Module 4 - Functions & Modules in Python

Welcome to Module 4! So far, you've learned how to make your programs smart with conditional statements and efficient with loops. Now, we're going to unlock the power of **functions** and **modules**, which are essential for writing organized, reusable, and professional Python code. We'll also explore special topics like **recursion** and **variable scope**.

Chapter 1: Functions - Building Reusable Blocks

Functions are blocks of organized, reusable code that perform a single, related action. They are one of the most fundamental concepts in programming.

1.1 Introduction: Why Use Functions?

Imagine you're building a house. Instead of building every brick from scratch for every part of the house, you use pre-made bricks, window frames, doors, etc. Functions are like these "pre-made components" for your code.

Here's why functions are incredibly important:

1. **Code Reusability (DRY Principle):** DRY stands for "Don't Repeat Yourself." If you find yourself writing the same lines of code multiple times, it's a strong sign that you should put that code into a function. Once defined, you can call (use) the function as many times as you need.
2. **Organization and Readability:** Functions break down large, complex problems into smaller, manageable pieces. This makes your code much easier to read, understand, and navigate. Each function has a clear purpose.
3. **Maintainability:** If you need to change a piece of logic, you only need to change it in one place (inside the function definition), rather than finding and changing it everywhere it's duplicated. This drastically reduces bugs and effort.
4. **Easier Debugging:** When an error occurs, it's often easier to pinpoint the problem within a small, focused function rather than sifting through hundreds of lines of linear code.
5. **Abstraction:** Functions allow you to hide the complex details of how something works. You just need to know *what* the function does, not *how* it does it. For example, when you use `print()`, you don't care how it writes to the screen, just that it does.

1.2 Defining and Calling a Function

To use a function, you first need to **define** it (tell Python what it does), and then you **call** it (make it run).

Syntax for Defining a Function:

Python

```
def function_name(parameters): # 'def' keyword, function_name, parentheses,
colon
    """
    Docstring: A brief explanation of what the function does. (Optional but
highly recommended)
    """
    # Indented block of code that the function performs
    # (This is the function's "body")
    pass # 'pass' is a placeholder if the function body is empty for now
```

- `def`: This keyword tells Python you are about to define a function.
- `function_name`: This is how you will refer to your function. Choose a meaningful name (lowercase with underscores _ is standard).
- `()`: Parentheses are mandatory, even if the function takes no inputs. They can contain `parameters` (inputs) later.
- `:`: A colon marks the end of the `def` statement header.
- **Indentation:** The code that belongs to the function *must* be indented (usually 4 spaces). This defines the function's body.
- `"""Docstring"""`: A string literal on the first line inside the function body is called a docstring. It provides documentation for your function and is accessible via `help(function_name)`.

Calling a Function:

Once defined, you execute a function by using its name followed by parentheses:

Python
```
function_name() # If it takes no arguments
function_name(argument1, argument2) # If it takes arguments
```

Example 1: A Simple Greeting Function (Console)

Python
```
>>> def greet(): # Function definition
...     print("Hello, welcome to Python!")
...     print("Nice to see you!")
...
>>> greet() # Function call
Hello, welcome to Python!
Nice to see you!
>>> greet() # Call it again!
Hello, welcome to Python!
Nice to see you!
>>>
```

Example 2: Displaying a Separator (Script)

Create a file named `my_program.py`:

Python
```
# my_program.py

def print_separator():
    """Prints a line of dashes to separate sections."""
    print("--------------------")
```

```
print("Start of program")
print_separator() # Call the function
print("Middle section")
print_separator() # Call it again
print("End of program")
```

Interaction:

```
Start of program
--------------------
Middle section
--------------------
End of program
```

Common Function Definition Errors:

- **Missing Colon:** `SyntaxError: expected ':'`
- **Incorrect Indentation:** `IndentationError: expected an indented block`
- **Calling before Defining:** You must define a function *before* you can call it.

  Python

  ```
  >>> say_hi() # This will cause an error
  NameError: name 'say_hi' is not defined
  >>> def say_hi():
  ...     print("Hi!")
  ...
  >>> say_hi() # Now it works
  Hi!
  ```

1.3 The `return` Statement

Functions can not only perform actions but also **produce results** that can be used by other parts of your program. The `return` statement is used to send a value back from a function to the place where it was called.

How `return` Works:

1. **Returns a Value:** When `return value` is executed, the `value` is sent back to the caller.
2. **Terminates Function Execution:** As soon as Python encounters a `return` statement, the function immediately stops executing, and control is given back to the calling code. Any code after `return` in that function will *not* be executed.
3. **Implicit `None`:** If a function doesn't have an explicit `return` statement, it implicitly returns `None`. `None` is a special Python value that represents the absence of a value.

Syntax:

Python
```
def function_name():
    # ... some code ...
    return value # Returns 'value' to the caller
```

Example 1: Adding Two Numbers (Console)

Python
```
>>> def add_numbers(a, b):
...     """Adds two numbers and returns their sum."""
...     sum_result = a + b
...     return sum_result # Return the calculated sum
...
>>> result = add_numbers(5, 3) # Call function, store returned value in
'result'
>>> print(result)
8
>>> print(add_numbers(10, 20)) # Print the returned value directly
30
>>>
```

Example 2: Function with No Explicit Return (Script)

Create `return_example.py`:

Python
```
# return_example.py

def greeting(name):
    """Prints a greeting but doesn't explicitly return anything."""
    print(f"Hello, {name}!")

def calculate_square(number):
    """Calculates the square of a number and returns it."""
    return number * number

# Call greeting - its return value will be None
return_value_from_greeting = greeting("Alice")
print(f"Return value from greeting: {return_value_from_greeting}") # This
will be None

print("-" * 20)

# Call calculate_square - its return value will be the calculated square
square_of_7 = calculate_square(7)
print(f"The square of 7 is: {square_of_7}")

another_square = calculate_square(12)
print(f"The square of 12 is: {another_square}")
```

Interaction:

```
Hello, Alice!
Return value from greeting: None
--------------------
The square of 7 is: 49
The square of 12 is: 144
```

Key Observation: Notice how `greeting("Alice")` prints "Hello, Alice!", but when you assign its return value to `return_value_from_greeting`, it's `None`. This is because `greeting` doesn't have an explicit `return` statement.

Returning Multiple Values:

Python functions can conveniently return multiple values. They are automatically packaged into a `tuple` (an ordered, immutable collection of items).

Python

```
>>> def get_min_max(numbers):
...     """Finds the minimum and maximum numbers in a list."""
...     return min(numbers), max(numbers) # Returns two values
...
>>> my_list = [10, 5, 20, 3, 15]
>>> minimum, maximum = get_min_max(my_list) # Unpack the returned tuple
>>> print(f"Min: {minimum}, Max: {maximum}")
Min: 3, Max: 20
>>>
```

1.4 Passing Arguments to a Function

To make functions more versatile, you can pass information into them. This information is called **arguments**, and the function receives them as **parameters**.

Parameters vs. Arguments:

- **Parameters:** These are the names listed in the function definition's parentheses (`def function_name(parameter1, parameter2):`). They act as placeholders for the values the function will receive.
- **Arguments:** These are the actual values you pass to the function when you call it (`function_name(argument1, argument2)`).

Types of Arguments:

1. **Positional Arguments (Required):**
    - The most common type. The order in which you pass arguments matters. Python matches them to parameters by their position.
    - If you define a function with parameters, you *must* provide corresponding arguments when calling it.

    Python

    ```
    >>> def describe_pet(animal_type, pet_name):
    ...     print(f"I have a {animal_type}.")
    ...     print(f"Its name is {pet_name}.")
    ...
    >>> describe_pet("dog", "Buddy") # Order matters: "dog" goes to
    animal_type, "Buddy" to pet_name
    I have a dog.
    Its name is Buddy.

    >>> describe_pet("Mittens", "cat") # Incorrect order, leads to
    logical error
    I have a Mittens.
    Its name is cat.

    >>> describe_pet("cat") # Missing an argument, causes TypeError
    ```

```
TypeError: describe_pet() missing 1 required positional argument:
'pet_name'
```

2. **Keyword Arguments (Optional, for Clarity):**
    - You can specify the parameter name along with the value when calling the function.
    - This makes the call clearer and the order doesn't matter (though it's good practice to stick to an order for readability).

    Python

```
>>> describe_pet(animal_type="hamster", pet_name="Harry")
I have a hamster.
Its name is Harry.

>>> describe_pet(pet_name="Fido", animal_type="dog") # Order doesn't
matter with keywords
I have a dog.
Its name is Fido.
```

3. **Default Arguments (Optional):**
    - You can provide a default value for a parameter in the function definition.
    - If an argument is provided for that parameter during the call, the default is overridden.
    - If no argument is provided, the default value is used.
    - **Important:** Parameters with default values *must* come after any non-default (positional) parameters.

    Python

```
>>> def greet_person(name="Guest"): # 'Guest' is the default value
...     print(f"Hello, {name}!")
...
>>> greet_person("Sarah") # Argument provided, default overridden
Hello, Sarah!

>>> greet_person() # No argument provided, default used
Hello, Guest!

>>> def power(base, exponent=2): # exponent has a default of 2
...     return base ** exponent
...
>>> print(power(3))    # 3 to the power of 2 (default exponent)
9
>>> print(power(3, 4)) # 3 to the power of 4 (exponent overridden)
81
```

1.5 Passing Functions as Arguments (Higher-Order Functions - Brief Look)

This is a more advanced concept, but it's important to know that in Python, functions are "first-class citizens." This means you can treat functions like any other variable: you can assign them to variables, store them in data structures, and most importantly for this topic, **pass them as arguments to other functions**.

Functions that take other functions as arguments are called **higher-order functions**.

Example: A Generic Operation Function (Script)

Let's create a function `do_math_operation` that takes two numbers and an *operation function* as arguments.

Python
```
# higher_order_example.py

# Define simple operation functions
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

# A higher-order function that takes another function as an argument
def do_math_operation(num1, num2, operation_func):
    """
    Performs a math operation using the provided function.
    num1: The first number.
    num2: The second number.
    operation_func: A function that takes two arguments and returns a
result.
    """
    print(f"Performing operation with {operation_func.__name__}...")
    result = operation_func(num1, num2) # Call the passed-in function
    return result

# --- Using the higher-order function ---

# Pass the 'add' function itself (without parentheses)
sum_result = do_math_operation(10, 5, add)
print(f"Sum: {sum_result}")

# Pass the 'subtract' function
diff_result = do_math_operation(10, 5, subtract)
print(f"Difference: {diff_result}")

# Pass the 'multiply' function
prod_result = do_math_operation(10, 5, multiply)
print(f"Product: {prod_result}")
```

Interaction:

```
Performing operation with add...
Sum: 15
Performing operation with subtract...
Difference: 5
Performing operation with multiply...
Product: 50
```

Explanation: Notice how `add`, `subtract`, and `multiply` are passed without `()`. This means we are passing the *function object itself*, not the *result* of calling the function. The

`do_math_operation` function then calls this passed-in function (`operation_func(num1, num2)`). This is a powerful concept for creating flexible and abstract code!

Chapter 2: Variables - Scope and Lifecycle

Understanding where a variable exists and can be accessed in your program is crucial. This concept is called **variable scope**.

2.1 Local Variables

Definition: A variable defined *inside* a function is a **local variable**.

Scope: Local variables can *only* be accessed (read or modified) from within the function where they are defined. They are "local" to that function.

Lifecycle:

- A local variable is **created** when the function is called and begins execution.
- It **exists** only for the duration of that function call.
- It is **destroyed** (memory released) when the function finishes executing (either by `return` or by reaching the end of its code).

Example: Demonstrating Local Scope (Console)

Python
```
>>> def my_function():
...     message = "This is a local message." # 'message' is a local
variable
...     print(message)
...
>>> my_function()
This is a local message.
>>> print(message) # Trying to access 'message' outside the function
NameError: name 'message' is not defined
>>>
```

Explanation: When `my_function()` runs, `message` is created and printed. As soon as `my_function()` finishes, `message` ceases to exist, so `print(message)` outside causes a `NameError`.

2.2 Global Variables

Definition: A variable defined *outside* of any function (at the top level of your script) is a **global variable**.

Scope: Global variables can be accessed (read) from *anywhere* in the script, including inside functions.

Lifecycle:

- A global variable is created when the script starts executing.
- It exists until the script finishes execution.

Example 1: Reading a Global Variable (Console)

Python
```
>>> global_var = "I am global!" # Defined outside any function
>>> def print_global():
...     print(global_var) # Can read global_var
...
>>> print_global()
I am global!
>>> print(global_var)
I am global!
>>>
```

Example 2: Modifying a Global Variable (The `global` keyword)

By default, if you try to *assign a new value* to a variable inside a function, Python assumes you are creating a *new local variable* with that name, even if a global variable with the same name exists. This can be confusing.

To explicitly tell Python that you want to *modify an existing global variable* from within a function, you must use the `global` keyword.

Python
```
>>> counter = 0 # Global variable

>>> def increment_counter_local():
...     counter = 100 # This creates a NEW LOCAL variable named 'counter'
...     print(f"Inside function (local): {counter}")
...
>>> def increment_counter_global():
...     global counter # Declare intent to modify the GLOBAL 'counter'
...     counter += 1    # This modifies the global 'counter'
...     print(f"Inside function (global): {counter}")
...
>>> print(f"Before calls: {counter}")
Before calls: 0

>>> increment_counter_local()
Inside function (local): 100
>>> print(f"After local call: {counter}") # Global 'counter' is unchanged!
After local call: 0

>>> increment_counter_global()
Inside function (global): 1
>>> print(f"After global call: {counter}") # Global 'counter' has changed!
After global call: 1
>>>
```

Warning: While the `global` keyword allows you to modify global variables, it's generally considered **bad practice** to rely heavily on global variables for communication between functions. It makes code harder to understand, test, and debug because functions become dependent on external state that can change unpredictably.

2.3 Best Practices for Variable Scope

- **Favor Local Variables:** Design your functions to be self-contained. Pass data *into* functions using arguments and get data *out* using `return` statements.
- **Minimize `global` Keyword Usage:** Use the `global` keyword sparingly, only when absolutely necessary (e.g., for very simple, top-level configurations or single-instance objects that genuinely need global access). For more complex scenarios, consider passing objects around or using classes (a more advanced topic).

Chapter 3: Modules - Organizing Your Code

As your programs grow larger, putting all your code in a single file becomes unmanageable. **Modules** provide a way to organize your Python code into separate, reusable files.

3.1 What are Modules?

- A **module** in Python is simply a file containing Python code (functions, variables, classes, etc.) that has a `.py` extension.
- The name of the module is the filename itself (without the `.py`). For example, a file named `my_functions.py` is a module named `my_functions`.
- **Purpose:**
  - **Organization:** Break down large programs into smaller, logical, and manageable files.
  - **Reusability:** Code defined in one module can be imported and used in other Python scripts or interactive sessions.
  - **Namespace Management:** Modules help avoid name conflicts by providing their own "namespace." (e.g., `math.pi` vs `my_constants.pi`).

3.2 Importing Modules

You use the `import` statement to bring code from one module into another.

Syntax Variations:

1. **`import module_name`**:
   - Imports the entire module.
   - You must use `module_name.item` to access anything defined within that module.

   Python

   ```
   # Example using built-in 'math' module
   import math
   print(math.pi)
   print(math.sqrt(25))
   ```

2. **`from module_name import item1, item2`**:

- Imports specific items (functions, variables, etc.) directly into your current namespace.
- You can then use `item1` or `item2` directly without the `module_name.` prefix.

Python

```
# Example using built-in 'math' module
from math import pi, sqrt
print(pi)
print(sqrt(36))
```

3. **`import module_name as alias`**:
   - Imports the entire module but gives it a shorter, more convenient alias.
   - Useful for long module names or to prevent name clashes.

   Python

   ```
   # Example using built-in 'random' module, commonly aliased as 'rnd'
   import random as rnd
   print(rnd.randint(1, 10)) # Generates a random integer between 1 and
   10 (inclusive)
   ```

Example using `random` module (Script):

Python

```
# dice_roller.py
import random # Import the random module

def roll_dice(sides):
    """Simulates rolling a dice with given number of sides."""
    return random.randint(1, sides) # Use random.randint() from the
imported module

print("Rolling a 6-sided die:", roll_dice(6))
print("Rolling a 20-sided die:", roll_dice(20))
```

3.3 Creating Your Own Modules

Creating your own module is as simple as saving your Python code in a `.py` file.

Steps:

1. **Create a module file:** Save your functions, variables, etc., in a `.py` file.
2. **Import it:** In another Python script (in the same directory or a location Python knows about), use `import` to bring in your module.

Example: Simple Math Module

Step 1: Create `my_calculator.py`

Python

```
# my_calculator.py - This is our custom module
```

```
def add(a, b):
    """Adds two numbers."""
    return a + b

def subtract(a, b):
    """Subtracts two numbers."""
    return a - b

PI = 3.14159 # A constant variable in our module
```

Step 2: Create `main_program.py` in the SAME directory

Python
```
# main_program.py - This script will use our custom module

import my_calculator # Import the module we just created

num1 = 15
num2 = 7

sum_result = my_calculator.add(num1, num2)
diff_result = my_calculator.subtract(num1, num2)

print(f"{num1} + {num2} = {sum_result}")
print(f"{num1} - {num2} = {diff_result}")

print(f"Pi from my_calculator module: {my_calculator.PI}")

# You could also do:
from my_calculator import add, PI
print(f"Using direct add: {add(5, 3)}")
print(f"Direct PI: {PI}")
```

Interaction:

```
15 + 7 = 22
15 - 7 = 8
Pi from my_calculator module: 3.14159
Using direct add: 8
Direct PI: 3.14159
```

This demonstrates how you can organize your code into separate, logical files and easily reuse functions and variables across your projects.

Chapter 4: Recursion - Functions Calling Themselves

Recursion is a powerful programming technique where a function calls itself, directly or indirectly, to solve a problem. It's like looking up a word in a dictionary, and the definition refers you to another word, which then refers you back to the first word!

4.1 Introduction to Recursion

- **Definition:** A recursive function is a function that calls itself during its execution.

- **Analogy:** Imagine a set of Russian nesting dolls, where each doll contains a smaller version of itself until you reach the smallest one. Or two mirrors facing each other, creating an infinite reflection.
- **Core Idea:** Recursion solves a problem by breaking it down into smaller, similar subproblems. It solves the simplest version of the problem directly (the **base case**), and for more complex versions, it calls itself to solve a slightly simpler version, eventually reaching the base case.

Every recursive function must have two main parts:

1. **Base Case:**
    - This is the condition that tells the function when to stop recursing.
    - It's the simplest version of the problem that can be solved directly, without further recursive calls.
    - **Crucial:** Without a base case, the function would call itself infinitely, leading to a "RecursionError: maximum recursion depth exceeded."
2. **Recursive Step:**
    - The part where the function calls itself.
    - The problem is typically broken down into a smaller, simpler subproblem, and the recursive call works on this simplified version.
    - The results from the recursive calls are then combined to solve the original problem.

4.2 Recursion Example 1: Simple Countdown

Let's make a function that counts down from a given number to 1 using recursion.

Python
```
def countdown(n):
    """
    Recursively counts down from n to 1.
    """
    if n <= 0: # Base Case: If n is 0 or less, stop
        print("Blast off!")
    else:      # Recursive Step: If n is greater than 0
        print(n)
        countdown(n - 1) # Call itself with a smaller number (n-1)

# Test the function
print("--- Countdown from 3 ---")
countdown(3)

print("\n--- Countdown from 0 ---")
countdown(0)

print("\n--- Countdown from 5 ---")
countdown(5)
```

Interaction:

```
--- Countdown from 3 ---
3
2
1
```

```
Blast off!

--- Countdown from 0 ---
Blast off!

--- Countdown from 5 ---
5
4
3
2
1
Blast off!
```

Trace of `countdown(3)`:

1. `countdown(3)`:
   - Is `3 <= 0`? No.
   - `print(3)`
   - Calls `countdown(2)`
2. `countdown(2)`:
   - Is `2 <= 0`? No.
   - `print(2)`
   - Calls `countdown(1)`
3. `countdown(1)`:
   - Is `1 <= 0`? No.
   - `print(1)`
   - Calls `countdown(0)`
4. `countdown(0)`:
   - Is `0 <= 0`? Yes. (Base Case reached!)
   - `print("Blast off!")`
   - Returns to `countdown(1)` (which then finishes)
5. `countdown(1)` finishes.
6. `countdown(2)` finishes.
7. `countdown(3)` finishes.

4.3 Recursive Function: Factorial Calculation

The factorial of a non-negative integer `n`, denoted `n!`, is the product of all positive integers less than or equal to `n`.

- `5! = 5 * 4 * 3 * 2 * 1 = 120`
- By definition, `0! = 1`.

The recursive definition is:

- `n! = n * (n-1)!` (for `n > 0`)
- `0! = 1` (Base Case)

Python

```
def factorial(n):
    """
    Calculates the factorial of a non-negative integer recursively.
    """
```

```
    if n == 0: # Base Case: Factorial of 0 is 1
        return 1
    else:      # Recursive Step: n! = n * (n-1)!
        return n * factorial(n - 1) # Calls itself with a smaller number

# Test the function
print(f"Factorial of 0: {factorial(0)}")
print(f"Factorial of 1: {factorial(1)}")
print(f"Factorial of 5: {factorial(5)}") # Expected: 120
print(f"Factorial of 7: {factorial(7)}") # Expected: 5040
```

Interaction:

```
Factorial of 0: 1
Factorial of 1: 1
Factorial of 5: 120
Factorial of 7: 5040
```

Trace of `factorial(3)`:

1. `factorial(3)`:
    - Is `3 == 0`? No.
    - Returns `3 * factorial(2)`
2. `factorial(2)`:
    - Is `2 == 0`? No.
    - Returns `2 * factorial(1)`
3. `factorial(1)`:
    - Is `1 == 0`? No.
    - Returns `1 * factorial(0)`
4. `factorial(0)`:
    - Is `0 == 0`? Yes. (Base Case!)
    - Returns `1`
    - Back to `factorial(1)`
5. `factorial(1)` now has `1 * 1 = 1`. Returns `1`.
    - Back to `factorial(2)`
6. `factorial(2)` now has `2 * 1 = 2`. Returns `2`.
    - Back to `factorial(3)`
7. `factorial(3)` now has `3 * 2 = 6`. Returns `6`.

Final Result: 6

4.4 When to Use Recursion (and When Not To)

- **Pros of Recursion:**
    - **Elegance and Readability:** For certain problems (especially those with a naturally recursive definition, like factorial, Fibonacci sequence, or tree traversals), recursive solutions can be very elegant and easier to read and understand than their iterative counterparts.
    - **Problem Simplification:** It allows you to solve complex problems by breaking them into simpler, self-similar subproblems.
- **Cons of Recursion:**

- **Performance Overhead:** Each function call adds overhead (memory for stack frames, time for setup/teardown). For very deep recursion, this can be slower and consume more memory than an iterative solution.
- **Stack Overflow:** Python has a default recursion limit (usually around 1000-3000 calls). If your recursion goes too deep without hitting a base case, you'll get a `RecursionError`.
- **Hard to Debug:** Tracing recursive calls can sometimes be more challenging than tracing simple loops.

In Python, for simple looping tasks (like counting or summing a range of numbers), an iterative solution (using `for` or `while` loops) is generally preferred due to its efficiency and avoidance of the recursion limit. Recursion is best used when it naturally maps to the problem's structure and makes the code clearer.

Chapter 5: Coding Challenges for Functions & Modules

Now, let's put your understanding of functions, scope, modules, and recursion into practice!

Challenge 1: Enhanced Temperature Converter (Functions)

Goal: Refactor your previous Celsius/Fahrenheit converter into well-defined functions.

Concepts Covered: `def`, `return`, passing arguments.

Requirements:

1. Create a function `celsius_to_fahrenheit(celsius)` that:
   - Takes a `celsius` temperature as an argument.
   - Calculates Fahrenheit using the formula: `Fahrenheit = (Celsius * 9/5) + 32`.
   - **Returns** the Fahrenheit temperature.
2. Create a function `fahrenheit_to_celsius(fahrenheit)` that:
   - Takes a `fahrenheit` temperature as an argument.
   - Calculates Celsius using the formula: `Celsius = (Fahrenheit - 32) * 5/9`.
   - **Returns** the Celsius temperature.
3. In the main part of your script:
   - Ask the user for a Celsius temperature and print its Fahrenheit equivalent using your function.
   - Ask the user for a Fahrenheit temperature and print its Celsius equivalent using your function.

Example Interaction:

```
Enter Celsius temperature: 25
25.0°C is 77.0°F

Enter Fahrenheit temperature: 68
```

```
68.0°F is 20.0°C
```

Challenge 2: Basic Geometry Calculator (Functions with Options)

Goal: Create functions to calculate the area of a rectangle and a circle, and allow the user to choose which calculation to perform.

Concepts Covered: `def`, `return`, passing arguments, `if-elif-else`, `input()`, `math` module.

Requirements:

1. **Import the `math` module** for `math.pi`.
2. Create a function `calculate_rectangle_area(length, width)` that:
    - Takes `length` and `width` as arguments.
    - Calculates `area = length * width`.
    - **Returns** the area.
3. Create a function `calculate_circle_area(radius)` that:
    - Takes `radius` as an argument.
    - Calculates `area = math.pi * radius**2`.
    - **Returns** the area.
4. In the main part of your script:
    - Present a menu: "1. Calculate Rectangle Area", "2. Calculate Circle Area".
    - Get the user's choice.
    - Use `if-elif-else` to call the appropriate function, get the required inputs from the user, and print the result. Handle invalid choices.

Example Interaction:

```
Geometry Calculator:
1. Calculate Rectangle Area
2. Calculate Circle Area
Enter your choice (1 or 2): 1
Enter length: 5
Enter width: 8
The area of the rectangle is: 40.0

Geometry Calculator:
1. Calculate Rectangle Area
2. Calculate Circle Area
Enter your choice (1 or 2): 2
Enter radius: 3
The area of the circle is: 28.274333882308138

Geometry Calculator:
1. Calculate Rectangle Area
2. Calculate Circle Area
Enter your choice (1 or 2): 3
Invalid choice.
```

Challenge 3: Custom String Utilities Module

Goal: Create your own Python module containing useful string manipulation functions, and then use it in another script.

Concepts Covered: Creating modules, `import`, functions.

Requirements:

Part A: Create `string_utils.py`

1. Create a new file named `string_utils.py`.
2. Inside `string_utils.py`, define the following functions:
    - `reverse_string(s)`: Takes a string `s` and **returns** its reverse. (Hint: `s[::-1]` is a Pythonic way to reverse a string).
    - `is_palindrome(s)`: Takes a string `s`, checks if it's the same forwards and backward (ignoring case), and **returns** `True` or `False`. (Hint: use `reverse_string` and `.lower()`).
    - `count_words(s)`: Takes a string `s` and **returns** the number of words in it. (Hint: `s.split()` splits a string into a list of words).

Part B: Create `main_string_app.py`

1. Create another file named `main_string_app.py` in the **same directory** as `string_utils.py`.
2. In `main_string_app.py`, `import string_utils`.
3. Use the functions from `string_utils` to:
    - Get a phrase from the user.
    - Print its reverse using `string_utils.reverse_string()`.
    - Print whether it's a palindrome or not using `string_utils.is_palindrome()`.
    - Print the number of words in it using `string_utils.count_words()`.

Example Interaction:

```
# When running main_string_app.py
Enter a phrase: Hello Python
Reversed: nohtyP olleH
Is Palindrome: False
Word count: 2

Enter a phrase: Madam
Reversed: madaM
Is Palindrome: True
Word count: 1
```

Challenge 4: Recursive Power Calculation

Goal: Write a recursive function to calculate the power of a number (base raised to exponent).

Concepts Covered: Recursion, base case, recursive step.

Requirements:

1. Create a function `power(base, exponent)` that:
    - Takes two arguments: `base` and `exponent`.
    - **Base Case:** If `exponent` is `0`, **return `1`** (any number to the power of 0 is 1).
    - **Recursive Step:** If `exponent` is greater than 0, **return `base * power(base, exponent - 1)`**.
2. Test your function with various inputs (e.g., `power(2, 3)`, `power(5, 0)`, `power(4, 2)`).

Example Interaction:

```
2 to the power of 3 is: 8
5 to the power of 0 is: 1
4 to the power of 2 is: 16
```

Challenge 5: Global Counter (Careful Usage)

Goal: Create a simple function that increments a global counter each time it's called.

Concepts Covered: Global variables, `global` keyword.

Requirements:

1. Define a global variable `call_count` and initialize it to `0`.
2. Create a function `track_calls()` that:
    - Uses the `global` keyword to indicate it will modify the global `call_count`.
    - Increments `call_count` by `1`.
    - Prints a message like "Function called X times." where X is the current `call_count`.
3. Call `track_calls()` several times and observe the output.

Example Interaction:

```
Function called 1 times.
Function called 2 times.
Function called 3 times.
```