



Functional Programming

Intended learning outcomes

By the end of this session, you will:

- be able to explain the principles of functional programming, including the avoidance of mutable data and the concept of functions as first-class citizens.
- be able to demonstrate the ability to apply functional programming concepts by constructing and using first-class functions in code exercises.
- be able to analyse code to identify and critique the use of recursion and side-effects, contrasting functional and imperative programming paradigms.



Introduction

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- In functional programming, functions are first-class citizens, meaning they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables.



Introduction

- Functional programming languages, encourage programming without side-effects.
- Recursion is a fundamental concept in functional programming, often used instead of traditional looping constructs.



Key Words

- **Functional Programming**
- **First-Class Functions**
- **Side-Effects**
- **Recursion**
- **Immutable Data**



Understanding Function Type

Definition:

- A function, represented by f , has a specific type denoted as $f : A \rightarrow B$

Components:

- Domain (A): The set of possible input values.
- Co-domain (B): The set where the output values are chosen.



Understanding Function Type

Example:

- Let's consider a function that maps students to their grades:

$$f : \text{Student} \rightarrow \text{Grade}$$

- Here, the domain is 'Student' (the set of all students) and the co-domain is 'Grade' (the possible grades a student can receive).



Key Points:

- A function takes each element from the domain and assigns it an output in the co-domain.
- Not every value in the co-domain needs to be an output, but every function output must be in the co-domain.



Example

- Domain** (Student): Alex, Bob, Carol, Dan, Ellie
- Co-domain** (Grade): A, B, C, D, F
- Now, let's say the following mappings exist:
 - Alex \rightarrow A
 - Bob \rightarrow B
 - Carol \rightarrow C
 - Dan \rightarrow B
 - Ellie \rightarrow A



Python Example



```

1 # Simple grading system
2
3 def grade_student(score):
4     """
5     This function takes a student's score as an input and returns the grade.
6     Domain: 0 <= score <= 100
7     Co-domain: {"F", "D", "C", "B", "A"}
8     """
9
10    if score < 50:
11        return "F"
12    elif 50 <= score < 60:
13        return "D"
14    elif 60 <= score < 70:
15        return "C"
16    elif 70 <= score < 90:
17        return "B"
18    else:
19        return "A"

```



Functions as First-Class Objects

- In functional programming, functions are considered as first-class citizens.
- When a function is treated as a "first-class object", it means it can be used in the same ways as other data types.
- This means they can be passed as arguments, returned from other functions, and assigned to variables or data structures.



1. Stored in a variable



- You can assign a function to a variable.

```

def greet():
    return "Hello!"

say_hello = greet
print(say_hello()) # This will print "Hello!"

```



2. Passed as an argument to another function



Activity 2a

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

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

def compute(func, value):
    return func(value)

print(compute(square, 4)) # This will print 16
print(compute(cube, 3))  # This will print 27
```



3. Returned from a function



Activity 2a

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

double = multiplier(2)
print(double(5)) # Prints 10
```



4. Data Structures



Activity 2b

```
3 def add(x, y):
4     return x + y
5
6 def subtract(x, y):
7     return x - y
8
9 def multiply(x, y):
10    return x * y
11
12 def divide(x, y):
13    return x / y
14
15 # Store functions in a list
16 operations = [
17     "+", add,
18     "-", subtract,
19     "*", multiply,
20     "/", divide
21 ]
```



Partial Functions

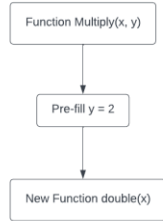


Activity 3a

```

1 from functools import partial
2
3 def multiply(x, y):
4     return x * y
5
6 double = partial(multiply, y=2)
7 print(double(4)) # Outputs: 8

```



Composition of Functions

- Function composition is the process of applying one function to the result of another function.
- Given two functions f and g , the composition of f and g is denoted as $f \circ g$ and defined as:

$$(f \circ g)(x) = (f(g(x)))$$

Let:

$$f(x) = x^2$$

$$g(x) = x + 3$$

then:

$$(f \circ g)(x) = f(g(x)) = f(x + 3) = (x + 3)^2$$

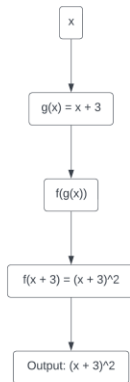


Image Processing



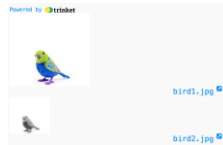
Activity 4

```

1 from PIL import Image, ImageFilter
2 # Define the individual transformations
3 def to_grayscale(img):
4     return img.convert("L")
5
6 def apply_blur(img):
7     return img.filter(ImageFilter.BLUR)
8
9 def reduce_size(img):
10    width, height = img.size
11    return img.resize((width // 2, height // 2))
12
13 # Now, to compose the functions:
14 def process_image(img):
15     return reduce_size(apply_blur(to_grayscale(img)))
16
17 # Load an image and apply all transformations
18 image = Image.open("bird.jpg")
19 processed_image = process_image(image)

```

Powered by DataCamp



Immutable Data



In functional programming ...

- data is immutable, it cannot be changed once it's created. Instead of modifying the original data, new data is produced whenever a change is required.
- this feature leads to more predictable and maintainable code in functional programming. It encourages the use of functions and recursion to manipulate data rather than updating variables in place.



Pure functions & Side Effects

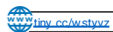


In functional programming ...

- functions should be pure, meaning their output should solely depend on their input, and they should not produce side effects.
- a side effect is any operation that modifies some state outside the function, like altering global variables or external files.
- A pure function shouldn't have any observable effects, besides returning a value to the caller.



Higher Order Functions



Functions that accept other functions as arguments, return functions, or both.

- **map**(function, list): Applies a function to each item of the list.
- **filter**(function, list): Returns items from the list for which function returns True.
- **reduce***(function, list): Continually applies a function to the elements of a list, reducing it to a single value

*The term reduce and fold are interchangeable



Head and Tail Operations



What are Head and Tail?

- **Head:** The first element of a list.
- **Tail:** The remaining elements after the head.

Example:

For the list [1, 2, 3, 4],
Head = 1
Tail = [2, 3, 4]



Procedural vs functional programming



```

1 # Imperative Programming
2 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3
4 # Double all numbers in the list
5 for i in range(len(numbers)):
6     numbers[i] = numbers[i] * 2
7
8 # Filter out even numbers
9 even_numbers = []
10 for num in numbers:
11     if num % 2 == 0:
12         even_numbers.append(num)
13
14 print(even_numbers)

```

```

16 # Functional Programming
17 from functools import partial
18
19 def multiply(val, multiplier):
20     return val * multiplier
21
22 def is_even(val):
23     return val % 2 == 0
24
25 numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
26
27 # Create a new function that doubles values
28 double = partial(multiply, multiplier=2)
29
30 # Doubling all numbers in the list
31 doubled_numbers = list(map(double, numbers))
32
33 # Filtering out even numbers
34 even_numbers = list(filter(is_even, doubled_numbers))
35
36 print(even_numbers)

```

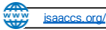



Intended learning outcomes

By the end of this session, you will:

- be able to explain the principles of functional programming, including the avoidance of mutable data and the concept of functions as first-class citizens.
- be able to demonstrate the ability to apply functional programming concepts by constructing and using first-class functions in code exercises.
- be able to will analyse code to identify and critique the use of recursion and side-effects, contrasting functional and imperative programming paradigms.

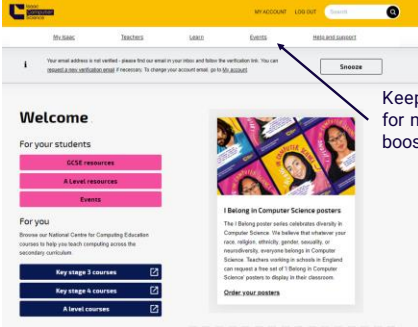


Isaac Computing



Functional Programming Booster Session

1	Compass yourself	Progressing parallelism / Functional programming	At level	>
2	Domain or co-domain?	Progressing parallelism / Functional programming	At level	>
3	Higher-order functions	Progressing parallelism / Functional programming	At level	>
4	What type?	Progressing parallelism / Functional programming	At level	>
5	Filtering	Progressing parallelism / Functional programming	At level	>
6	Function composition	Progressing parallelism / Functional programming	At level	>
7	Ready or talk?	Progressing parallelism / Functional programming	At level	>
8	Lucky 7	Progressing parallelism / Functional programming	At level	>

Check for more ISAAC boosters



Keep an eye out for more student booster events

Thank you



