Welcome to this CoGrammar Lecture: Extended Learning - Functions

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators
answering questions.

CoGrammar

Software Engineering Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** throughout this session, should you wish to ask any follow-up questions.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Software Engineering Session Housekeeping cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**

Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:

Ian Wyles
Designated Safeguarding Lead

Simone Botes

Nurhaan Snyman

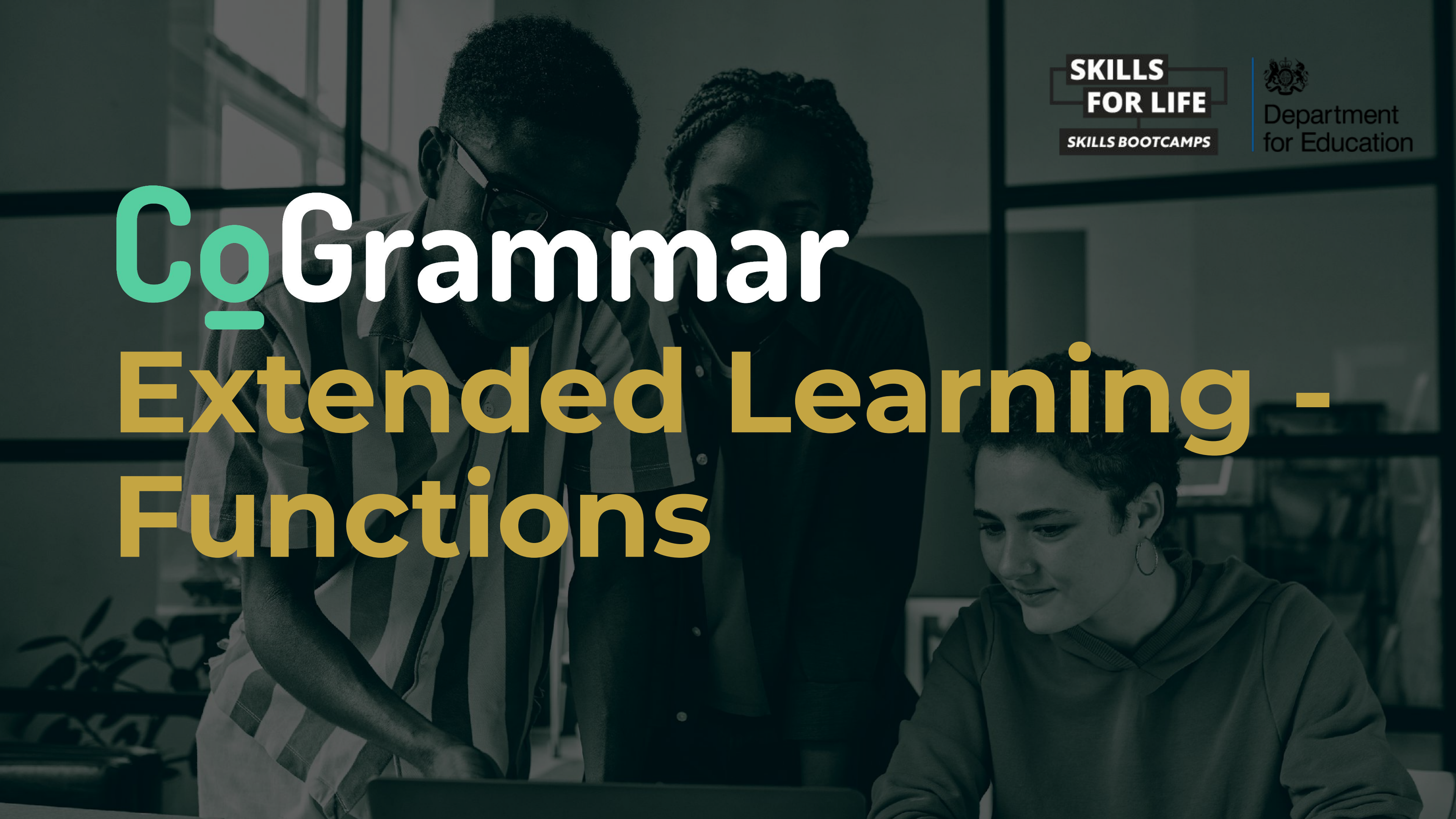
Rafiq Manan

Ronald Munodawafa

Tevin Pitts

Scan to report a safeguarding concern

or email the Designated Safeguarding Lead:
Ian Wyles
safeguarding@hyperiondev.com



SKILLS FOR LIFE

SKILLS BOOTCAMPS

Department for Education

CoGrammar

Extended Learning - Functions

Learning Objectives & Outcomes

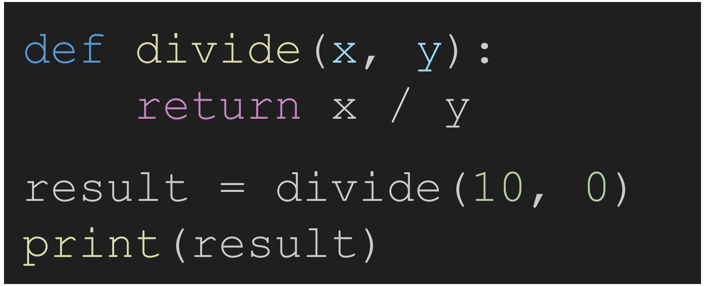
- Utilise built-in functions such as `print`, `len`, and `range`.
- `import` additional functions provided by Python
- Create functions, defining custom behaviours and logic to solve specific problems
- Design functions that `return` data to the caller, allowing for the output to be stored, printed, or further processed
- Apply higher-order functions, such as `map()`, `filter()`, and `reduce()`, and decorators

CoGrammar

Polls

Poll

1. Identify the error in the following code and its source using a stack trace

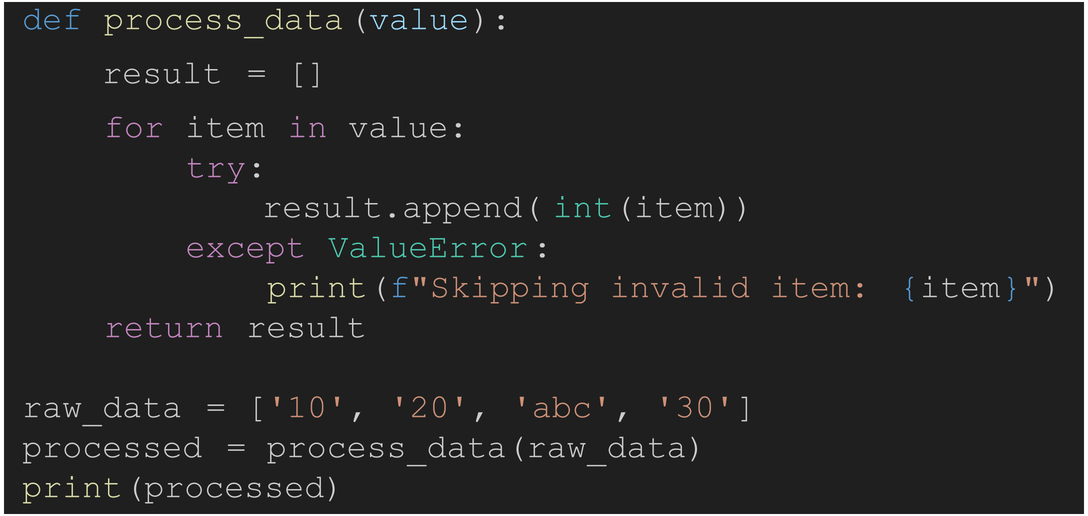


```python
def divide(x, y):
    return x / y

result = divide(10, 0)
print(result)
```

1. The stack trace will show a syntax error in the `print` statement.
2. The stack trace will show a missing return statement in the function `divide`.
3. **3.** The stack trace will show a division by zero error in the function `divide`

Poll

2. Analyze the following code. What is the output?

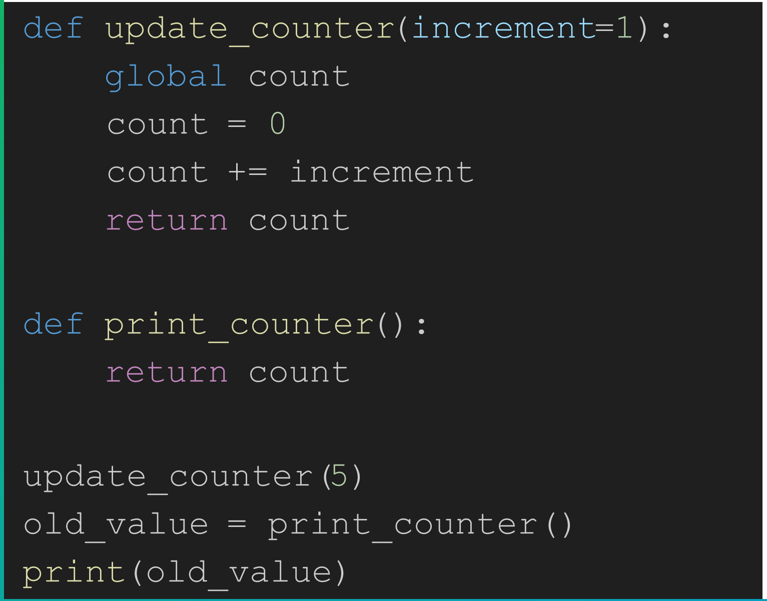


```python
def process_data (value):
    result = []
    for item in value:
        try:
            result.append( int(item))
        except ValueError:
            print(f"Skipping invalid item:  {item}")
    return result

raw_data = ['10', '20', 'abc', '30']
processed = process_data(raw_data)
print(processed)
```

1. [10, 20, 30, 'abc']
2. A runtime error occurs
3. [10, 20, 30]

Poll

3. What is the final value of count after these operations?



```
def update_counter(increment=1):
    global count
    count = 0
    count += increment
    return count

def print_counter():
    return count


update_counter(5)
old_value = print_counter()
print(old_value)
```

1. 5
2. 10
3. 0

Introduction

CoGrammar

Functions: The Heart of Programming Efficiency

Think of functions as tools in a workshop. Instead of carving every piece of wood by hand for a project, you use **specialized tools** to save **time**, reduce **effort**, and ensure **precision**. Similarly, in programming, functions are predefined tools or custom-made **solutions** that allow you to **automate** repetitive tasks, making your code more **efficient**, **organized**, and **reusable**.

Recap: Function Definition

CoGrammar

What are Functions?

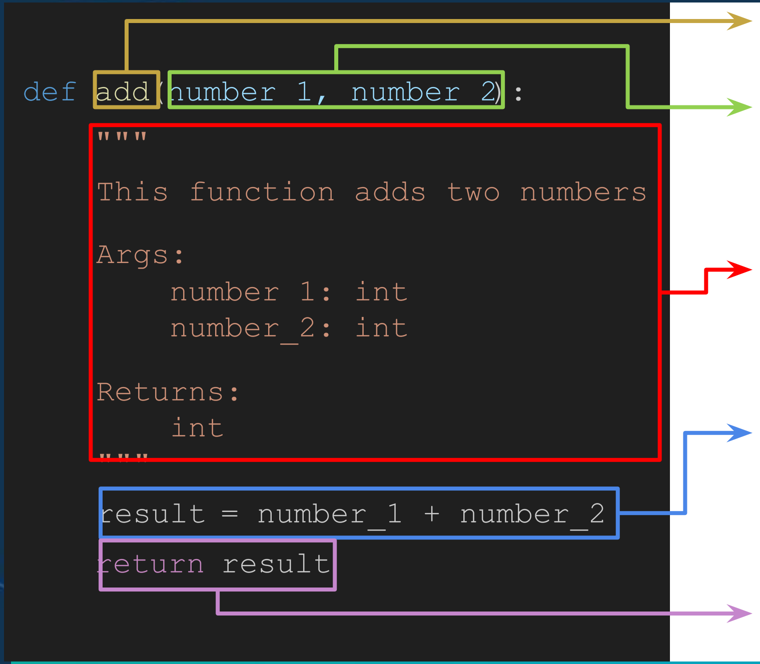
- Definition: Functions are self-contained blocks of code designed to execute specific tasks, promoting modularity and code reuse.
- Structure: A function definition includes a name, parameters (inputs), and a body containing the executable statements. The `return` statement specifies the function's output.
- Execution: Functions are invoked (called) by their name, optionally passing arguments that correspond to the defined parameters.
- Parameter Passing: Mechanisms for providing data to functions, allowing for flexible and dynamic behavior.

CoGrammar

Why Functions?

- Benefits:
  - Reusability: Write a function once and use it in different parts of your program, saving time and effort. DRY concept
  - Readability: Break down complex tasks into smaller, more manageable functions. This makes your code easier to understand and follow.
  - Maintainability: Functions isolate specific tasks, making it easier to update or modify code without affecting other parts of your program.

Function Syntax



```python
def add(number 1, number 2):
    """
    This function adds two numbers

    Args:
        number 1: int
        number_2: int

    Returns:
        int
    """
    result = number_1 + number_2
    return result
```

Function name: A unique identifier for the function

Parameters or Arguments : Inputs to the function, specified within parentheses.

docstring: *Optional* block used to document a module, class, or function. It explains what the code does, its parameters, and its return values.

Function Body: The block of code that performs the desired task.

`return` **Statement:** *Optional* statement that specifies the value returned by the function.

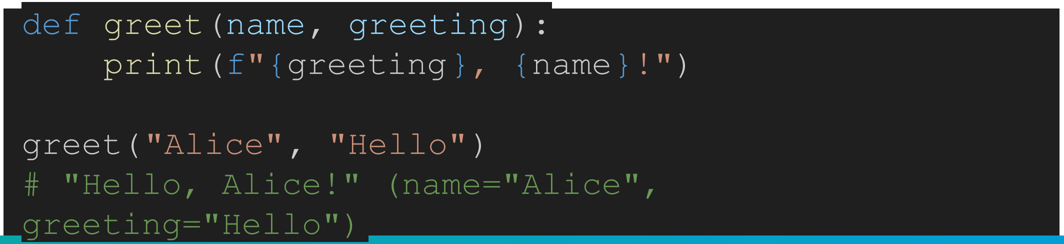
CoGrammar

Function Arguments in Python

CoGrammar

Normal (Positional) Arguments (**args)

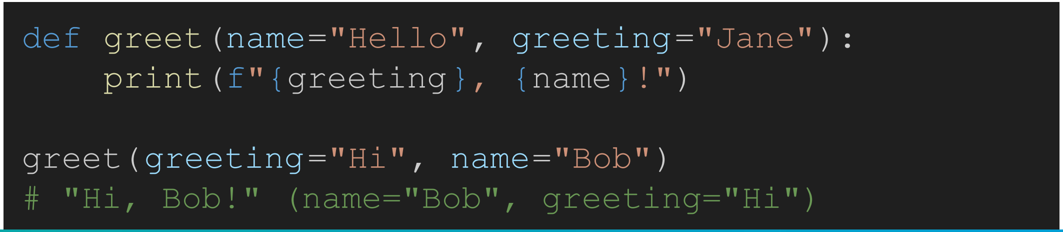
- Definition: Arguments passed to a function based on their position in the function call.
- Matching: The **order of arguments** in the call **must match** the order of parameters in the function definition.
- Syntax:



```python
def greet(name, greeting):
    print(f"{greeting}, {name}!")

greet("Alice", "Hello")
# "Hello, Alice!" (name="Alice",
greeting="Hello")
```

Keyword Arguments (**kwargs)

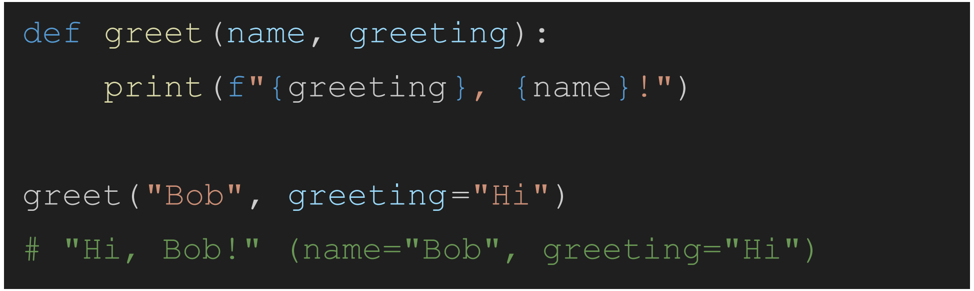
- Definition: Arguments passed to a function using the parameter name followed by an equals sign and the value (e.g., `parameter=value`).
- Matching: The **order of keyword arguments** in the call **does not need to match** the order of parameters in the function definition.
- Syntax:



```python
def greet(name="Hello", greeting="Jane"):
    print(f"{greeting}, {name}!")

greet(greeting="Hi", name="Bob")
# "Hi, Bob!" (name="Bob", greeting="Hi")
```

Combined Use

- **Normal arguments** must precede **keyword arguments** in a function call.



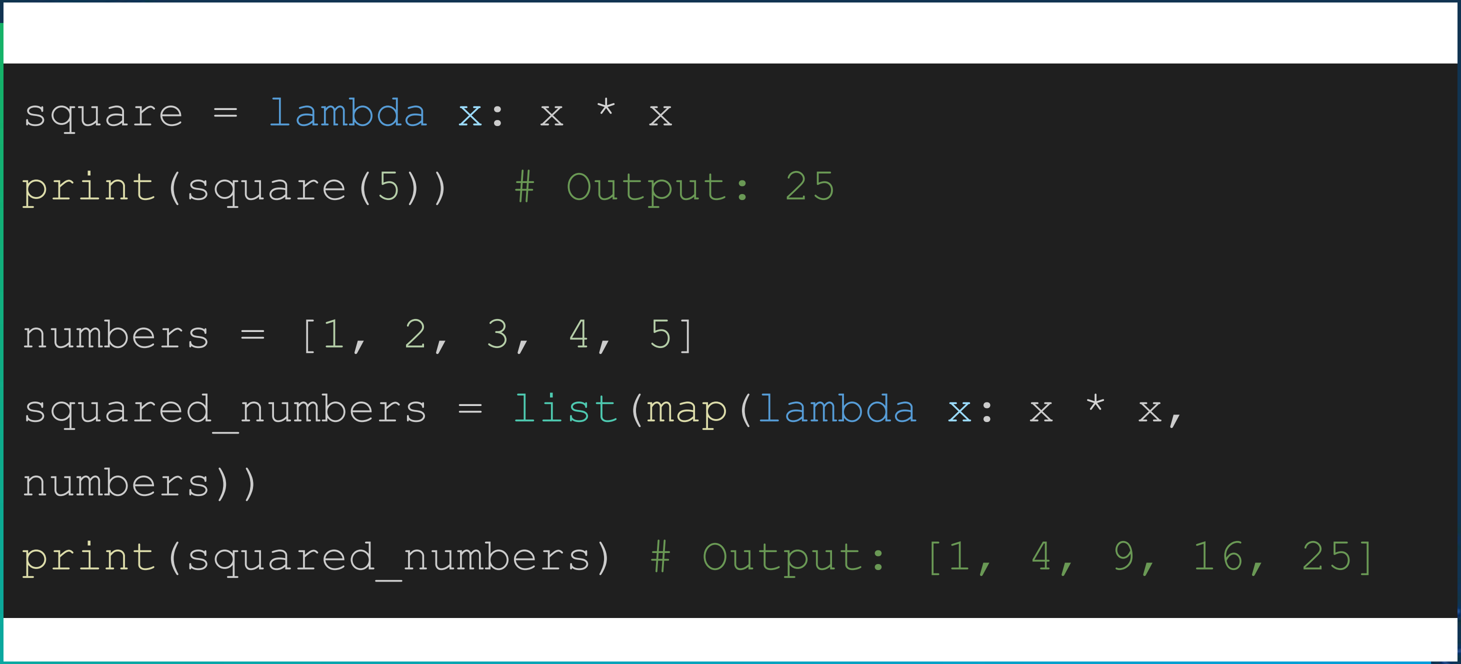
```python
def greet(name, greeting):
    print(f"{greeting}, {name}!")

greet("Bob", greeting="Hi")
# "Hi, Bob!" (name="Bob", greeting="Hi")
```

Anonymous Functions (Lambda Expressions)

- Definition: Anonymous functions are small, single-expression functions that can be defined inline without a formal name.
- Keyword: Created using the `lambda` keyword.
- Syntax: `lambda arguments: expression`
  - `arguments`: Comma-separated list of input parameters.
  - `expression`: Single expression that is evaluated and returned.
- Use Cases: Often used for short operations within other functions (e.g., `map`, `filter`, `sort`).

CoGrammar

Anonymous Functions (Lambda Expressions)



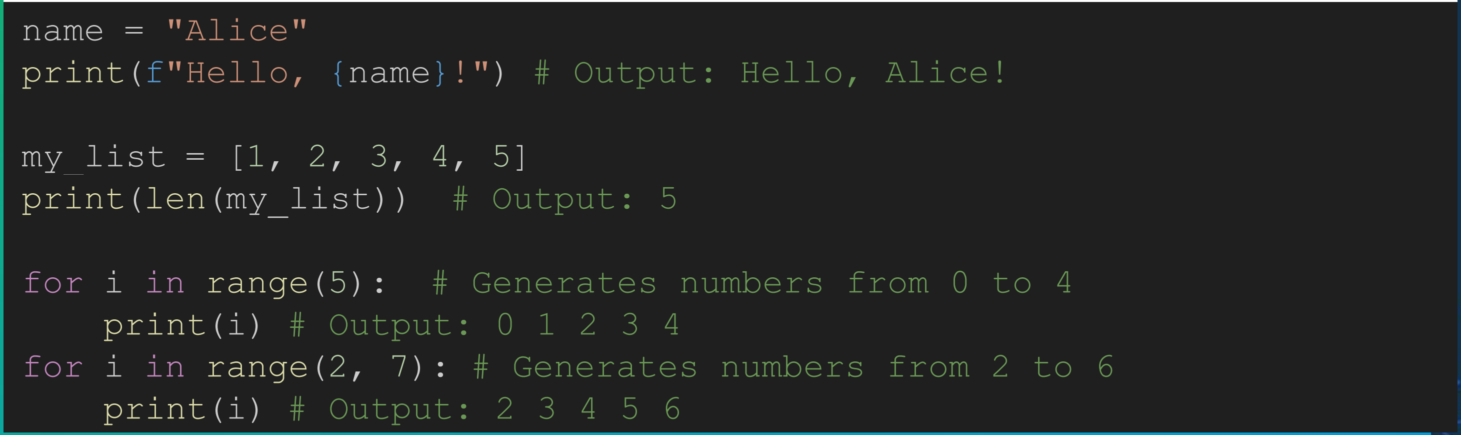
```
square = lambda x: x * x
print(square(5))  # Output: 25


numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x * x,
numbers))
print(squared_numbers) # Output: [1, 4, 9, 16, 25]
```

Built-In Functions and Imports

CoGrammar

Python's Built-in Functions: Essential Tools

- What are they?: Built-in functions are functions that are readily available in Python without needing any imports. They provide core functionalities for common tasks.

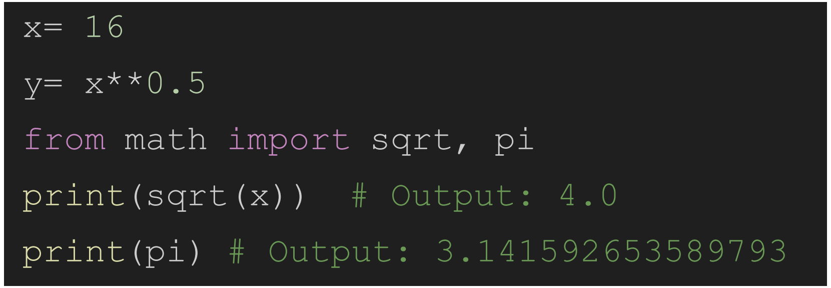


```python
name = "Alice"
print(f"Hello, {name}!") # Output: Hello, Alice!

my_list = [1, 2, 3, 4, 5]
print(len(my_list))  # Output: 5

for i in range(5):  # Generates numbers from 0 to 4
    print(i) # Output: 0 1 2 3 4
for i in range(2, 7): # Generates numbers from 2 to 6
    print(i) # Output: 2 3 4 5 6
```

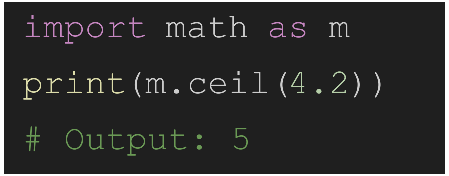
Modules and Imports: Extending Python's Power

- Python Standard Library: A vast collection of modules providing additional functionalities beyond the built-in functions.
- `import` Statement: Used to access functionality from modules.
- Import Methods:
  - `import module_name`: Imports the entire module. Access functions using `module_name.function_name()`
  - `from module_name import function1, function2`: Imports specific functions from a module. Access functions directly.
  - `import module_name as alias`

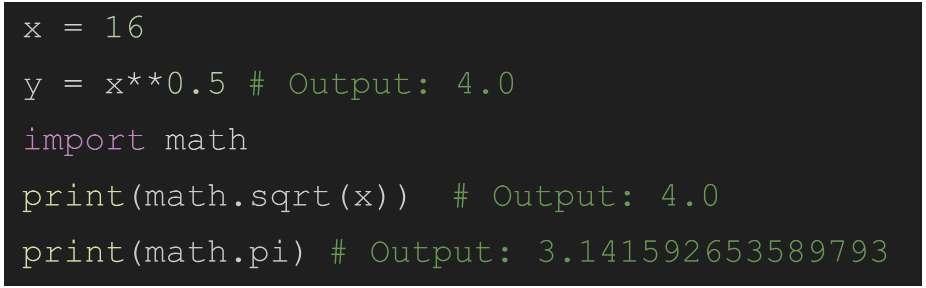
Modules and Imports: Extending Python's Power



```
x= 16
y= x**0.5
from math import sqrt, pi
print(sqrt(x))  # Output: 4.0
print(pi) # Output: 3.141592653589793
```



```
import math as m
print(m.ceil(4.2))
# Output: 5
```



```
x = 16
y = x**0.5 # Output: 4.0
import math
print(math.sqrt(x))  # Output: 4.0
print(math.pi) # Output: 3.141592653589793
```

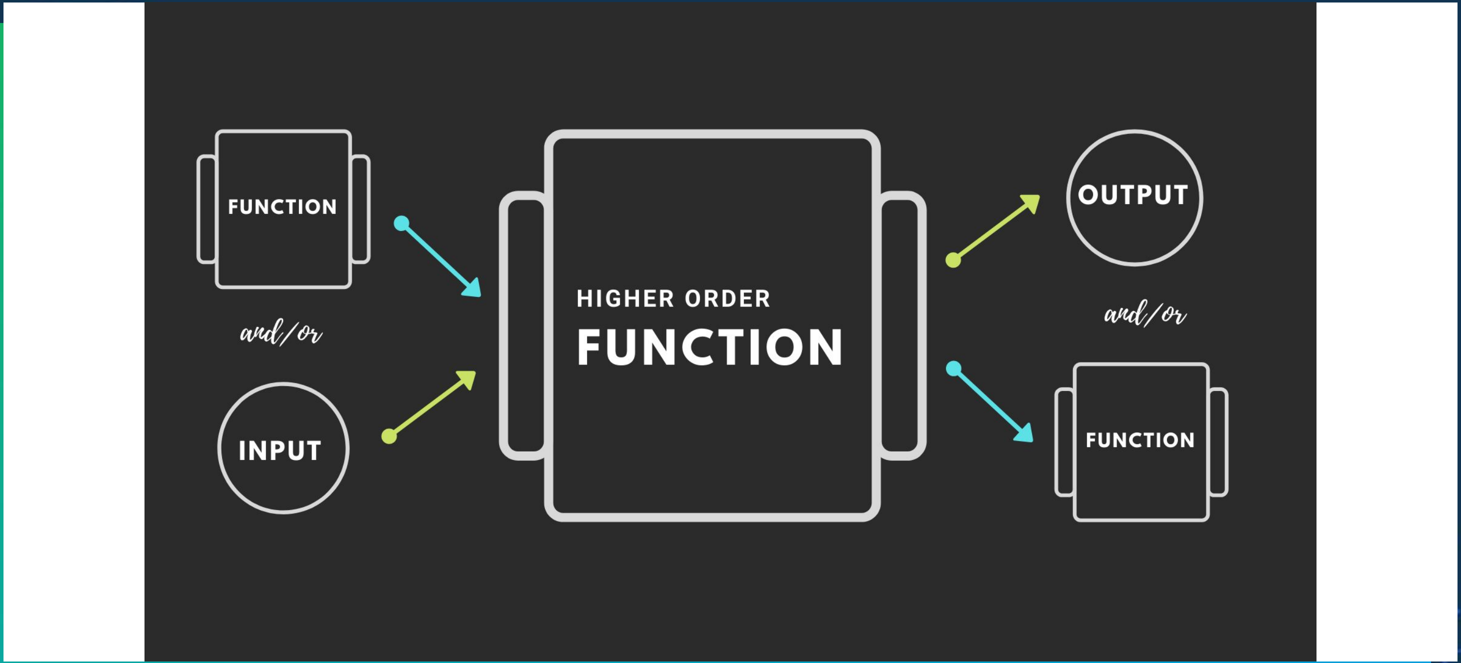
Higher-Order Functions

CoGrammar

What Are Higher-Order Functions?

- Definition: Functions that take other functions as arguments or return functions.
- Purpose: Enables code reuse, abstraction, and more concise code.
- Key Idea: Functions are treated as first-class objects.

What makes Higher-Order Functions so powerful?



CoGrammar

Applying Higher-Order Functions: map(), filter() & reduce()

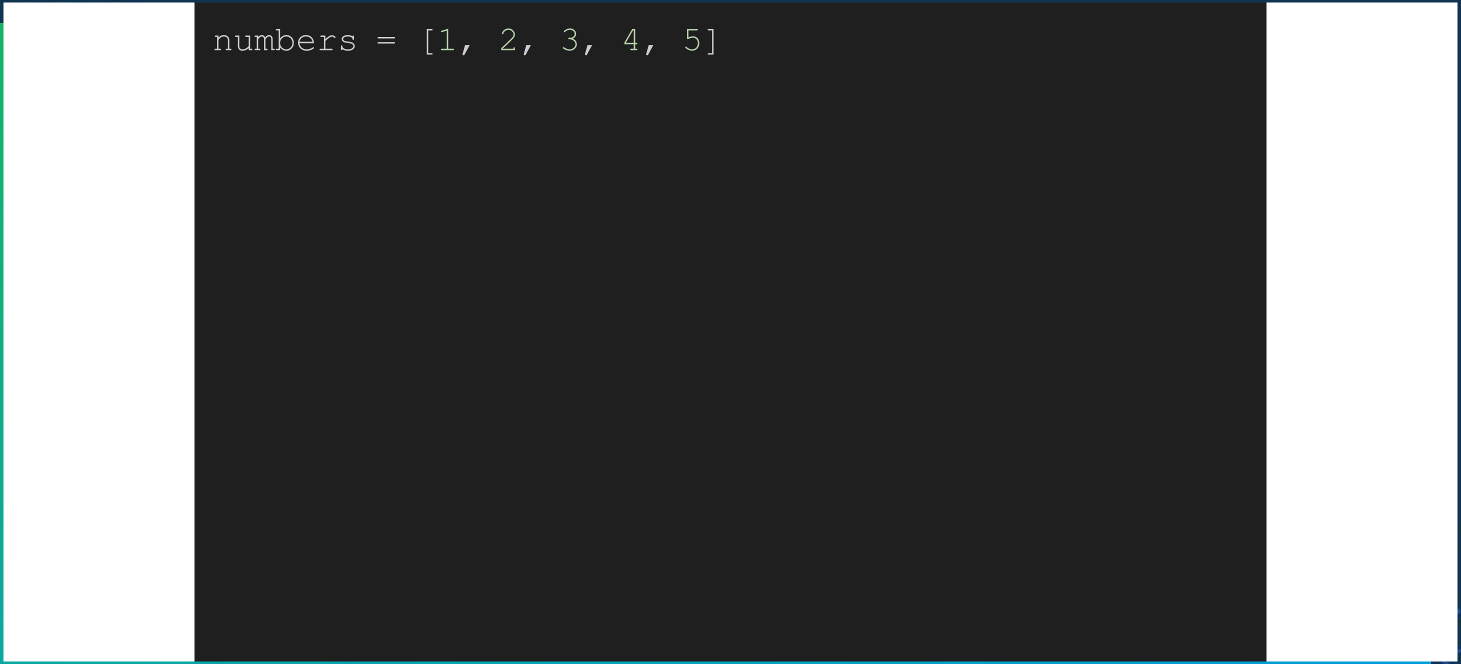
- map:
  - Applies `function` to each item in `iterable`.
  - Syntax: map(function, iterable)

- filter:
  - Filters `iterable` based on `function` (returns `True`/`False`).
  - Syntax: filter(function, iterable)

- reduce:
  - Reduces `sequence` to a single value using `function`. (Requires `from functools import reduce`)
  - Syntax: reduce(function, iterable)

CoGrammar

Applying Higher-Order Functions: map(), filter() & reduce()



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

Introduction to Decorators

CoGrammar

Decorators: Enhancing Function Behavior

- Definition: Decorators are a powerful feature in Python that allows you to modify or enhance the behavior of functions (or methods) without actually changing their core code

- Purpose: They provide a clean and reusable way to add functionalities like Logging

- Analogy: Think of decorators like wrapping a gift. The gift itself (the original function) remains unchanged, but the wrapping (the decorator) adds extra features or presentation.

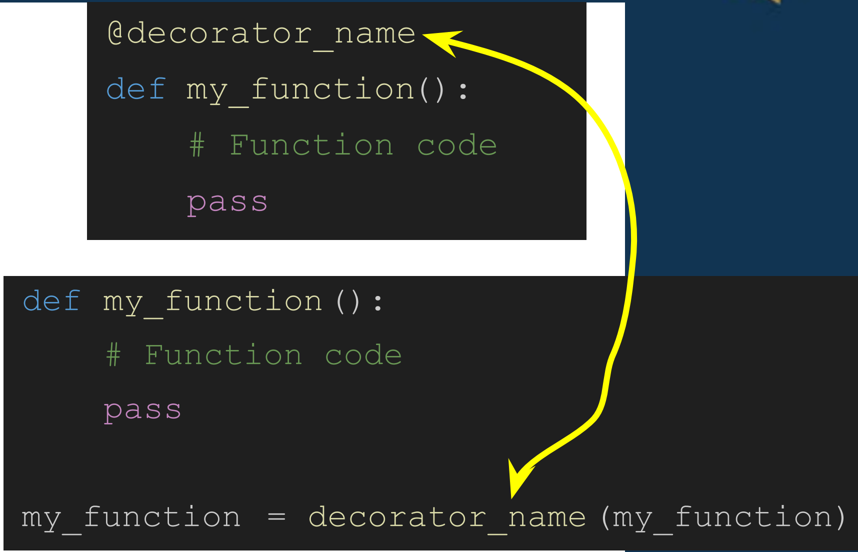
CoGrammar

How Decorators Work: Syntax and Usage

- Syntax: Decorators are applied using the `@` symbol followed by the decorator function's name, placed directly above the function you want to decorate.

- Behind the scenes: The `@` syntax is **syntactic sugar**. It's equivalent to:

- Flexibility and Reusability: Decorators can be applied to multiple functions, making code more DRY (Don't Repeat Yourself). They promote code organization and maintainability.



```
@decorator_name
def my_function():
    # Function code
    pass
```

```
def my_function():
    # Function code
    pass

my_function = decorator_name(my_function)
```

Practical

Filtering Adults and Minors, Calculating retirement age, and computing average Age with Higher-Order Functions

1. **Objective:** The objective of this exercise is to practice using Python's higher-order functions (**`filter()`**, **`map()`** and **`reduce()`**) along with decorators to filter, transform, and analyze data.
2. **Steps to Implement:**
   - Use loops and conditions, then, use `filter()` to get a list of adults (18+),
   - Use loops and conditions, then apply `map()` to calculate the years left for each adult to reach retirement age (65).
   - Use loops and conditions, then calculate the average age of the group using `reduce()`
   - Implement a simple decorator to perform the operations above.

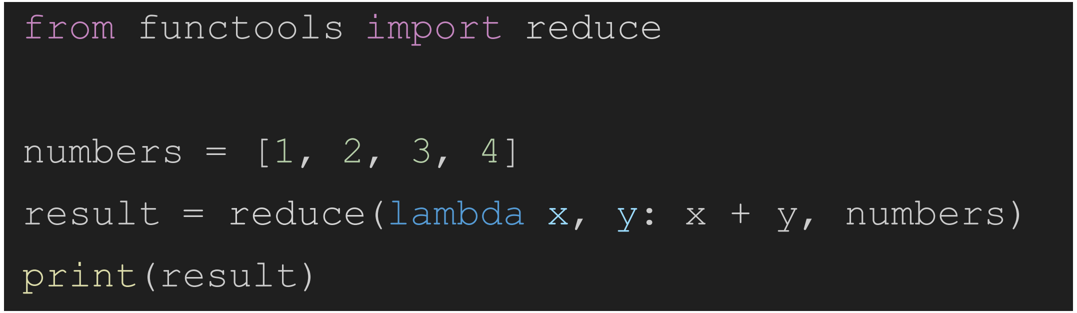
CoGrammar

Polls

CoGrammar

Poll

1. What will the following code output?

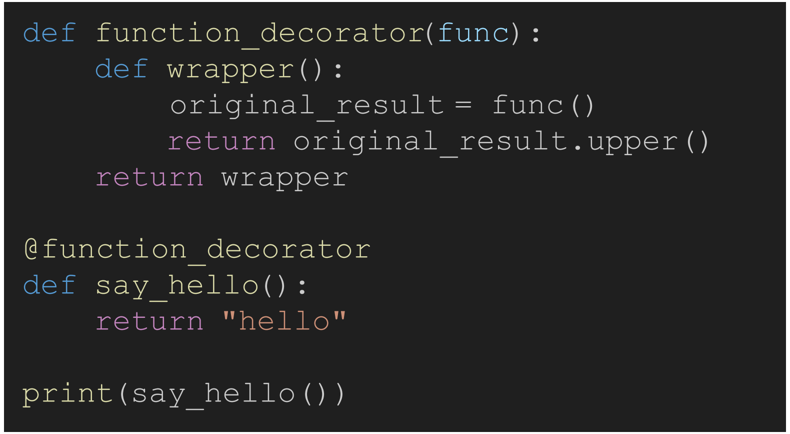


```
from functools import reduce


numbers = [1, 2, 3, 4]
result = reduce(lambda x, y: x + y, numbers)
print(result)
```

a. 24
b. [2,4,6,8]
c. 10

Poll

2. In the following code snippet, what will be the final value of `result`?



```python
def function_decorator(func):
    def wrapper():
        original_result = func()
        return original_result.upper()
    return wrapper

@function_decorator
def say_hello():
    return "hello"

print(say_hello())
```

1. HELLOhello
2. HellO
3. HELLO

Lesson Conclusion and Recap

Recap the key concepts and techniques covered during the lesson.

- **Built-In Functions**: Use for common tasks like printing, finding lengths, or generating sequences.
- **Custom Functions**: Define flexible, reusable blocks of logic.
- **Higher-Order Functions**: Simplify data processing with `map`, `filter`, and `reduce`.
- **Decorators**: Dynamically enhance function behavior.

Resources

- **Additional Resources**
  - Lists
  - Glossary — Python 3.13.1 documentation
  - 2. Functions and Modules

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE

SKILLS BOOTCAMPS

Department for Education

CoGrammar