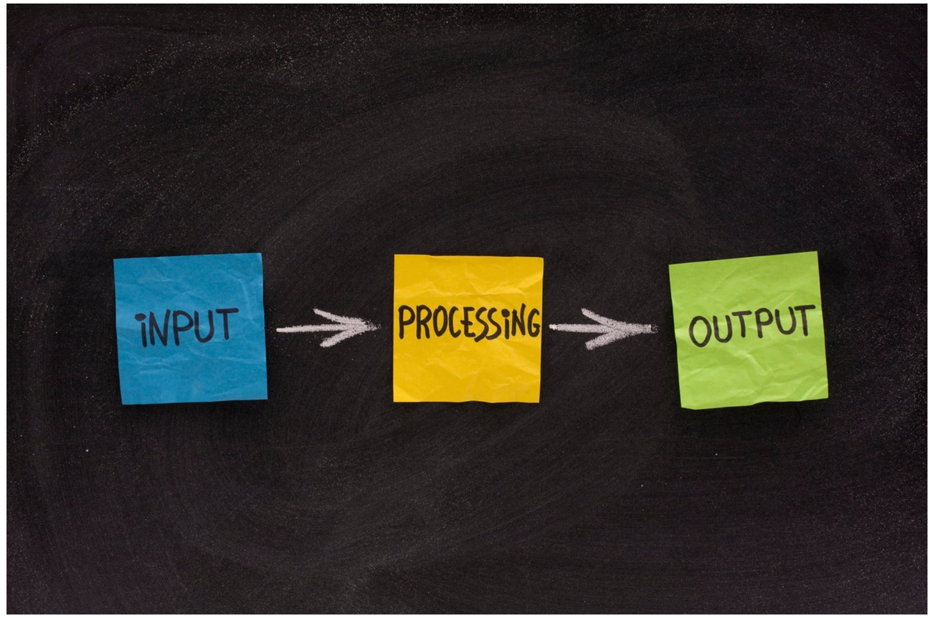
What is functional programming?

PROGRAMMING PARADIGM CONCEPTS

Eleanor Thomas
Senior Data Analytics Engineer

What is functional programming?

- **Functional programming:** a programming paradigm involving functions, specifically *pure functions*
- **Pure functions:** a process that takes input values, produces output values based only on the input values, and does not do anything else
- **Separation of responsibilities** is achieved in functional programming via functions



What is a pure function?

- Concept of **pure function** in functional programming comes from mathematics
- Pure functions only *look at input* and only *produce output*
- Pure functions have no **"side effects"**

- **No side effects** means:
  - No influence on other variables in the program
  - No writing to files
  - No saving information to a database

Example of a pure function

Pure function

```python
def pure_sum(x, y):
    output = x + y

    return output
```

Not a pure function

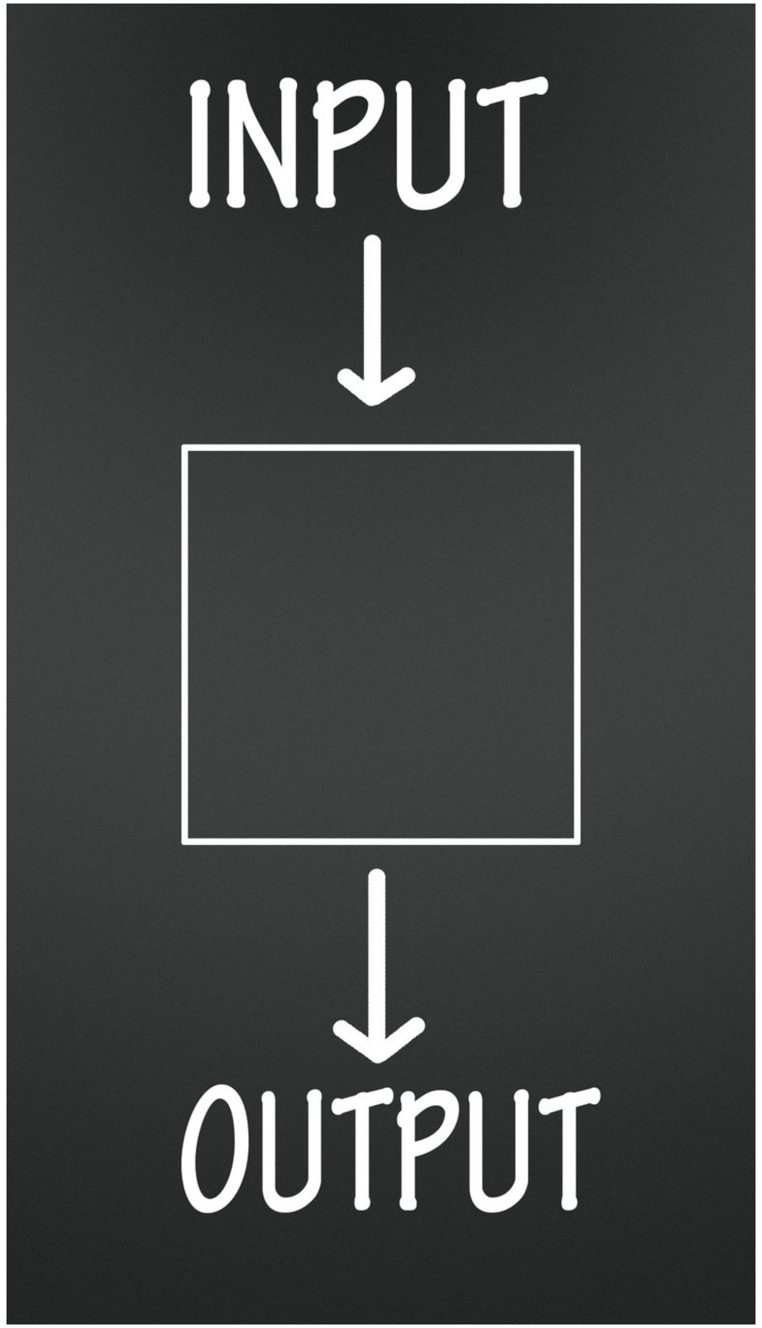
```python
def not_pure_sum(x, y):
    output = x + y

    print(output)

    return output
```

Benefits of pure functions

- Pure functions are easier to understand and debug
- Testing of pure functions is easier
- Output for a given input is entirely predictable
    - Similar to mathematical functions: 5 squared is always 25



Let's practice!

PROGRAMMING PARADIGM CONCEPTS

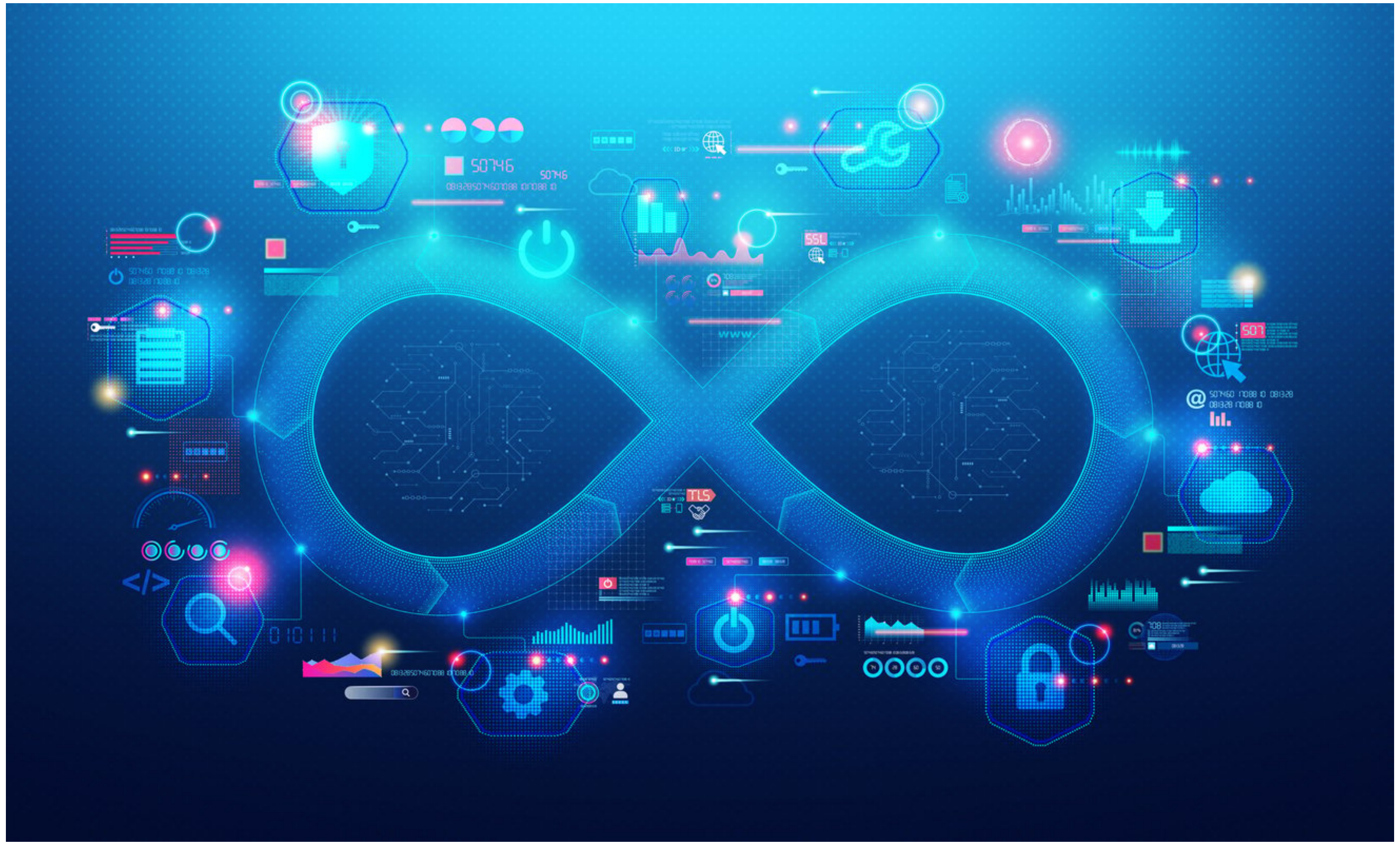
When is functional programming used?

PROGRAMMING PARADIGM CONCEPTS

Eleanor Thomas
Senior Data Analytics Engineer

Applications of functional programming

- Machine learning, deep learning, artificial intelligence
- Analyzing and processing large datasets
- Data engineering applications (e.g. in Scala and Clojure)



Example of functional programming

```
def process_data(raw_data):
    processed_data = raw_data

    ... further processing steps here! ...

    return processed_data
```

- Function takes input data stored in `raw_data`
- Function creates new variable for output data, called `processed_data`
- Function performs some consistent set of steps to further process the data
- Function returns `processed_data`

Pros and cons of functional programming

PROS

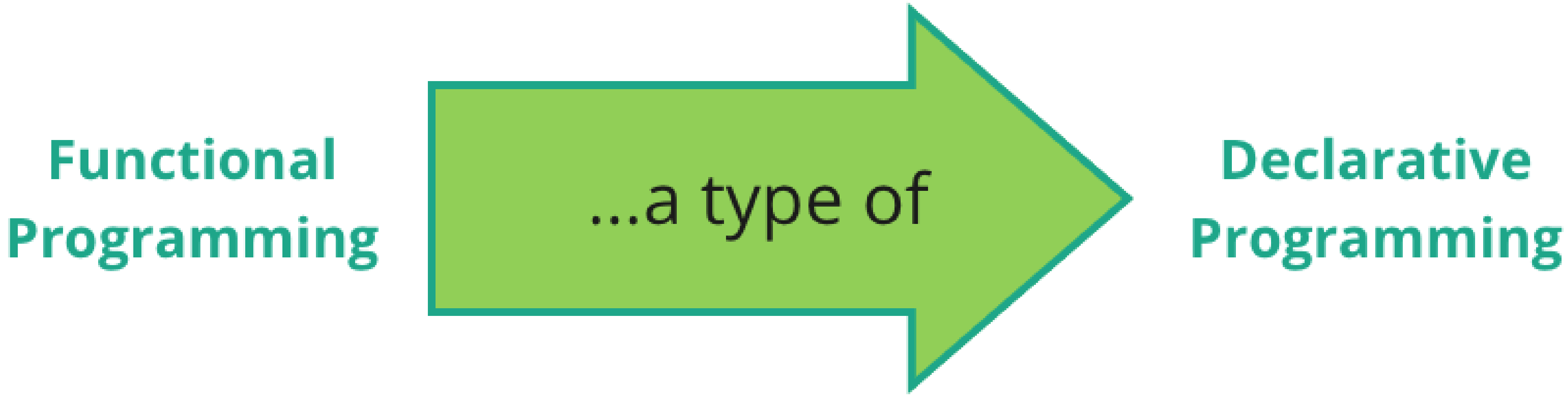
- Easier to read and debug pure functions
- Easier to test pure functions
- Fewer unexpected consequences in the code
- Pure functions are highly reusable from project to project
- Can run different functions in parallel to make code faster

CONS

- Tricky to get used to thinking in this paradigm, can feel limiting: "side effects" (writing to files, etc.) are most of what we *want* to do in programming
- Fewer experts, tools, frameworks exist for functional programming
- Steeper learning curve and fewer educational resources
- Larger memory usage limits applications

Functional programming and declarative programming

- Functional programming is a type of declarative programming
- Declarative programming: tell the computer *what* to do, not *how* to do it
- Functional programming is just *one type* of declarative programming
- Programmer tells the computer what functions to execute, not the exact steps to follow



Let's practice!

PROGRAMMING PARADIGM CONCEPTS

Functional programming in action

PROGRAMMING PARADIGM CONCEPTS

Eleanor Thomas

Senior Data Analytics Engineer

Functional programming in action

- Three examples of pure functions
- Key difference between a *pure* function and a general Python function: *no side effects*
- Pure functions can call other pure functions and remain pure

Example 1 - Writing a pure function

```python
def square_list(input_list):
    new_list = []
    for item in input_list:
        new_item = item ** 2
        new_list.append(new_item)
    return new_list
```

- First create a new, empty list
- Go through each item in the input list
  - Square it
  - Append it to new list
- Return new list

Example 2 - Correcting an "impure" function

```python
sample_mean = 10
scale_factor = 2

def scale_list(input_list):
    new_list = []
    for item in input_list:
        new_item = (item - sample_mean) / scale_factor
        new_list.append(new_item)
    return new_list
```

- Depends on variables outside of the function body
- Not a pure function

Example 2 - "Impure" function corrected

```
def scale_list(input_list, sample_mean, scale_factor):
    new_list = []
    for item in input_list:
        new_item = (item - sample_mean) / scale_factor
        new_list.append(new_item)
    return new_list
```

- Variables `sample_mean` and `scale_factor` have become input parameters for the function
- Function is now "pure"

Example 3 - Combining pure functions

```python
def scale_value(value, sample_mean, scale_factor):
    scaled_value = (value - sample_mean) / scale_factor
    return scaled_value

def scale_list(input_list, sample_mean, scale_factor):
    new_list = []
    for item in input_list:
        new_item = scale_value(item, sample_mean, scale_factor)
        new_list.append(new_item)
    return new_list
```

Let's practice!

Recursion in Functional Programming

PROGRAMMING PARADIGM CONCEPTS

Eleanor Thomas
Senior Data Analytics Engineer

What is recursion?

- *Recursive function*: a function that calls itself
- Must contain a termination condition, or base case
- Also contains the recursive call to itself with modified input

```python
def my_recursive_function(input_value):
    # base case
    if base_case_condition:
        return base_case_output_value
    # recursive call
    else:
        return my_recursive_function(modified_input_value) + some_modification
```

Why use recursion?

- Some problems are more straightforward when defined recursively
- Fibonacci numbers:
  - 0, 1, ...
  - 0, 1, 1, ...
  - 0, 1, 1, 2, ...
  - 0, 1, 1, 2, 3, ...

Some more examples of recursion

- Searching through a file system

Some more examples of recursion

- Searching through a file system
- Certain sort algorithms, such as Merge Sort

Some more examples of recursion

- Searching through a file system
- Certain sort algorithms, such as Merge Sort
- Various data structures are defined recursively

Recursion versus iteration

- Every recursive function can also be written iteratively
- Iterative function uses a loop rather than a recursive call

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

Let's practice!