Coding Club: Functions & Recursion

Saif Latifi

April 2025

Agenda

- Fundamentals
  - What is a function?
  - How are they written?
  - Why use functions?
  - Best practices
- Recursion
- Functional programming

Fundamentals: What is a function?

Mathamatically speaking:

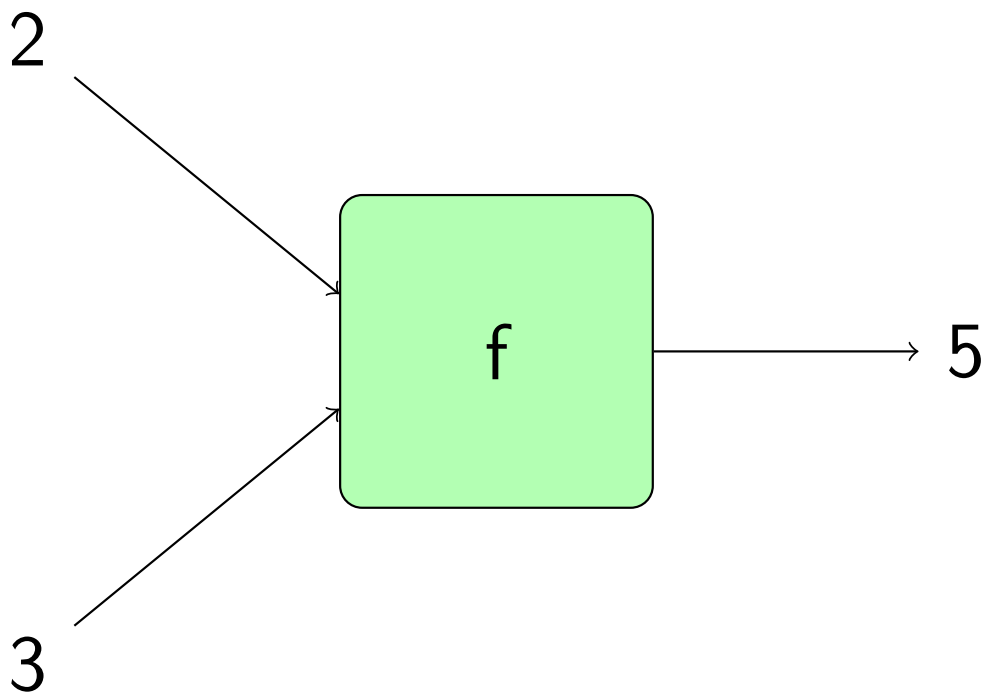
> *A function is a relation that maps elements from one set to another, such that each element in the domain is associated with exactly one element in the codomain.*

e.g.

$$f(x) = 2x + 1 \tag{1}$$

Fundamentals: What is a function?

Functions can have multiple inputs, i.e. multi-dimensionality.



Fundamentals: What is a function?

So applying this to programming, we can think of a function as a block of code that takes some input(s) known as parameters and returns a single output.

It's purpose is to perform a specific task or calculation.

Fundamentals: How are they written?

```
def function_name(parameter1, parameter2):
    # code block
    return result
```

The components of a function are:

- **Function name**: A descriptive name that indicates what the function does.
- **Parameters**: Variables that are passed into the function to be used within it.
- **Code block**: The set of instructions that define what the function does.
- **Return statement**: The value that the function outputs after executing its code block.

Fundamentals: How are they written?

Functions can also be written in a `lambda` form, which is a shorthand way to define small, compact functions.

```
sum = lambda num1, num2: num1 + num2
print(sum(2, 3)) # Output: 5
```

In this example, the `sum` function takes two parameters, `num1` and `num2`, and returns their sum.

Exercise 1

Exercise 1: Write functions for:

1. Checking whether a number is even.
2. Checking whether a number is odd (using the above).
3. Finding all even numbers under a given number.
4. Find all prime numbers under a given number.

Fundamentals: Why use functions?

- **Code reusability**: Functions allow you to write code once and reuse it multiple times.

Fundamentals: Why use functions?

- **Code reusability**: Functions allow you to write code once and reuse it multiple times.
- **Modularity**: Functions help break down complex problems into smaller, manageable pieces.

Fundamentals: Why use functions?

- **Code reusability**: Functions allow you to write code once and reuse it multiple times.
- **Modularity**: Functions help break down complex problems into smaller, manageable pieces.
- **Readability**: Functions make code easier to read and understand by providing clear names and purposes.

Fundamentals: Why use functions?

- **Code reusability**: Functions allow you to write code once and reuse it multiple times.
- **Modularity**: Functions help break down complex problems into smaller, manageable pieces.
- **Readability**: Functions make code easier to read and understand by providing clear names and purposes.
- **Testing**: Functions can be tested independently, making it easier to identify and fix bugs.

Fundamentals: Best practices

- Use descriptive names for functions and parameters.
- Keep functions small and focused on a single task.
- Avoid side effects (modifying global variables) within functions.
- Document your functions with comments or docstrings.
- Test your functions with various inputs to ensure correctness.

Fundamentals: Best practices

Find what's wrong with the following code:

```
def stuff(x,y):
  global z
  if y in x.split():
    z = True
  else:
    z = False

def main():
  print(stuff("hello world", "hello"))
```

Exercise 2

Exercise 2: Extend the password validation function in `lesson01-ex02.py` to check for:

- At least one uppercase letter
- At least one lowercase letter
- At least one digit
- At least one special character (e.g., @, !, ?, %, etc.)

Lambdas: A Warning!

While lambdas can be useful for short, simple functions, they can also lead to less readable code if overused or used inappropriately.

```
result = [(lambda x: x**2)(x) if (lambda y: y % 2
↪   == 0)(x) else (lambda z: z + 1)(x) for x in
↪   range(10) if (lambda w: w > 3)(x)]
```

Lambdas: A Warning!

While lambdas can be useful for short, simple functions, they can also lead to less readable code if overused or used inappropriately.

```
result = [(lambda x: x**2)(x) if (lambda y: y % 2
↪   == 0)(x) else (lambda z: z + 1)(x) for x in
↪   range(10) if (lambda w: w > 3)(x)]
```

Exercise 3: Rewrite the above code using regular functions.

Agenda

- Fundamentals
  - What is a function?
  - How are they written?
  - Why use functions?
  - Best practices
- Recursion
- Functional programming

Recursion

- A function that calls itself.

Recursion

- A function that calls itself.
- An alternative to iteration (loops).

Recursion

- A function that calls itself.
- An alternative to iteration (loops).
- Each recursive call should bring the function closer to a base case, which is a condition that stops the recursion.

Recursion: Example

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

What happens if we call `example(3)`?
What does this function do?

Recursion: Example

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

What happens if we call `example(3)`?
What does this function do?

What happens if we call `example(-1)`?

Recursion: Example

It could also be done using a loop:

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Recursion: Pros and Cons

- **Pros:**
  - Simplifies code for problems that have a recursive structure.
  - Easier to read and understand in some cases.
- **Cons:**
  - Can lead to stack overflow if the recursion depth is too high.

Exercise 4

Exercise 4: Write a recursive function to:

1. Calculate the nth Fibonacci number.
2. Reverse an array.

Agenda

- Fundamentals
  - What is a function?
  - How are they written?
  - Why use functions?
  - Best practices
- Recursion
- Functional programming

Functional Programming

- A programming paradigm that treats computation as the evaluation of mathematical functions.
- Based on the lambda calculus.
- Unlike imperative programming, it avoids changing state and mutable data (i.e., variables).
- Functions are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

Functional Programming

- A programming paradigm that treats computation as the evaluation of mathematical functions.
- Based on the lambda calculus.
- Unlike imperative programming, it avoids changing state and mutable data (i.e., variables).
- Functions are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

Functional Programming

- A programming paradigm that treats computation as the evaluation of mathematical functions.
- Based on the lambda calculus.
- Unlike imperative programming, it avoids changing state and mutable data (i.e., variables).
- Functions are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

Functional Programming

- A programming paradigm that treats computation as the evaluation of mathematical functions.
- Based on the lambda calculus.
- Unlike imperative programming, it avoids changing state and mutable data (i.e., variables).
- Functions are first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables.

Functional Progamming: An Example

We will use Scala for this example as it supports more functional programming features than Python.

```scala
def numMap(list: List[Int], f: Int => Int): List[Int]
↪  = {
    list match {
        case Nil => Nil
        case head :: tail => f(head) :: numMap(tail,
        ↪  f)
    }
}

val numbers = List(1, 2, 3, 4, 5)
val squaredNumbers = numMap(numbers, x => x * x)
println(squaredNumbers) // Output: List(1, 4, 9, 16,
↪  25)
```

Functional Programming: Key Differences

- **Immutability**: Data is immutable, meaning it cannot be changed once created.
- **Higher-order functions**: Functions can take other functions as arguments or return them as results.
- **Recursion**: Recursion is often used instead of loops for iteration.
- **Pure functions**: Functions that have no side effects and always produce the same output for the same input.

Functional Programming: Pros and Cons

- **Pros:**
  - Easier to reason about code.
  - Better support for parallelism and concurrency.
  - More concise and expressive code.
- **Cons:**
  - Can be less efficient due to immutability and recursion.
  - Steeper learning curve for those used to imperative programming.

Exercise 5

Exercise 5: Write functions in Scala (without using variables) to:

1. Calculate the factorial of a number.
2. Calculate the sum of a list of numbers.
3. Find the maximum element in a list.
4. Filter even numbers from a list.

End of Session!

Thank you for attending!