

Module 4: Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions allow us to break a large program into smaller, manageable, and modular chunks.

4.1 Defining and Calling Functions

Syntax: def keyword

Functions are defined using the `def` keyword, followed by the function name, a set of parentheses `()`, and a colon `:`. The function body is indented.

Real-Time Example: Creating a reusable function for temperature conversion.

Python

```
def celsius_to_fahrenheit(celsius):
    """
    Converts a temperature from Celsius to Fahrenheit.
    Formula: F = C * (9/5) + 32
    """
    fahrenheit = celsius * (9/5) + 32
    return fahrenheit
```

Function Call and Execution Flow

To execute the code inside a function, you must **call** it by using its name followed by parentheses and any required arguments. When a function is called, the program's execution jumps to the function's body, executes the statements, and then returns to where it was called from.

Example Program:

```
Python
# Function Call
temp_c = 25
temp_f = celsius_to_fahrenheit(temp_c) # Execution jumps to the function

print(f"{temp_c}°C is equal to {temp_f:.2f}°F")
# Output: 25°C is equal to 77.00°F
```

Docstrings ("""...""")

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It's used to explain what the function does and how to use it. They are accessed using `help(function_name)` or `function_name.__doc__`.

Example: (See the celsius_to_fahrenheit function above for the docstring example)

```
Python
help(celsius_to_fahrenheit)
# Output will display the docstring content
```

The return statement

The return statement is used to exit a function and pass an object (value) back to the caller.

1. A function can return any type of object (data type, list, even another function).
2. If the return statement is omitted, the function implicitly returns the special value **None**.
3. Once return is executed, the function immediately terminates.

Real-Time Example: Returning multiple values (e.g., calculation and status).

Python

```
def calculate_discount(price, discount_rate):
    """Calculates the final price after applying a discount."""
    if discount_rate >= 1.0:
        return price, "Error: Discount must be less than 100%"

    final_price = price * (1 - discount_rate)
    # Returns a tuple containing two values
    return final_price, "Success"

# Unpacking the returned tuple
final_price, status = calculate_discount(price=200, discount_rate=0.25)
print(f"Status: {status}, Final Price: ${final_price:.2f}")

# Example of implicit None return
def log_message(msg):
    print(f"LOG: {msg}")
    # No return statement

result = log_message("App started")
print(f"Return value of log_message: {result}") # Output: Return value of log_message: None
```

4.2 Arguments and Parameters

Parameters are the names defined in the function definition. **Arguments** are the actual values passed to the function when it is called.

Positional Arguments

The arguments are matched to the parameters based on their position/order.

Example Program:

Python

```
def check_credentials(username, password):
    # username is matched to the 1st argument, password to the 2nd
    print(f"Checking credentials for: {username}")
```

```
# ... logic ...

check_credentials("admin_user", "secure_pass") # Positional matching
```

Keyword Arguments

Arguments are matched to parameters using the parameter name, which allows them to be passed out of order.

Example Program:

Python

```
def process_order(item_name, quantity, customer_id):
    print(f"Order for {item_name} (x{quantity}) placed by customer ID {customer_id}")

# Arguments are passed using their keyword, order does not matter
process_order(quantity=2, customer_id="C456", item_name="T-shirt")
```

Default Arguments

Parameters can be given a default value in the function definition. If the caller does not provide an argument for that parameter, the default value is used. Default arguments must be defined **after** any non-default arguments.

Real-Time Example: Logging function with an optional severity level.

Python

```
def log_event(message, level="INFO"): # 'level' has a default value
    """Logs an application event with an optional severity level."""
    print(f"[{level.upper()}]: {message}")

log_event("User logged in successfully")      # Uses default level="INFO"
log_event("Database connection failed", "ERROR") # Overrides default
```

Variable-Length Arguments: *args (Non-Keyword) and **kwargs (Keyword)

These allow a function to accept an arbitrary, unknown number of arguments.

- ***args (Non-Keyword Arguments):** Collects a variable number of positional arguments into a **tuple**.
- ****kwargs (Keyword Arguments):** Collects a variable number of keyword arguments into a **dictionary**.

Real-Time Example: Function to calculate the sum of any number of values (*args) and configure settings (**kwargs).

Python

```
# *args example
def calculate_total_sum(*numbers):
    """Calculates the sum of all passed numbers."""
    # 'numbers' is a tuple of all positional arguments
    total = sum(numbers)
    return total

print(f"Total sales: {calculate_total_sum(10, 20, 30, 45.5)}")
# Output: Total sales: 105.5
```