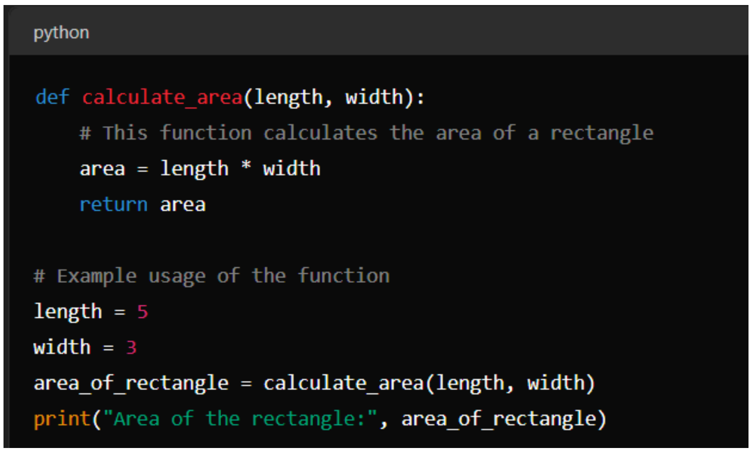
100 Days of Code - Section 10

Functions

In the lecture about functions with outputs, we saw how Python functions use the `return` keyword to give back results after doing something. ***Functions start with `def`, followed by a name and sometimes things we want to work on in brackets***. For example, the function `format_name` showed us how to change `f_name` and `l_name` into title case using `.title()` and then give them back as outputs with `return`. This lets us use the formatted names in our code later on. We used Thonny to step through how the function works, showing how `return` decides what the function sends back. We also compared this to built-in functions like `len()`, which also send back results that we can use. Overall, `return` helps functions tell us what they've done or found, making Python programming clearer and more useful.



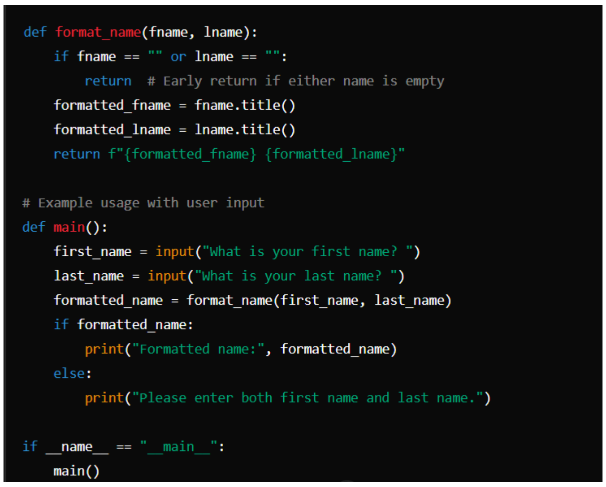
```python
def calculate_area(length, width):
    # This function calculates the area of a rectangle
    area = length * width
    return area

# Example usage of the function
length = 5
width = 3
area_of_rectangle = calculate_area(length, width)
print("Area of the rectangle:", area_of_rectangle)
```

- The function `calculate_area` ***takes two parameters***: `length` and `width`.
- Inside the function, it calculates the area of the rectangle using the formula `length * width`.
- It then uses the ***`return` keyword to send back the calculated area*** as the result of the function.
- Outside the function, we assign values to `length` and `width`, call the `calculate_area` function with these values, and store the returned area in the variable `area_of_rectangle`.
- Finally, we print the calculated area.

Multiple Return Values

In the last lesson, we explored functions that return values and examined what happens when a function contains multiple `return` statements. When the computer encounters a `return` statement, it marks the end of the function execution, and subsequent code after `return` won't be executed. Multiple `return` statements can be used within a function, including cases where `return` is used by itself without returning any specific value.

For instance, in our `format_name` function example, we considered scenarios where the function checks if `fname` or `lname` are empty strings. If either is empty, using an early `return` statement allows the function to exit early without further processing. This prevents unnecessary execution of code that would otherwise operate on invalid inputs.



```python
def format_name(fname, lname):
    if fname == "" or lname == "":
        return  # Early return if either name is empty
    formatted_fname = fname.title()
    formatted_lname = lname.title()
    return f"{formatted_fname} {formatted_lname}"

# Example usage with user input
def main():
    first_name = input("What is your first name? ")
    last_name = input("What is your last name? ")
    formatted_name = format_name(first_name, last_name)
    if formatted_name:
        print("Formatted name:", formatted_name)
    else:
        print("Please enter both first name and last name.")

if __name__ == "__main__":
    main()
```

Interactive Coding Exercise : Days in a month

Sure, let's break down the steps needed to complete this task along with the final code.

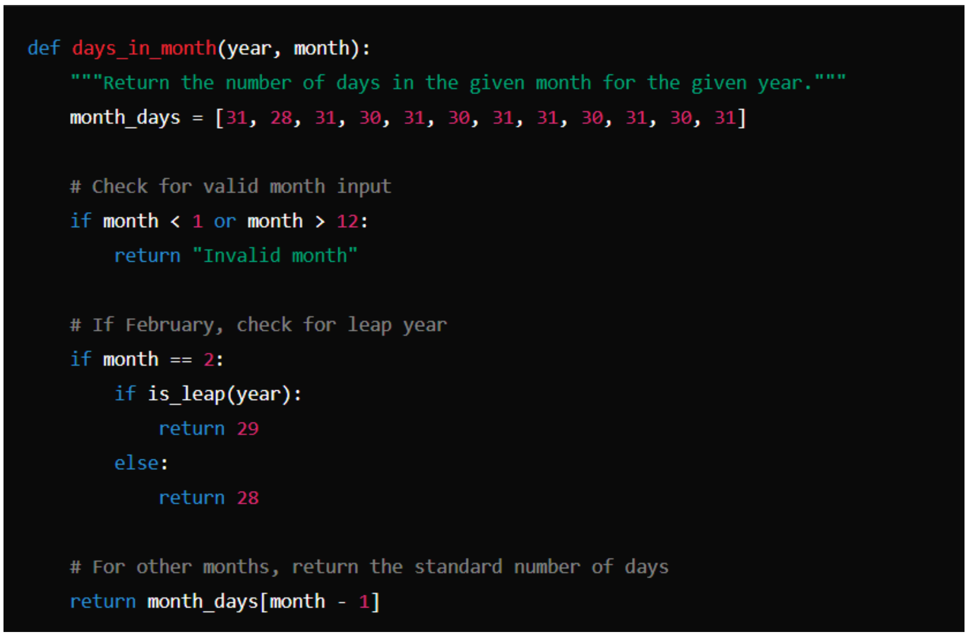
Steps to Complete the Task

1. Modify the `is_leap` Function:
- This function takes a year as input and returns `True` if it's a leap year, otherwise `False`.

2. Define the `days_in_month` Function:
- This function takes two inputs: `year` and `month`.
- It checks if the month is February and whether the given year is a leap year.
- If it's February and a leap year, return 29.
- If it's February and not a leap year, return 28.
- For other months, return the standard number of days from the `month_days` list.

3. Handle Input Validation:
- Ensure the month is within the valid range (1-12).
- If the month is invalid, handle it appropriately (e.g., by returning an error message).

Final Code Implementation



```python
def days_in_month(year, month):
    """Return the number of days in the given month for the given year."""
    month_days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

    # Check for valid month input
    if month < 1 or month > 12:
        return "Invalid month"

    # If February, check for leap year
    if month == 2:
        if is_leap(year):
            return 29
        else:
            return 28

    # For other months, return the standard number of days
    return month_days[month - 1]
```

- In Python, lists are zero-indexed, meaning that the first element of a list has an index of 0, the second element has an index of 1, and so on. However, months in a year are conventionally numbered from 1 to 12.

- To access the correct element in the month_days list, ***we need to adjust the month number by subtracting 1.*** This ***ensures that January (month 1) corresponds to index 0 in the list, February (month 2) corresponds to index 1, and so on.***

Docstrings

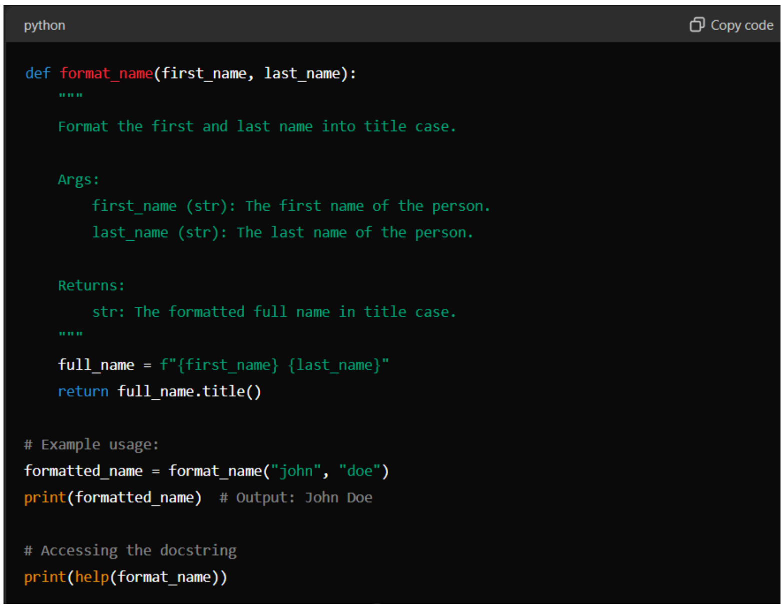
1. Definition:
- Docstrings are inline documentation for Python functions, methods, modules, or classes.
- They are ***enclosed in triple quotes (`""" """`).***

2. Placement:
- The docstring should be the first statement in the function, class, or module.

3. Format:
- Describe the purpose of the function.
- List parameters (`Args:`) and their types.
- Describe the return value (`Returns:`) and its type.
- Use multiline strings for detailed documentation.

4. Accessing Docstrings:
- Use `help(function_name)` or `function_name.__doc__` to view the docstring.

Example Code

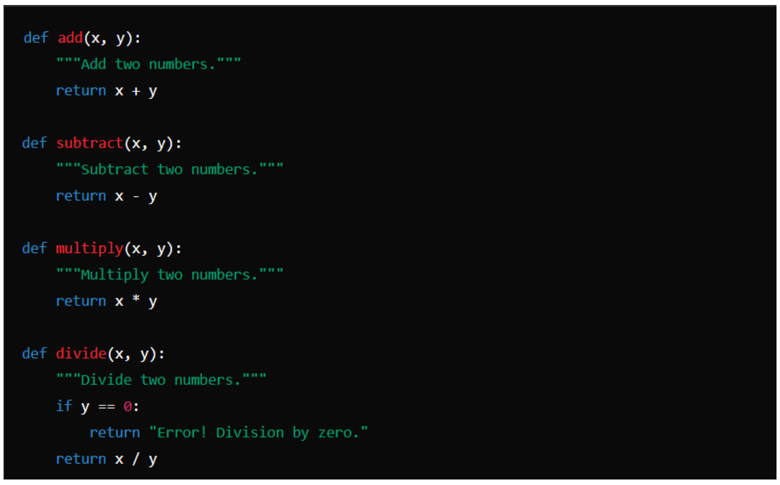


python Copy code

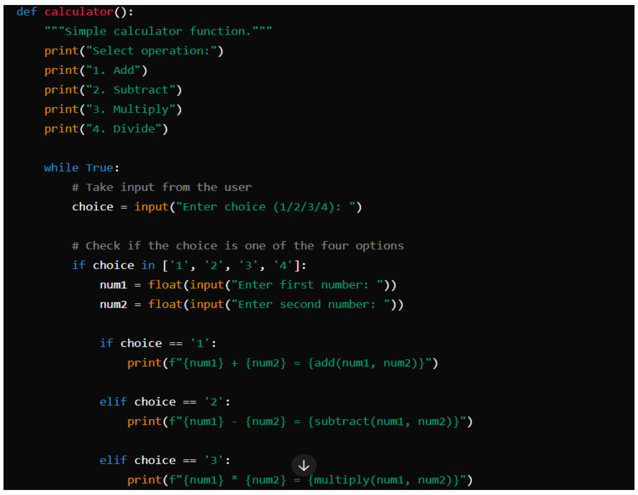
```python
def format_name(first_name, last_name):
    """
    Format the first and last name into title case.

    Args:
        first_name (str): The first name of the person.
        last_name (str): The last name of the person.

    Returns:
        str: The formatted full name in title case.
    """
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Example usage:
formatted_name = format_name("john", "doe")
print(formatted_name)  # Output: John Doe

# Accessing the docstring
print(help(format_name))
```

Best Practices
- Use clear, concise language.
- Follow a consistent style.
- Document all parameters and return values.
- Provide examples if necessary.

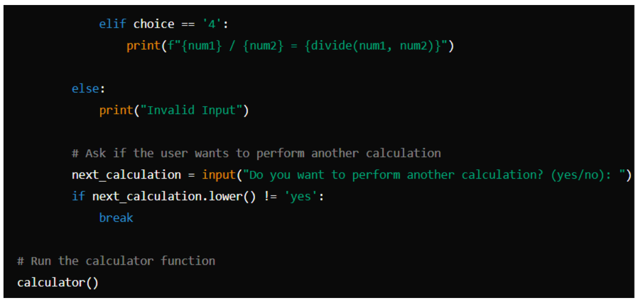
Daily Project - Calculator



```python
def add(x, y):
    """Add two numbers."""
    return x + y

def subtract(x, y):
    """Subtract two numbers."""
    return x - y

def multiply(x, y):
    """Multiply two numbers."""
    return x * y

def divide(x, y):
    """Divide two numbers."""
    if y == 0:
        return "Error! Division by zero."
    return x / y
```



```python
def calculator():
    """Simple calculator function."""
    print("Select operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")

    while True:
        # Take input from the user
        choice = input("Enter choice (1/2/3/4): ")

        # Check if the choice is one of the four options
        if choice in ['1', '2', '3', '4']:
            num1 = float(input("Enter first number: "))
            num2 = float(input("Enter second number: "))

            if choice == '1':
                print(f"{num1} + {num2} = {add(num1, num2)}")

            elif choice == '2':
                print(f"{num1} - {num2} = {subtract(num1, num2)}")

            elif choice == '3':
                print(f"{num1} * {num2} = {multiply(num1, num2)}")
```



```python
            elif choice == '4':
                print(f"{num1} / {num2} = {divide(num1, num2)}")

        else:
            print("Invalid Input")

        # Ask if the user wants to perform another calculation
        next_calculation = input("Do you want to perform another calculation? (yes/no): ")
        if next_calculation.lower() != 'yes':
            break

# Run the calculator function
calculator()
```

Difference Between print and return in Python Functions

print Statement:

- Outputs text to the console.
- Used for debugging or displaying information to the user.
- Does not provide a value that can be used for further computation.

return Statement:

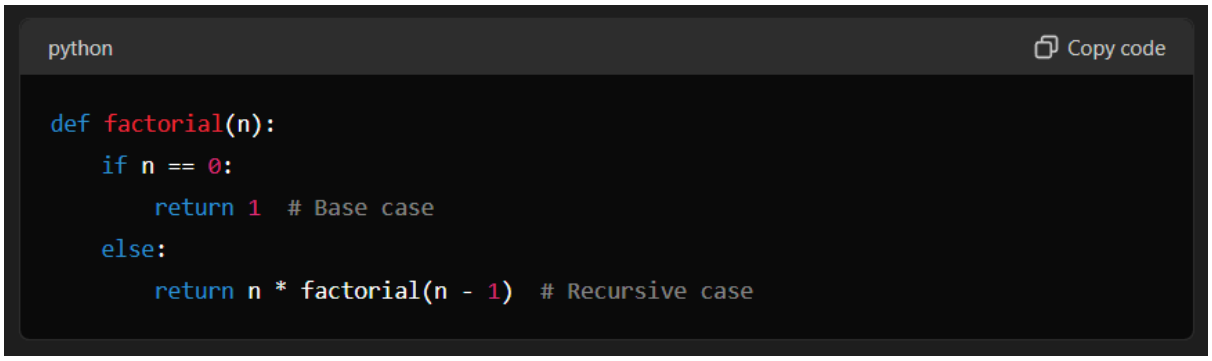
- Exits the function and sends a value back to the caller.
- Allows the function’s output to be used in other parts of the program.
- Essential for creating modular and reusable code.

Recursion

1. **Definition:** Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem.

2. **Base Case:** Every recursive function must have one or more base cases that determine when the function stops calling itself. This prevents infinite recursion.

3. **Recursive Case:** The part of the function that calls itself with a modified version of the original problem.



```python
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case
```