loops for iterations. Instead, they are done by using recursion. Recursive functions define by themselves.

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

result = fibonacci(10)
print(result)
```

The result should print out:

```
55
```

In this program, the `fibonacci` function calculates the n^{th} Fibonacci number:

- If n is 0, it returns 0.
- If n is 1, it returns 1.
- Otherwise, it recursively calls itself to calculate the sum of the two preceding numbers in the sequence.

2.3 Higher-Order Functions

In functional programming, a **higher-order function** is a function that takes a function as an argument or returns a function as a result.

Higher-order functions are a core concept in functional programming and offer several benefits that enhance the flexibility, readability, and maintainability of code. By passing functions as arguments, you can create generic operations that work with any specific behavior defined by the passed functions. They help in writing more reusable, modular, and maintainable code, facilitating a declarative and expressive coding style.

Here are some examples of higher-order functions.

```python
def apply_function(func, lst):
    return [func(x) for x in lst]

# Function to double the input
def double(x):
    return x * 2

# Function to square the input
def square(x):
    return x ** 2

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

doubled_numbers = apply_function(double, numbers)
squared_numbers = apply_function(square, numbers)

print(doubled_numbers)
print(squared_numbers)
```

The result would be:

```
[2, 4, 6, 8, 10]
[1, 4, 9, 16, 25]
```

In this example:

- `apply_function` is a higher-order function because it takes another function (`func`) as an argument.
- `double` and `square` are regular functions that are passed as arguments to `apply_function`.
- `apply_function` applies the passed function (`func`) to each element in the list `lst` and returns a new list with the results.

Higher-order functions enable function composition, using lambda expressions. You can build complex functionality by combining simpler functions.

```python
def compose(f, g):
    return lambda x: f(g(x))

def add_one(x):
    return x + 1

def multiply_by_two(x):
    return x * 2

composed_function = compose(add_one, multiply_by_two)
result = composed_function(5) #Result: 11
```

In this program, the `compose` function creates a new function by composing `add_one` and `multiply_by_two` .

2.4 First-Class Functions

In a functional programming language, **first-class functions** are treated like any other variable. They can be passed as arguments to other functions, or put them in other variables.

- They can be assigned to variables:

```
def onetoFive():
    return [1, 2, 3, 4, 5]

list = onetoFive()

def addtoList(arr, n):
    return arr.append(n)

n = 6
addtoList(list, n)
print(list)

```

The `onetoFive` function is assigned into the `list` variable.

- They can be passed as an argument:

```
def pi():
    return 3.14159

def areaCircle(radius):
    return radius * radius * pi()

r = 3

print(areaCircle(r))

```

In this case, the `pi` function acted as a float, representing pi.

- And, they can return functions from other functions.

```
def greet(name):
    return f"Hello, {name}!"

def farewell(name):
    return f"Goodbye, {name}!"

def call_function(func, name):
    return func(name)

print(call_function(greet, "Adam"))  # Output: Hello, Alice!
print(call_function(farewell, "Eva"))  # Output: Goodbye, Bob!

```

The `call_function` function takes a function (`func`) and a name, then calls the passed function with the given name.

2.5 How to Create a Functional Program

A step-by-step instruction to write a functional program in Python, with an example.

Step 1. Write the pure functions
Define the functions that always return the same output for the same input, with no side effects.

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

For example, `square(5)` always returns `25`.

Step 2. Write higher-order functions
Next, use higher-order functions that use the said pure functions as arguments.

```
def listSquare(numbers):
    squares = map(square, numbers)
    return list(squares)
```

The `map` function assigns the `square` function to all elements of the `numbers` list, squaring all of them.

Step 3. Use built-in functions
Then, use some of the functions already built-in and available to use.

In the code above, the built-in `map` function assigns a function to all variables of a list.

Step 4. Ensure immutability
Ensure that the functions do not modify any external state, or creating side effects. In this case, a tuple can be used instead of a list, if immutability is critical.

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

def listSquare(numbers):
    squares = map(square, numbers)
    return list(squares)

numbers = [1, 2, 3, 4, 5]
result = listSquare(numbers)
print(result)
#Output: [1, 4, 9, 16, 25]
```

3 A Demo Program: Nim

After learning the functional programming principles that Python supports, how about writing a simple program yourself? Here is an implementation of the Nim game in Python using functional programming.

Nim is a mathematical game of strategy, where two players take turns to remove objects from distinct lines (or heaps). On each turn, a player removes at least one object from a line, and may remove multiple or all objects as long as they belong to a line. The first player to take the last object remaining wins.

In this simple version of Nim implemented in Python, we represent the objects as stars lying on lines. Each player enters the number of row and number of stars they want to remove. There are five lines with the number of stars decreasing after each line - five down to one.

3.1 Step by Step

We start by declaring a function that returns a list, which contains the number of stars in each line. Then, a function to switch turns between players.

```
# Switch between each player's turn
def next_player(player):
    if player == 1:
        return 2
    elif player == 2:
        return 1

# Representing the number of stars in each line
def initial():
    return [5, 4, 3, 2, 1]
```

We need functions to display the game board in progress on the user's interface. First, the `put_row` function to display the number of stars in each line, using an F-string:

```
def put_row(row, num):
    print(f"{row}: {'* ' * num}")
```

Then, apply the higher-order function principle for the `put_board` function, to print the entire board:

```
def put_board(board):
    for i, num in enumerate(board, start=1):
        put_row(i, num)
```

The built-in `enumerate` function is used to loop over the board list while also keeping track of the current index. It returns pairs of index and value for each element in the list. `start=1` is used to count the indices from 1, since the board's rows start with 1.

Next, a `finished` function that loops through the board to check if it's is empty (no more stars remaining):

```
def finished(board):
    return all(stars == 0 for stars in board)
```

Now, we need a function to update the board after each player's move.

```
def move(board, row, num):
    new_board = board[:]
    new_board[row - 1] -= num
    return new_board
```

The `valid` function checks if a user's move is valid, based on the number of stars in a line.

```
def valid(board, row, num):
    if row == 0 or row > 5:
        return False
    elif board[row - 1] == 0 or board[row - 1] < num:
        return False
    else:
        return board[row - 1] >= num
```

The function checks the row and the number of stars the user input and compare them with the number of stars in such row.

- If the row is out of range (equals to 0 or greater than 5), it returns False.
- If the number of stars in a row is empty or smaller than the number of stars from input, it returns False.
- Otherwise, return True.

Additionally, `get_digit` function reads the values from player input.

```
def get_digit(prompt):
    while True:
        try:
            x = input(prompt)
            if x.isdigit():
                return int(x)
            else:
                print("ERROR: Invalid digit")
        except EOFError:
            sys.exit(0)
```

After implementing the helper functions, we put them together as a main logic in the `play` function.

```
def play(board, player):
    print()
    put_board(board)
    if finished(board):
        print()
        print(f"Player {next_player(player)} wins!")
    else:
        print()
        print(f"Player {player}")
        row = get_digit("Enter a row number: ")
        num = get_digit("Stars to remove: ")
        if valid(board, row, num):
            play(move(board, row, num), next_player(player))
        else:
            print()
            print("ERROR: Invalid move")
            play(board, player)
```

Each player enters their turn and the program automatically moves to the next player after each valid turn. The program exits when the board is empty via the `finished` function and prints out the winner. The first to take all the remaining stars in the board wins the game.

Finally, the `nim` function handles the operation of the game. It starts with player 1 by default.

```
def nim():
    play(initial(), 1)

nim()
```

3.2 Demonstration

An example of how the program works.

The game starts by printing the initial board, with all stars available.

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

Player 1 goes first, he/she enters a random line and a number of stars to remove.

```
Player 1
Enter a row number: 3
Stars to remove: 2
```

Two stars are removed from line 3. The board becomes:

```
1: * * * * *
2: * * * *
3: *
4: * *
5: *
```

It's now Player 2's turn. Now here's an example of an invalid move.

```
Player 2
Enter a row number: 6
Stars to remove: 1
```

The program prints an error message, prints the current board, and play reset to Player 2 again.

```
ERROR: Invalid move

1: * * * * *
2: * * * *
3: *
4: * *
5: *

Player 2
Enter a row number:
```

Play continue between players. A player can remove all stars in a line in one move.

```
Player 2
Enter a row number: 1
Stars to remove: 3

1: * *
2: * * * *
3: *
4: * *
5: *

Player 1
Enter a row number: 5
Stars to remove: 1

1: * *
2: * * * *
3: *
4: * *
5:

Player 2
Enter a row number: 2
Stars to remove: 4

1: * *
2:
3: *
4: * *
5:

Player 1
Enter a row number: 4
Stars to remove: 2

1: * *
2:
3: *
4:
5:

Player 2
Enter a row number: 1
Stars to remove: 1

1: *
2:
3: *
4:
5:
```

```
Player 1
Enter a row number: 3
Stars to remove: 1

1: *
2:
3:
4:
5:

Player 2
Enter a row number: 1
Stars to remove: 1

1:
2:
3:
4:
5:

Player 2 wins!
```

Player 2 removed the last star and won the game. The program prints out the winner and exit.

4 Conclusion

Functional programming in Python offers a robust framework for solving basic to complex problems through simpler, cleaner, and more efficient code. By applying Python's support for functional programming's principles such as higher-order functions, pure functions and immutability, developers can write code that is not only more predictable and easier to test but also more parallelizable, and less prone to side effects and errors.

Although Python is not a purely functional programming language, it provides enough features to support the benefits of the paradigm, making it a valuable tool for both starters and experts.

References

[1] Graham Hutton. *Programming in Haskell, Second Edition*, 2016.

[2] Turing. *Introduction to Functional Programming.* URL: https://www.turing.com/kb/introduction-to-functional-programming

[3] GeeksForGeeks. *Functional Programming Paradigm.* URL: https://www.geeksforgeeks.org/functional-programming-paradigm/

[4] W3Schools. *Python Tutorial.* URL: https://www.w3schools.com/python/

[5] GeeksForGeeks. *Functional Programming in Python.* URL: https://www.geeksforgeeks.org/functional-programming-in-python/

[6] Real Python. *Functional Programming in Python: When and How to Use It.* URL: https://realpython.com/python-functional-programming/

[7] Ryan Julian. *The Game of Nim.* Department of Mathematics - University of Wisconsin, USA. URL: $https://wiki.math.wisc.edu/images/Nim_sol.pdf$

Tower of Hanoi figure courtesy of O2 Education.