# Topic 1: Functions

## Pre-defined and User-defined Functions

**Learning Outcomes:**

- Identify concepts of functions
- Understand function components
- Write simple user-defined functions

**Duration:** 3 hours

## Opening Problem

**Imagine you need to calculate the average of test scores for 100 students.**

Would you write the same calculation 100 times?

```
student1_avg = (80 + 90 + 85) / 3
student2_avg = (75 + 88 + 92) / 3
student3_avg = (95 + 87 + 90) / 3
# ... 97 more times?
```

**There must be a better way**

## The Solution: Functions

**Functions let us write code once and use it many times**

Instead of copying the same calculation, we can:

1. Define the calculation once
2. Give it a name
3. Use it whenever we need it

This is what **functions** do.

## Two Types of Functions

Every function you'll use falls into one of two categories:

1. **Pre-defined Functions** - Built into Python
2. **User-defined Functions** - Created by you

Let's explore both...

## Pre-defined Functions

**Definition:** Functions that are readily available in Python

**You've already used them:**

```
# Getting user input
name = input("Enter your name: ")

# Displaying output
print("Hello, " + name)

# Finding length
length = len(name)

# Calculating power
result = pow(2, 3)  # 2 to the power of 3 = 8
```

## Exploring Pre-defined Functions

**Let me show you how versatile they are:**

```python
# len() works on strings
message = "Hello World"
print(len(message))         # Counts characters

# len() also works on other types
fruits = "apple"
print(len(fruits))          # Still counts characters

# input() gets data from users
age = input("Enter your age: ")

# print() displays information
print("You are", age, "years old")
```

## User-defined Functions

**Definition:** Functions that YOU create to solve specific problems

**A simple example:**

```
def greet():
    print("Hello from my function!")

greet()
```

**Output:**

```
Hello from my function!
```

We just created our own function and used it.

## Exercise 1: Identify Function Types

**Look at this code. Which functions are pre-defined and which are user-defined?**

```python
def welcome_student(name):
    print("Welcome to CP125, " + name)

student = input("Enter your name: ")
welcome_student(student)
length = len(student)
print("Your name has", length, "characters")
```

**Questions:**

- List the pre-defined functions
- List the user-defined functions

## Anatomy of a Function

**Every function has three parts:**

```
def greet(name):                   # 1. Function Header
    message = "Hello, " + name     # 2. Function Body
    print(message)                 #    (indented)

greet("Sarah")                     # 3. Function Call
```

Let's examine each part in detail...

## Part 1: Function Header

**The function header is the first line that defines your function:**

```python
def calculate_total(price, tax):
```

**It has four elements:**

```
def calculate_total(price, tax):
│   │               │          │
│   │               │          └─ 4. Colon (:)
│   │               └──────────── 3. Parameters
│   └──────────────────────────── 2. Function name
└──────────────────────────────── 1. def keyword
```

## Element 1: The def Keyword

**Every function definition starts with `def`:**

```
def greet():
    print("Hello")

def calculate():
    return 5 + 3

def display_info():
    print("Information")
```

**The `def` keyword tells Python:** "I'm about to define a function"

## Element 2: Function Name

**Name your functions using snake_case:**

**Good names:**

```
def calculate_average()
def find_maximum()
def display_student_info()
def get_user_input()
```

**Bad names:**

```
def CalculateAverage()  # Wrong: CamelCase (use for classes)
def avg()               # Wrong: Not descriptive
def calc avg()          # Wrong: Spaces not allowed
```

**Rule:** Use lowercase words separated by underscores

## Element 3: Parentheses and Parameters

**Parentheses `()` come after the function name:**

```python
# Without parameters (empty parentheses)
def greet():
    print("Hello")

# With one parameter
def greet(name):
    print("Hello, " + name)

# With multiple parameters
def calculate_total(price, quantity, tax):
    total = price * quantity * (1 + tax)
    return total
```

**Parameters are inputs the function needs to do its job**

## Element 4: The Colon

**The colon `:` marks the end of the header:**

```python
def greet():     # Colon here
    print("Hi")

def add(a, b):   # Colon here
    return a + b
```

**Without the colon, you'll get an error:**

```python
def greet()      # Error: Missing colon
    print("Hi")
```

## Exercise 2: Create Function Headers

**Write the function headers for these descriptions:**

1. A function named `display_message` with no parameters
2. A function named `calculate_sum` with two parameters: `num1` and `num2`
3. A function named `find_average` with three parameters: `a` , `b` , and `c`
4. A function named `greet_user` with one parameter: `username`

*Remember: Just the header line, ending with a colon*

*Answers will be presented separately*

## Part 2: Function Body

**The function body contains the actual code that runs:**

```python
def calculate_circle_area(radius):
    pi = 3.14159                        # These lines
    area = pi * radius * radius         # are the
    return area                         # function body
```

**Key rules:**

- Must be **indented** (4 spaces or 1 tab)
- Contains one or more statements
- Performs the actual work of the function

## Indentation is Required

**Python uses indentation to know what's inside the function:**

**Correct:**

```python
def greet():
    print("Hello")         # Indented - part of function
    print("Welcome")       # Indented - part of function

print("Outside")           # Not indented - outside function
```

**Incorrect:**

```python
def greet():
print("Hello")             # Error: Not indented
```

## Part 3: Function Call

**To use a function, you must call (invoke) it:**

```python
# Function definition
def greet():
    print("Hello!")

# Function call
greet()
```

**The function call:**

- Uses the function name
- Includes parentheses `()`
- Can be done multiple times

## Calling Functions Multiple Times

**Once defined, call it as many times as needed:**

```python
def greet():
    print("Hello!")

# Call it three times
greet()
greet()
greet()
```

**Output:**

```
Hello!
Hello!
Hello!
```

This is the **reusability** power of functions.

## Exercise 3: Complete the Code

**Given this function definition:**

```python
def display_welcome():
    print("Welcome to CP125")
    print("Let's learn Python!")
```

**Write the code to:**

1. Call this function once
2. Call this function three times in a row

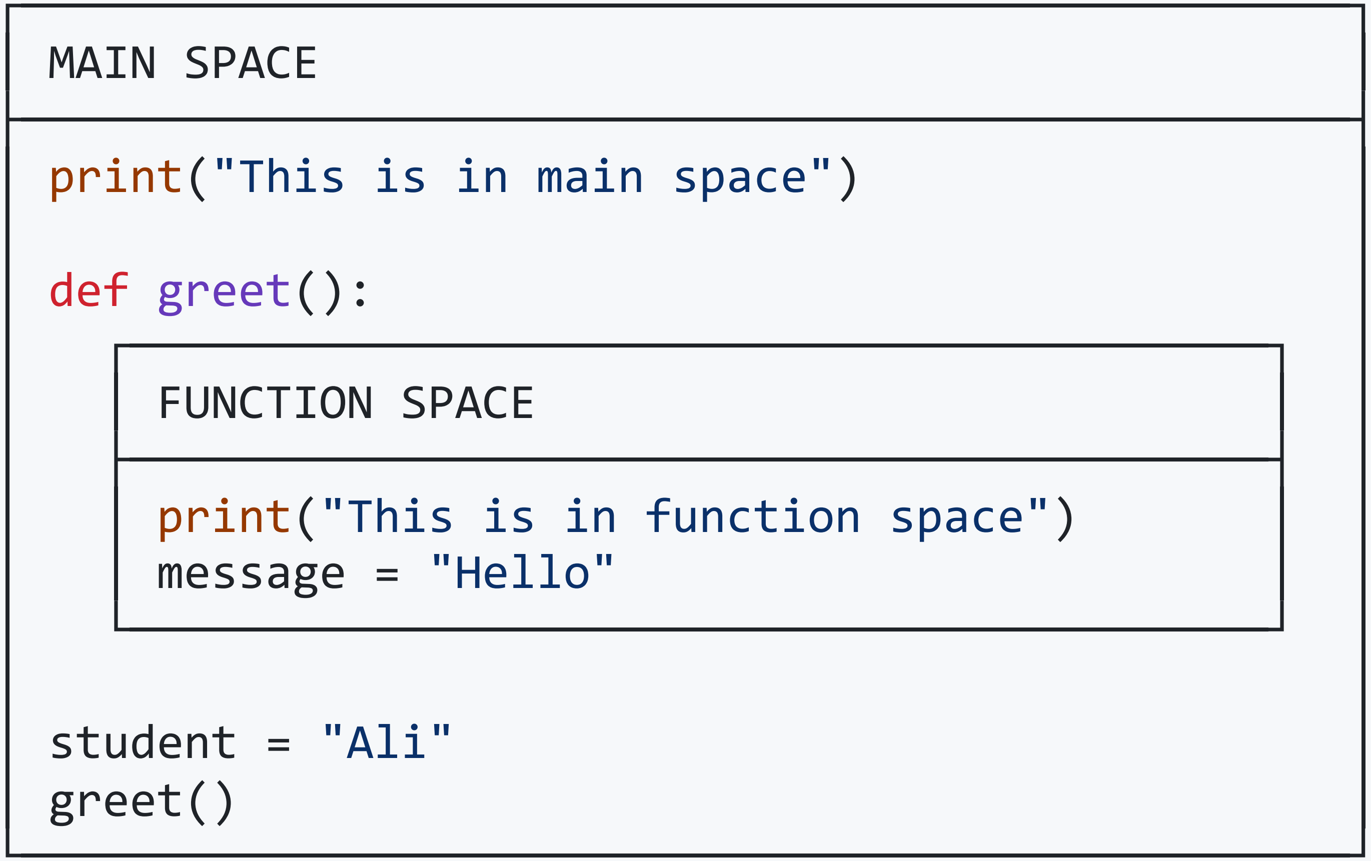
## Understanding Code Spaces

**Your Python program has two distinct spaces where code lives:**

1. **Main Space** - Code outside any function
2. **Function Space** - Code inside a function

Let's see the difference...

## Main Space vs Function Space



```
MAIN SPACE

print("This is in main space")

def greet():
    FUNCTION SPACE

    print("This is in function space")
    message = "Hello"

student = "Ali"
greet()
```

**Key idea:** Variables in function space are separate from variables in main space

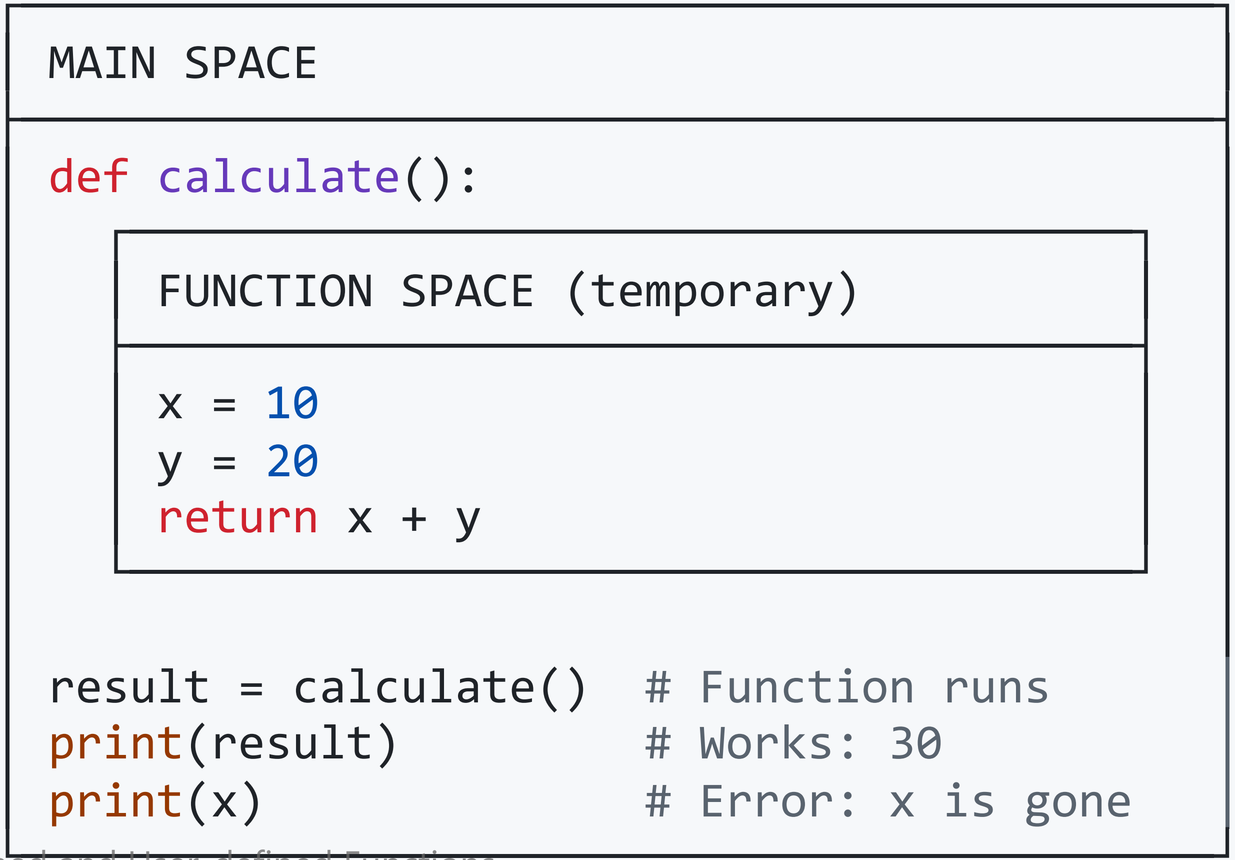
## Visualizing the Two Spaces

```
## MAIN SPACE ##

name = "Sarah"

def display_greeting():
    ## FUNCTION SPACE ##
    greeting = "Hello, " + name
    print(greeting)


display_greeting()
print(name)         # Works: name is in main space
print(greeting)     # Error: greeting doesn't exist
                    # in main space
```

**Important:** Variables created inside a function only exist while that function runs

## Why This Matters

**Function space is temporary:**



```
MAIN SPACE

def calculate():
    FUNCTION SPACE (temporary)

    x = 10
    y = 20
    return x + y


result = calculate()  # Function runs
print(result)         # Works: 30
print(x)              # Error: x is gone
```

## Execution Order Matters

**Python reads code from top to bottom:**

**This works:**

```
def greet():                # Define first
    print("Hello")

greet()                     # Call after
```

**This fails:**

```
greet()                     # Error: function not defined yet

def greet():
    print("Hello")
```

**Rule: Always define functions BEFORE calling them**

## Understanding the Error

**When you call before defining:**

```python
message()

def message():
    print("Hi")
```

**Python gives you:**

```
NameError: name 'message' is not defined
```

**Why?** Python hasn't reached the definition yet, so it doesn't know `message` exists. Recall that Python is interpreted language, not compiled.

## Exercise 4: Fix the Order

**This code has an error. Fix it:**

```
calculate_sum(5, 10)

print("Program starting")

def calculate_sum(a, b):
    result = a + b
    print("Sum is:", result)

print("Program ending")
```

## How Execution Flows

**Watch how Python jumps into and out of functions:**

```
def greet(name):
    print("Hello, " + name)
print("Start")    # 1. Execute this
greet("Ali")      # 2. Jump to function
                  # 3. Execute function body
                  # 4. Return here
print("End")      # 5. Execute this
```

**Output:**

```
Start
Hello, Ali
Welcome!
End
```

## Parameters and Arguments

**These terms are often confused:**

```
def greet(name):                    # 'name' is a PARAMETER
    print("Hello, " + name)

greet("Sarah")                      # "Sarah" is an ARGUMENT
```

**Remember:**

- **Parameters** = Variables in the function definition (empty boxes)
- **Arguments** = Actual values when calling the function (values in boxes)

## Parameters: The Empty Boxes

**Parameters are placeholders for data:**

```python
def add(x, y):      # x and y are parameters
    total = x + y   # We use them in the function
    print(total)

# When called, they get actual values
add(5, 3)           # x becomes 5, y becomes 3
add(10, 20)         # x becomes 10, y becomes 20
```

**Parameters let functions work with different data**

## Multiple Parameters

**Functions can have multiple parameters:**

```python
# One parameter
def greet(name):
    print("Hello, " + name)

# Two parameters
def add(a, b):
    print(a + b)

# Three parameters
def calculate_total(price, quantity, tax):
    subtotal = price * quantity
    total = subtotal * (1 + tax)
    print(total)
```

**Separate multiple parameters with commas**

## Arguments Must Match Parameters

**The number of arguments must match the number of parameters:**

```python
def greet(first_name, last_name):
    print("Hello", first_name, last_name)

greet("Ahmad", "Ali")      # Correct: 2 arguments for 2 parameters
greet("Ahmad")             # Error: Missing last_name
greet("Ahmad", "Ali", "X") # Error: Too many arguments
```

## Exercise 5: Parameters vs Arguments

**Examine this code:**

```python
def calculate_bmi(weight, height):
    bmi = weight / (height ** 2)
    print("Your BMI is:", bmi)

calculate_bmi(70, 1.75)
```

**Questions:**

1. What are the parameters?
2. What are the arguments?
3. What value does `weight` have when the function runs?
4. What value does `height` have when the function runs?

## Exercise 6: Create and Call a Function

**Create a function that:**

- Is named `display_student_info`
- Has two parameters: `name` and `student_id`
- Prints both pieces of information

**Then call your function with:**

- Name: "Fatimah"
- Student ID: "12345"

## Two Ways to Output: Print vs Return

**Functions can give output in two ways:**

1. **print()** - Display to screen
2. **return** - Give value back to use later

**Let's explore the difference**

## Using Print in Functions

**Functions can print information directly:**

```python
def greet(name):
    print("Hello, " + name)

greet("Ali")
```

**Output:**

```
Hello, Ali
```

**The function displays something but doesn't give back a value**

## Using Return in Functions

**Functions can return (give back) a value:**

```python
def add(a, b):
    total = a + b
    return total

result = add(5, 3)
print("The sum is:", result)
```

**Output:**

```
The sum is: 8
```

**The function gives back a value that we can store and use**

## The Difference: Print vs Return

**With print only:**

```
def add_print(a, b):
    total = a + b
    print(total)

x = add_print(5, 3)
print("x is:", x)
```

**Output:**

```
8
x is: None
```

## With return:

```python
def add_return(a, b):
    total = a + b
    return total

y = add_return(5, 3)
print("y is:", y)
```

## Output:

```
y is: 8
```

## The Mystery of None

**Why does the print-only function return None?**

```python
def add(a, b):
    total = a + b
    print(total)      # Displays but doesn't return

result = add(5, 3)   # What gets stored in result?
```

**Answer:** When a function has **no return statement**, it automatically returns `None`

**`None` is a Python way of saying, "I am returning you something, but it's empty"**

## When to Use Print vs Return

**Use `print()` when:**

- You want to display information to the user
- The function's job is to show output
- Example: `display_report()`, `show_menu()`

**Use `return` when:**

- You need to use the result in calculations
- You want to store the value in a variable
- You want to pass the result to another function
- Example: `calculate_total()`, `find_max()`, `get_average()`

## You Can Use Both

### A function can both print AND return:

```
def calculate_total(price, quantity):
    total = price * quantity
    print("Calculating:", price, "×", quantity)  # Show process
    return total                                  # Give back result

amount = calculate_total(50, 3)
print("Total amount:", amount)
```

### Output:

```
Calculating: 50 × 3
Total amount: 150
```

## Exercise 7: Predict the Output

**What will this code print?**

```
def mystery_function(x):
    result = x * 2
    print("Inside function:", result)

answer = mystery_function(5)
print("Outside function:", answer)
```

**Write down what you think will be printed**

## Creating Your Own Functions

**Now you know all the parts. Let's practice creating functions step-by-step:**

**Process:**

1. Write the function header (def, name, parameters, colon)
2. Write the function body (indented code)
3. Call the function (use it)

## Example: Create a Simple Function

Let's create a function to display a border:

### Step 1: Header

```
def show_border():
```

### Step 2: Body

```
def show_border():
    print("=" * 40)
```

## Step 3: Call it

```python
def show_border():
    print("=" * 40)

show_border()
```

## Example: Function with Parameters

Create a function to display a custom message:

Step 1: Header with parameter

```python
def show_message(text):
```

Step 2: Body

```python
def show_message(text):
    print("=" * 40)
    print(text)
    print("=" * 40)
```

## Step 3: Call with different arguments

```
show_message("Welcome to CP125")
show_message("Hello World")
```

## Example: Function with Return

Create a function to calculate rectangle area:

### Step 1: Header

```
def calculate_area(length, width):
```

### Step 2: Body with return

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

## Step 3: Call and use the result

```python
room_area = calculate_area(5, 4)
print("The room area is:", room_area, "square meters")
```

## Exercise 8: Create Function Without Parameters

**Create a function that:**

- Is named `display_greeting`
- Has no parameters
- Prints "Hello, welcome to CP125!"
- Prints "Let's learn about functions"

**Then call your function**

*Answer will be presented separately*

## Exercise 9: Create Function with Parameters, No Return

**Create a function that:**

- Is named `greet_student`
- Has two parameters: `name` and `course`
- Prints "Hello [name], welcome to [course]"

**Then call your function with:**

- name: "Ahmad"
- course: "CP125"

## Exercise 10: Create Function with Return

**Create a function that:**

- Is named `multiply`
- Has two parameters: `a` and `b`
- Returns the product of a and b

**Then:**

- Call it with 6 and 7
- Store the result in a variable
- Print the result

## Why Use Functions? Deeper Principles

**Now that you can create functions, let's learn when and why to use them**

We'll explore five key principles:

1. Factorization
2. Single Responsibility
3. Composition
4. Decomposition
5. Abstraction

## Principle 1: Factorization

**When you see repetition, extract it into a function**

**Problem: Repetitive code**

```
student1_avg = (80 + 90 + 85) / 3
print("Student 1:", student1_avg)

student2_avg = (75 + 88 + 92) / 3
print("Student 2:", student2_avg)

student3_avg = (95 + 87 + 90) / 3
print("Student 3:", student3_avg)

# Same calculation repeated 3 times
```

## Factorization: The Solution

**Extract the repeated pattern:**

```
def calculate_average(score1, score2, score3):
    avg = (score1 + score2 + score3) / 3
    return avg

# Now use it for all students
print("Student 1:", calculate_average(80, 90, 85))
print("Student 2:", calculate_average(75, 88, 92))
print("Student 3:", calculate_average(95, 87, 90))
```

**Benefits:**

- Less code to write
- Easier to modify (change once, affects all)
- Fewer chances for errors

## Factorization Rule

**If you copy-paste code 3 or more times, make it a function**

**Signs you need factorization:**

- Similar code blocks repeated
- Only the values change, logic stays the same
- You're using copy-paste frequently

**Action:** Extract the common logic into a function with parameters

## Exercise 11: Apply Factorization

**This code is repetitive. Refactor it using a function:**

```
r1 = 5
area1 = 3.14159 * r1 * r1
print("Circle 1 area:", area1)

r2 = 10
area2 = 3.14159 * r2 * r2
print("Circle 2 area:", area2)

r3 = 7
area3 = 3.14159 * r3 * r3
print("Circle 3 area:", area3)
```

**Create a function and use it to eliminate the repetition**

## Principle 2: Single Responsibility

**Each function should do ONE thing and do it well**

**Let's continue improving our grade system**

**Version 2 had one function, but still had problems:**

- Final grade calculation mixed with letter grade logic
- Display mixed in main code
- Each part doing multiple jobs

## Apply Single Responsibility

**Split complex code into focused functions:**

```python
# Each function does ONE thing
def calculate_average(score1, score2, score3):
    return (score1 + score2 + score3) / 3

def calculate_final_grade(assign_avg, midterm, final):
    return (assign_avg * 0.4) + (midterm * 0.3) + (final * 0.3)

def get_letter_grade(grade):
    if grade >= 90:
        return "A"
    elif grade >= 80:
        return "B"
    elif grade >= 70:
        return "C"
    else:
        return "D"
```

## Now use them

```python
assign_avg = calculate_average(85, 90, 78)
final_grade = calculate_final_grade(assign_avg, 85, 90)
letter = get_letter_grade(final_grade)
print(f"Final: {final_grade:.2f} ({letter})")
```

**Much clearer! Each function has ONE job**

## Single Responsibility (More)

```
    name = input("Enter name: ")           # Getting input
    age = int(input("Enter age: "))        # Getting input
    grade = (80 + 90 + 85) / 3             # Calculating
    print(f"{name} is {age} years old")    # Displaying
    print(f"Grade: {grade}")               # Displaying
    return grade                           # Returning
```

**Problem:** This function gets input, calculates, displays, AND returns. Too many jobs.

## Single Responsibility: The Solution

**Split into focused functions:**

```python
def get_name():
    return input("Enter name: ")

def get_age():
    return int(input("Enter age: "))

def calculate_grade(scores):
    return sum(scores) / len(scores)

def display_info(name, age, grade):
    print(f"{name} is {age} years old")
    print(f"Grade: {grade}")
```

**Now each function has ONE clear purpose**

## Single Responsibility Test

**Ask yourself:** "Can I describe what this function does in 5 words or less?"

**Good (one responsibility):**

- `calculate_average` - "Finds the average of numbers"
- `display_menu` - "Shows the menu options"
- `get_user_input` - "Gets input from user"

**Bad (multiple responsibilities):**

- `process_and_display_student_grades_and_save` - Too many jobs

**Rule:** If you can't describe it briefly, the function does too much

## Exercise 12: Identify Multiple Responsibilities

**This function does too many things. What are they?**

```python
def handle_order(item_name, price, quantity):
    total = price * quantity
    tax = total * 0.06
    final = total + tax
    print(f"Item: {item_name}")
    print(f"Quantity: {quantity}")
    print(f"Total: ${final}")
    return final
```

**List all the different jobs this function does**

## Principle 3: Composition

**Combine simple functions to build complex behavior**

**Simple building blocks:**

```
def add(a, b):
    return a + b

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

**Compose them into something more complex:**

```
def calculate_rectangle_perimeter(length, width):
    side1 = multiply(2, length)
    side2 = multiply(2, width)
    return add(side1, side2)
```

## Composition: Functions Calling Functions

**Build layers of abstraction:**

```python
# Layer 1: Basic operations
def calculate_subtotal(price, quantity):
    return price * quantity

def calculate_tax(subtotal, rate):
    return subtotal * rate

# Layer 2: Composed function
def calculate_total(price, quantity, tax_rate):
    subtotal = calculate_subtotal(price, quantity)
    tax = calculate_tax(subtotal, tax_rate)
    return subtotal + tax
```

```python
# Layer 3: Even more complex
def process_order(price, quantity, tax_rate):
    total = calculate_total(price, quantity, tax_rate)
    print(f"Order total: ${total:.2f}")
    return total
```

## Exercise 13: Function Composition

**Use these two functions:**

```
def square(n):
    return n * n

def add(a, b):
    return a + b
```

**Create a new function `sum_of_squares` that:**

- Takes two numbers as parameters
- Returns the sum of their squares
- Uses both `square()` and `add()` functions

**Example:** `sum_of_squares(3, 4)`

## Principle 4: Decomposition

**Break big problems into smaller, manageable pieces**

**Bad: One giant function**

```
def process_order(price, qty, tax_rate):
    subtotal = price * qty
    tax = subtotal * tax_rate
    shipping = 10 if subtotal < 50 else 0
    discount = subtotal * 0.1 if subtotal > 200 else 0
    total = subtotal + tax + shipping - discount
    return total
```

**Problem:** Hard to understand, hard to test, hard to modify

## Decomposition: The Solution

**Break into smaller functions:**

```python
def calculate_subtotal(price, quantity):
    return price * quantity

def calculate_tax(subtotal, rate):
    return subtotal * rate

def calculate_shipping(subtotal):
    return 10 if subtotal < 50 else 0

def calculate_discount(subtotal):
    return subtotal * 0.1 if subtotal > 200 else 0
```

```python
def process_order(price, qty, tax_rate):
    subtotal = calculate_subtotal(price, qty)
    tax = calculate_tax(subtotal, tax_rate)
    shipping = calculate_shipping(subtotal)
    discount = calculate_discount(subtotal)
    return subtotal + tax + shipping - discount
```

## Principle 5: Abstraction

**Hide complexity behind meaningful names**

**Good function names let you read code like English:**

```
# Instead of seeing complex calculations
final_grade = (sum(assignments)/len(assignments) * 0.4 +
               midterm * 0.3 + final_exam * 0.3)

# You see clear, readable code
assignment_avg = calculate_assignment_average(assignments)
final_grade = calculate_final_grade(assignment_avg, midterm, final_exam)
```

**Users don't need to know HOW it works, just WHAT it does**

## Abstraction: Choosing Good Names

### Good abstractions:

```
def calculate_bmi(weight, height)
def find_maximum(numbers)
def is_valid_email(email)
def convert_to_celsius(fahrenheit)
```

### Bad abstractions:

```
def do_stuff(x, y)              # What stuff?
def calc(a, b, c)               # Calculate what?
def process(data)               # Process how?
```

**Rule:** Function names should clearly describe what they do, not how they do it

## Example: Grade Calculator

**Let's apply all principles to build a complete program**

**Requirements:**

- Calculate assignment average
- Calculate final course grade (weighted)
- Determine letter grade
- Display a report

## Grade Calculator: Implementation

```
def calculate_assignment_average(scores):
    return sum(scores) / len(scores)

def calculate_final_grade(assignments, midterm, final):
    assign_avg = calculate_assignment_average(assignments)
    final_grade = (assign_avg * 0.4) + (midterm * 0.3) + (final * 0.3)
    return final_grade

def get_letter_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"
```

## Grade Calculator: Display and Main

```
def display_report(score, letter):
    print("\n" + "=" * 30)
    print(f"Final Score: {score:.2f}")
    print(f"Letter Grade: {letter}")
    print("=" * 30)

# Using the program (composition)
assignments = [85, 90, 78, 92]
midterm_score = 85
final_exam = 90

final = calculate_final_grade(assignments, midterm_score, final_exam)
letter = get_letter_grade(final)
display_report(final, letter)
```

**Notice:**

- Each function has one job (Single Responsibility)

## Exercise: BMI Calculator

**Build a complete BMI calculator with multiple functions:**

**Requirements:**

1. Function to get weight from user
2. Function to get height from user
3. Function to calculate BMI: `weight / (height²)`

4. Function to interpret BMI:
   - < 18.5: "Underweight"
   - 18.5-24.9: "Normal"
   - 25-29.9: "Overweight"
   - ≥ 30: "Obese"
5. Function to display results

**Apply all the principles we learned**

## Common Mistakes to Avoid

### 1. Calling before defining

```
greet()            # Error!
def greet():
    print("Hi")
```

### 2. Forgetting return

```
def add(a, b):
    total = a + b  # No return!
```

### 3. Wrong number of arguments

```python
def greet(first, last):
    print("Hello")

greet("Ali")     # Error: Missing argument
```

### 4. Using wrong variable names

```python
def add(x, y):
    return a + b  # Error: Should be x + y
```

### 5. Missing indentation

```python
def greet():
print("Hi")      # Error: Not indented
```

## 6. Missing colon

```python
def greet()       # Error: Missing colon
    print("Hi")
```

## 7. Code after return (unreachable)

```python
def check(x):
    return x > 0
    print("This never runs")  # Never executed
```

## 8. Confusing print with return

```python
def double(x):
    print(x * 2)  # Displays but doesn't return

result = double(5)  # result is None, not 10
```

## Key Takeaways

**What we covered:**

1. Two types of functions (pre-defined and user-defined)
2. Function anatomy (header, body, call)
3. Execution order matters
4. Parameters vs arguments
5. Print vs return
6. Creating your own functions
7. Five engineering principles:
   - Factorization, Single Responsibility, Composition, Decomposition, Abstraction

**Remember:** Good functions make code readable, reusable, and maintainable

## Review Questions

**Can you:**

1. Identify pre-defined vs user-defined functions?
2. Write a complete function with header, body, and call?
3. Explain the difference between parameters and arguments?
4. Decide when to use print vs return?
5. Apply the five principles to write better functions?

## Hands-On Transformation: From Messy to Beautiful

**Let's transform beginner code into professional code**

**Problem:** Student grade system

- Calculate assignment average
- Calculate final grade (weighted)
- Determine letter grade
- Display results

**Watch the transformation happen step-by-step**

## Version 1: Beginner Code (Main Space Only)

```python
# Student 1
a1, a2, a3 = 85, 90, 78
assign_avg1 = (a1 + a2 + a3) / 3
grade1 = (assign_avg1 * 0.4) + (85 * 0.3) + (90 * 0.3)
if grade1 >= 90:
    letter1 = "A"
elif grade1 >= 80:
    letter1 = "B"
else:
    letter1 = "C"
print(f"Student 1: {grade1:.2f} ({letter1})")

# Student 2 - SAME CODE AGAIN
a1, a2, a3 = 92, 88, 95
assign_avg2 = (a1 + a2 + a3) / 3
grade2 = (assign_avg2 * 0.4) + (90 * 0.3) + (88 * 0.3)
# ... copy-paste nightmare continues
```

## Version 2: Apply Factorization

```
def calc_avg(s1, s2, s3):
    return (s1 + s2 + s3) / 3

# Still messy, but less repetition
assign_avg1 = calc_avg(85, 90, 78)
grade1 = (assign_avg1 * 0.4) + (85 * 0.3) + (90 * 0.3)
if grade1 >= 90:
    letter1 = "A"
# ...

assign_avg2 = calc_avg(92, 88, 95)
grade2 = (assign_avg2 * 0.4) + (90 * 0.3) + (88 * 0.3)
# ... still repeating letter logic
```

## Version 3: Add Single Responsibility

```python
def calc_avg(s1, s2, s3):
    return (s1 + s2 + s3) / 3

def calc_final(assign_avg, mid, final):
    return (assign_avg * 0.4) + (mid * 0.3) + (final * 0.3)

def get_letter(grade):
    if grade >= 90:
        return "A"
    elif grade >= 80:
        return "B"
    else:
        return "C"

# Better! Each function has ONE job
```

## Version 4: Apply Composition

```
def calc_avg(s1, s2, s3):
    return (s1 + s2 + s3) / 3

def calc_final(assign_avg, mid, final):
    return (assign_avg * 0.4) + (mid * 0.3) + (final * 0.3)

def get_letter(grade):
    if grade >= 90:
        return "A"
    elif grade >= 80:
        return "B"
    else:
        return "C"

def process_student(a1, a2, a3, mid, final):
    avg = calc_avg(a1, a2, a3)
    grade = calc_final(avg, mid, final)
    letter = get_letter(grade)
    return grade, letter

# Functions calling functions!
```

## Version 5: Apply Abstraction (Final)

```python
def calculate_assignment_average(score1, score2, score3):
    return (score1 + score2 + score3) / 3

def calculate_weighted_grade(assign_avg, midterm, final_exam):
    return (assign_avg * 0.4) + (midterm * 0.3) + (final_exam * 0.3)

def determine_letter_grade(numeric_grade):
    if numeric_grade >= 90:
        return "A"
    elif numeric_grade >= 80:
        return "B"
    elif numeric_grade >= 70:
        return "C"
    else:
        return "D"

def format_report(student_id, grade, letter):
    return f"Student {student_id}: {grade:.2f} ({letter})"
```

## Using the Final Version

```python
# Student 1
avg1 = calculate_assignment_average(85, 90, 78)
grade1 = calculate_weighted_grade(avg1, 85, 90)
letter1 = determine_letter_grade(grade1)
print(format_report(1, grade1, letter1))

# Student 2
avg2 = calculate_assignment_average(92, 88, 95)
grade2 = calculate_weighted_grade(avg2, 90, 88)
letter2 = determine_letter_grade(grade2)
print(format_report(2, grade2, letter2))
```

**Output:**

```
Student 1: 85.27 (B)
Student 2: 90.30 (A)
```

## The Transformation Summary

**All 5 Principles Applied:**

- **Factorization:** Extracted repeated calculations
- **Single Responsibility:** Each function does one thing
- **Composition:** Functions call other functions
- **Decomposition:** Big problem broken into pieces
- **Abstraction:** Clear names hide complexity

**From 50+ lines of repetitive code to clean, maintainable functions**

## Thank You

**Questions?**

Keep practicing and happy coding with functions!