Welcome to this CoGrammar Lecture:

Functions

The session will start shortly...

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

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

CoGrammar HyperionDev


SKILLS FOR LIFE
SKILLS BOOTCAMPS
Department for Education
CoGrammar
Functions

Learning Objectives & Outcomes

- Explain the purpose and benefits of using functions in programming.
- Define and call functions with parameters and return values.
- Implement functions to modularise and organise code effectively.
- Describe the concept of variable scope and its importance in programming.
- Differentiate between local and global scope.

Learning Objectives & Outcomes

- Explain what a stack trace is and how it is generated during program execution.
- Interpret stack traces to debug and identify the source of errors in their code.
- Use stack traces to trace the flow of function calls and follow the sequence of execution.
- Implement the steps in debugging

Polls

CoGrammar

Poll

1. Given the following code snippet, which of the following statements is true?

```
my_string = "Hello World"
print(my_string[0:5].upper() + my_string[6:])
my_list = [1, 2, 3, 4]
my_list.append(5)
my_dict = {"key1": "value1", "key2": 10}
my_dict["key3"] = 20
print(my_dict["key2"] + my_list[2])
```

    1. The code prints "HELLO WORLD" and 12.
    2. The code prints "HELLO World" and 14.
    3. The code prints "HELLO World" and 13

Poll

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

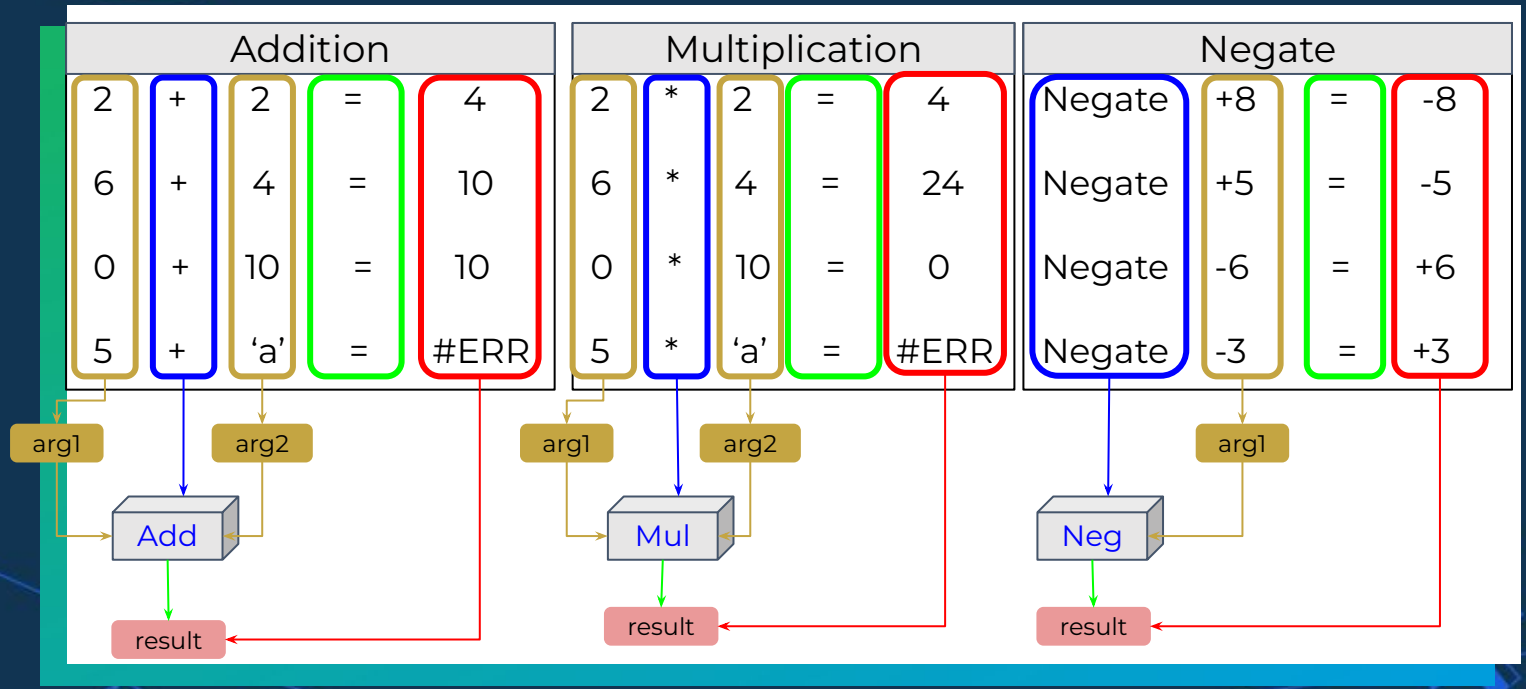
```
my_string = "Software Engineering"
print(f"{my_string[:8].title()}                    -
{my_string[9:].upper()}")
my_list = [1, 2, 3]
my_list[2] = 4
my_dict = {"Department": "IT", "Employees": 50}
my_dict["Employees"] -= 5
print(my_list, my_dict)
```

1. "Software - engineering", [1, 2, 3], {"Department": "IT", "Employees": 45}.
2. "SOFTWARE - Engineering", [1, 2, 4], {"Department": "IT", "Employees": 50}.
3. "Software - ENGINEERING", [1, 2, 4], {"Department": "IT", "Employees": 45}.

Introduction

CoGrammar

Intuition: Arithmetics



CoGrammar

Function Definition, Calling, and Parameter Passing

CoGrammar

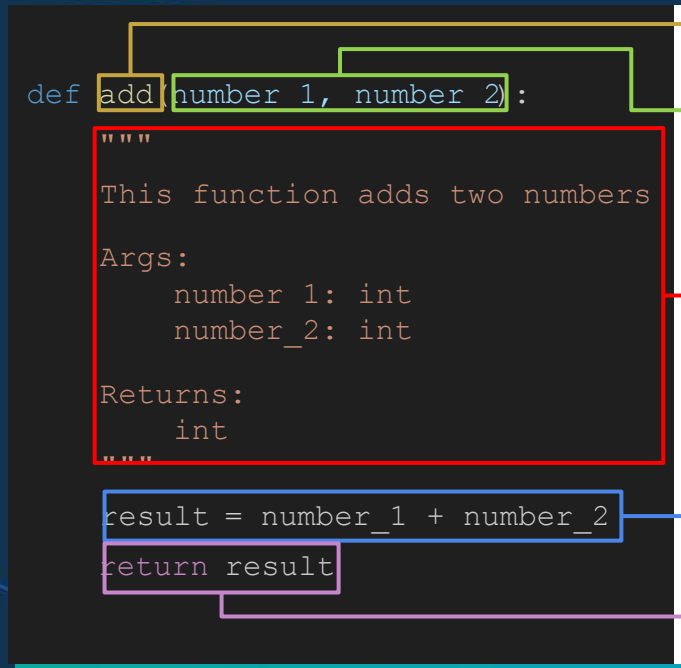
What are Functions?

- Definition: Functions are blocks of code that perform specific tasks. They allow you to encapsulate functionality for reuse, making programs modular and efficient.
    - Parameters: Input values passed to the function when called.
    - Return Statement: Outputs the result of a function, enabling further computation.

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



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.

Return Statement

- In Python, the `return` statement is like sending a message back to where you called a function from.
- It's a way for the function to finish its job and share its final answer with the rest of the program.

Imagine you ask a friend to solve a math problem for you. After working on it, your friend comes back to you with the solution. In Python, the `return` statement is like your friend giving you that solution. It's the way the function tells the rest of the program what answer it found.

Default arguments and keyword arguments

- Definition: Default arguments allow you to specify default values for function parameters. If the caller doesn't provide a value for that parameter, the default value is used.
- Syntax:

```
def function_name(parameter1, parameter2=default_value,
...):
    # Function code
```

CoGrammar

Calling functions

- Functions with one required positional input:
  - my_function(input1)
- Functions with two required positional inputs:
  - my_function(input1, input2)
- Functions with one required positional input and one optional keyword input:
  - my_function(input1, keyword_arg=input2)

Comparing Programs: With vs. Without Functions

Without Functions:

```
print("Hello, Alice!")
print("Hello, Bob!")
print("Hello, Charlie!")
```

With Functions:

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

greet("Alice")
greet("Bob")
greet("Charlie")
```

Examples

```
def print_hello():
    print('Hello!')

print_hello()  # Prints Hello!
```

```
def calculate_area(length=0, width=0):
    area = length * width
    return area

print(calculate_area(5, 6)) # Prints 30
print(calculate_area(8, 6)) # Prints 48
```

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

print(greet("Julien")) # Prints Hello Julien!
```

Built-in Functions

- `print()`: Outputs data to the console.
- `len()`: Returns the length of an object.
- `range()`: Generates a sequence of numbers.
- `sum()`: Returns the sum of elements in an iterable.
- `type()`: Returns the type of an object.

Variable Scope (Local and Global)

CoGrammar

What is a Scope?

- Definition: Scope refers to the accessibility of variables in different parts of a program. Variables defined within a function are typically scoped to that function, meaning they can only be accessed within that function.
- Types of scopes:
    - Global scope: Variables defined outside of any function, accessible throughout the entire program. Use sparingly to maintain code clarity
    - Local scope: Variables defined within a function, only accessible within that function. Destroyed after function execution

Local Scope

```
def local_scope():
    x = 10
    print(x)

local_scope()
print(x)  # Error: NameError: name 'x' is not
defined
```

Global Scope

```
x = 15 # <-- Defined outside the function

def global_scope():
    print(x)
    global y # <-- Defined within the function
    y = 10

global_scope()
print(x)  # Outputs 15
print(y)  # Outputs 10 <-- Accessible outside the function
```

Code Modularization with Functions

CoGrammar

What is a Modularisation?

- Definition: Breaking down a complex program into smaller, well-defined functions or modules. This promotes code organization and maintainability.
- Benefits:
    - Improved code organization: Modular code is easier to understand and follow for both you and others.
    - Enhanced Reusability: Functions can be reused in different parts of your program, reducing code duplication
    - Easier Testing and Debugging: By isolating specific tasks within functions, testing and debugging become more manageable.

Example: Breaking Down a Task

```
data = [1, 2, 3, 4, 5]
total = 0
for num in data:
    total += num
avg = total / len(data)
print(f"Total: {total},
Average: {avg}")
```

Without Modularisation

Example: Breaking Down a Task

```python
def calculate_total(data):
    total = 0
    for num in data:
        total += num
    return total

def calculate_average(total, count):
    return total / count

data = [1, 2, 3, 4, 5]
total = calculate_total(data)
average = calculate_average(total, len(data))
print(f"Total: {total}, Average: {average}")
```

With Modularisation

Stack Traces and Error Interpretation

CoGrammar

What is a Stack Trace?

- Definition: A stack trace is a report showing the sequence of function calls leading to a specific point in a program, usually an error.
- How Stack Traces are Generated: When an error occurs, Python provides a stack trace that identifies the error type, message, and the function calls involved.
- Reading Stack Traces: Stack traces include the error type, message, and the sequence of functions that were called before the error happened. This information helps pinpoint where the error originated in your code.

Decoding a Stack Trace

When an error occurs, Python captures the current state of the program's execution stack and prints a stack trace to the console. This stack trace includes:

- Error Type: The type of error that occurred (e.g., SyntaxError, NameError, TypeError).
- Error Message: A description of the error, providing additional details about what went wrong.
- Traceback: A list of function calls, starting from the point where the error occurred and going back to the initial entry point of the program.

Stack Trace Example

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

def main():
    result = divide(10, 0)

main()
```

```
Traceback (most recent call last):
  File "example.py", line 7, in <module>
    main()
  File "example.py", line 5, in main
    result = divide(10, 0)
  File "example.py", line 2, in divide
    return x / y
ZeroDivisionError: division by zero
```

Debugging Process and Techniques

CoGrammar

Debugging in Python

- Debugging is the process of identifying and fixing errors or bugs in a program.
- It involves analysing the behaviour of the code to understand why it is not working as expected and making the necessary corrections to resolve the issues.
- Debugging is an essential skill for programmers, and mastering it can greatly improve your ability to write reliable and efficient code.

Steps in Debugging...

- Reproduce the Error: Consistently triggering the error to understand when and where it occurs.
- Examine Error Messages: Note any error messages or exceptions.
- Use Print Statements: Add print or log statements to track variable values and execution flow, helping identify issues in the code.

Steps in Debugging...

- Inspect Data: Verify input data and intermediate values to ensure accuracy and correct manipulation.
- Use Debugging Tools: Utilize IDE debugging features like breakpoints, step-through debugging, and variable inspection.
- Isolate the Problem: Focus on the specific code section causing the error to identify the root issue.

Steps in Debugging...

- Fix the Issue: Correct the problem by updating code, logic, or data structures.
- Test the Fix: Re-run the program to ensure the issue is resolved and behavior is as expected.
- Document Changes: Record the steps taken and fixes applied for future reference and team communication.

Polls

CoGrammar

Poll

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

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

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

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

Poll

```
def process_data(value):
    global result result = []
    for item in value:
        if item.isdigit():
            result.append(int(item))
        else:
            print(f"Skipping invalid item:
{item}")
    return result

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

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

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

CoGrammar

Poll

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

def reset_counter():
    global count
    previous = count
    count = 0
    return previous

update_counter(5)
old_value = reset_counter()
```

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

1. 5
2. 10
3. 0

CoGrammar

Lesson Conclusion and Recap

Recap the key concepts and techniques covered during the lesson.

- **Function Definition, Calling, and Parameter Passing**: How to create functions (using def), call them, pass data using parameters, and return values.
- **Variable Scope (Local and Global)**: The difference between variables accessible only within a function (local) and those accessible everywhere (global).
- **Code Modularization with Functions**: Using functions to break down complex problems into smaller, reusable parts for better organization.
- **Stack Traces and Error Interpretation**: Understanding stack traces (records of function calls during errors) to identify error locations.
- **Debugging Process and Techniques**: Steps and methods for finding and fixing errors, including using print statements or debuggers.

Practical: Simple Calculator

1. **Objective:** To build a simple calculator that performs basic arithmetic operations (addition, subtraction, multiplication, division) using functions. This exercise will help learners understand function definition, calling, and parameter passing, variable scope, code modularization, stack traces, and debugging techniques.
2. **Steps to Implement:**
    - Define Arithmetic Operations without Functions
    - Modularized Operations with Functions
    - Variable Scope (Local and Global)
    - Code Modularization with Functions
    - Stack Traces and Error Interpretation
    - Debugging Process and Techniques

Resources

- **Additional Resources**
  - Functions
  - 2. Functions and Modules

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar