# MODULE 1: CORE PYTHON & DATA

WEEK: 1 LECTURE: 8

DATE: 26/08/2025

INSTRUCTOR: ORANGZAIB RAJPOOT

# THE ART OF REUSABILITY: BUILDING WITH FUNCTIONS

**Welcome back!** Today, we transition from writing simple scripts to creating structured, professional code. Functions are the fundamental building blocks that allow us to name a piece of logic, package it up, and reuse it. Mastering them is the key to writing clean, efficient, and scalable programs.

# TODAY'S AGENDA

- **The "What" and "Why" of Functions**
  - The DRY Principle: Don't Repeat Yourself
  - Defining a Function: The def Statement
  - The pass Statement: A Placeholder for Future Code
  - Python Function Naming Conventions
  - Parameters vs. Arguments
  - The return Statement: Getting a Value Back
  - Interactive Exercises: Simple Calculators and Greeters
- **Flexible Arguments & Advanced Returns**
  - Positional vs. Keyword Arguments & Default Values
  - Arbitrary Arguments: *args and **kwargs
  - Advanced Returns: return vs. yield (Introducing Generators)
  - *Interactive Exercises: Flexible Pizza Order & Simple Generators*

# AGENDA CONT.

- **Structuring Python Scripts & The Lab**
  - Variable Scope: Local vs. Global
  - Docstrings: Documenting Your Functions
  - The main function pattern in Python
  - Making Scripts Executable: if __name__ == "__main__"
  - Handling Command-Line Arguments with sys.argv
  - **Hands-On Lab: Refactoring Previous Exercises**

# THE "WHAT" AND "WHY" OF FUNCTIONS

## THE DRY PRINCIPLE: DON'T REPEAT YOURSELF

In programming, this is a core philosophy. If you find yourself writing the same block of code in multiple places, it's a sign that you need a function.

- **Why is repeating code bad?**
  - **It's error-prone:** If you need to fix a bug, you have to find and fix it in every single place you copied it.
  - **It's hard to read:** A long script is harder to understand than a script that is broken down into logical, named chunks.
  - **It's hard to maintain:** If you want to change how something works, you have to update it everywhere.
- Functions solve this by providing a single source of truth for a piece of logic.

# DEFINING A FUNCTION: THE DEF STATEMENT

- A function is a named block of code that performs a specific task. You define it using the def keyword. The code block inside the function must be indented.

```
def print_welcome_message():
    print("--------------------")
    print("Welcome to the App!")
    print("--------------------")
```

# THE `PASS` STATEMENT: A PLACEHOLDER FOR FUTURE CODE

- Python's syntax requires an indented block after a statement like `def`. But what if you want to define a function now but write its logic later? An empty block will cause an `IndentationError`.
- The `pass` statement is a null operation—nothing happens when it executes. It's used as a placeholder where code is syntactically required, but you have nothing to write yet.

```
• def fetch_user_data_from_db():
•     # TODO: Add database connection and query logic later
•     pass # This is a valid, empty function

• def validate_input():
•     # I'll implement this after fetching data
•     pass

• print("Planning out my program structure...")
• fetch_user_data_from_db() # This runs without error
• print("Structure is valid.")
```

# PYTHON FUNCTION NAMING CONVENTIONS

- Just like variables, functions have a standard naming convention that improves readability.
  - **snake_case:** Use all lowercase letters, with words separated by underscores.
  - **Verb-Noun:** Names should be descriptive and often follow a verb-noun pattern.

```
# Good, Pythonic function names

def calculate_average(numbers): ...

def send_email(recipient, subject): ...

def validate_user_input(): ...
```

# PARAMETERS VS. ARGUMENTS

- **Parameter:** The variable name inside the function's parentheses. It's a placeholder for the data the function expects to receive.
- **Argument:** The actual value you pass to the function when you call it.

```
# 'name' is the parameter
def greet_user(name):
    print(f"Hello, {name}!")

# "Alice" is the argument
greet_user("Alice")
```

# THE `RETURN` STATEMENT: GETTING A VALUE BACK

- Many functions don't just `print` something; they perform a calculation and need to give the result back to the main program. The `return` statement does this. When `return` is hit, the function immediately stops and sends the value back.

```
def add_numbers(a, b):
    result = a + b
    return result

sum_of_numbers = add_numbers(5, 10)
print(f"The sum is: {sum_of_numbers}")
```

# IN-CLASS EXERCISE: AREA CALCULATOR FUNCTION

- Write a function called calculate_area that takes two parameters: width and height.
- Inside the function, it should calculate the area (width * height).
- The function should return the calculated area.
- Call the function with some arguments and print the returned result.

# FLEXIBLE AND POWERFUL ARGUMENTS

## POSITIONAL VS. KEYWORD ARGUMENTS

- **Positional:** The arguments are matched to parameters based on their order.
- **Keyword:** You explicitly name which parameter you are providing a value for. This frees you from the order.

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

# Positional arguments (order matters)
describe_pet("hamster", "Harry")

# Keyword arguments (order does NOT matter)
describe_pet(pet_name="Willow", animal_type="dog")
```

# DEFAULT PARAMETER VALUES

- You can provide a default value for a parameter. If an argument for that parameter is not provided when the function is called, the default value is used.

```
# animal_type now has a default value
def describe_pet(pet_name, animal_type="dog"):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet("Willow") # Uses the default "dog"
describe_pet("Goldie", animal_type="fish") # Overrides the default
```

# ARBITRARY ARGUMENTS: *ARGS AND **KWARGS

- What if you don't know how many arguments a function will receive?
  - *args: Gathers any number of **positional** arguments into a **tuple**.
  - **kwargs: Gathers any number of **keyword** arguments into a **dictionary**.

```
# *args example: a function that can sum any number of values
def sum_all(*numbers):
    total = 0
    for num in numbers: # 'numbers' is a tuple
        total += num
    return total

print(sum_all(1, 2, 3)) # -> 6
print(sum_all(10, 20, 30, 40)) # -> 100
```

```
# **kwargs example: a function that builds a user profile
def build_profile(first, last, **user_info):
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

profile = build_profile('albert', 'einstein',
                        location='princeton',
                        field='physics')
print(profile)
# {'location': 'princeton', 'field': 'physics', 'first_name': 'albert', 'last_name': 'einstein'}
```

# IN-CLASS EXERCISE: PIZZA ORDER

- Write a function make_pizza(size, *toppings).
- It should print a summary of the pizza being ordered, like:
  "Making a 12-inch pizza with the following toppings:"
  Then, it should loop through the toppings tuple and print each one.
  Call the function a couple of times with different sizes and numbers of toppings.

# ADVANCED RETURNS: RETURN VS. YIELD

- The return statement terminates a function and sends back a single value. The yield keyword turns a function into a **generator**, which produces a sequence of values over time without storing them all in memory.
- **Key Takeaway:** Use yield when you are working with a potentially huge sequence of data and want to be memory-efficient.

```
# A generator function that yields values one by one
def get_even_numbers_generator(limit):
    for i in range(limit):
        if i % 2 == 0:
            yield i # Pauses here, returns i, and waits

# You can loop over it just like a list
for number in get_even_numbers_generator(10):
    print(number) # Prints 0, 2, 4, 6, 8
```

# IN-CLASS EXERCISE: COUNTDOWN GENERATOR

- Write a generator function countdown(start) that takes a number and yields each number from start down to 1. Then, use a for loop to print the countdown from 5.

# SCOPE, BEST PRACTICES & THE HANDS-ON LAB

## VARIABLE SCOPE: LOCAL VS. GLOBAL

- **Global Scope:** A variable defined in the main body of a Python file. It can be accessed anywhere in the file.
- **Local Scope:** A variable defined inside a function. It can only be accessed *within that function.*

- It's best practice to avoid modifying global variables from within functions. Pass data in as parameters and return results.

```
global_var = "I am global" # Global scope

def my_function():
    local_var = "I am local" # Local scope
    print(global_var) # Can access global variables
    print(local_var)

my_function()
# print(local_var) # This would cause a NameError!
```

# DOCSTRINGS: DOCUMENTING YOUR FUNCTIONS

- A docstring is a string literal that occurs as the first statement in a function definition. It explains what the function does. It's a crucial part of writing professional, reusable code.

```
def calculate_average(numbers):
    """Calculates the average of a list of numbers.


    Args:
        numbers (list): A list of numbers (integers or floats).


    Returns:
        float: The average of the numbers, or 0 if the list is empty.
    """
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)


# You can now get help on your own function!
help(calculate_average)
```

# THE MAIN FUNCTION PATTERN

- In many other languages, execution starts in a main function. Python doesn't require this, but it's a very strong convention. It organizes your code, making it clear where the primary logic of your script begins.

```
def main():
    """The main entry point for the script."""
    print("Starting the program...")
    # ... call other functions ...
    print("Program finished.")

# --- Script execution starts here ---
main()
```

# MAKING SCRIPTS EXECUTABLE: IF __NAME__ == "__MAIN__"

- This is one of the most important idioms in Python. It allows a Python file to be used in two ways: as a standalone script or as an importable module. The code inside this block **only** runs when the file is executed directly.

```python
def reusable_function():
    return "This can be imported."


def main():
    print("This script is being run directly.")
    print(reusable_function())


# This check ensures main() is only called when we run `python my_script.py`
if __name__ == "__main__":
    main()
```

# HANDLING COMMAND-LINE ARGUMENTS WITH SYS.ARGV

Python's sys module lets your script accept arguments from the command line. sys.argv is a **list** of strings containing these arguments.

- sys.argv[0] is always the name of the script itself.
- sys.argv[1] is the first argument, sys.argv[2] is the second, and so on

```
# Save as greet.py
import sys

def main():
    if len(sys.argv) > 1:
        name = sys.argv[1] # Get the first argument
        print(f"Hello, {name}!")
    else:
        print("Hello, world!")

if __name__ == "__main__":
    main()

# In your terminal:
# > python greet.py Alice
# Output: Hello, Alice!
```

# HANDS-ON LAB: REFACTOR PREVIOUS EXERCISES

## PART 1: REFACTOR THE CALCULATOR

- Open a new file, functional_calculator.py.
- Create separate functions for add, subtract, multiply, and divide.
- Create a main() function. Inside main, check sys.argv.
- Your script should expect 3 arguments: num1, operation, num2.
- Example: python functional_calculator.py 10 + 20
- In main, parse these arguments, convert the numbers to floats, and call the correct function. Print the result.
- Add error handling for the wrong number of arguments or invalid operations.
- Wrap your main() call in an if __name__ == "__main__" block.

# PART 2: REFACTOR THE TO-DO LIST

- Open a new file, functional_todo.py.
- Create functions for display_tasks, add_task, and remove_task.
- Create a main() function that contains the main while loop for the user menu.
- Wrap the call to main() in an if __name__ == "__main__" block.

# CHALLENGE:

- Modify your to-do list to accept command-line arguments.
  - python functional_todo.py add "Buy milk" should add the task without showing the menu.
  - python functional_todo.py list should just display the tasks and exit.
  - python functional_todo.py (with no arguments) should run the interactive menu loop. This will require you to put if/elif/else logic inside your main() function to check the contents of sys.argv.