Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #3 - Functions

Summary

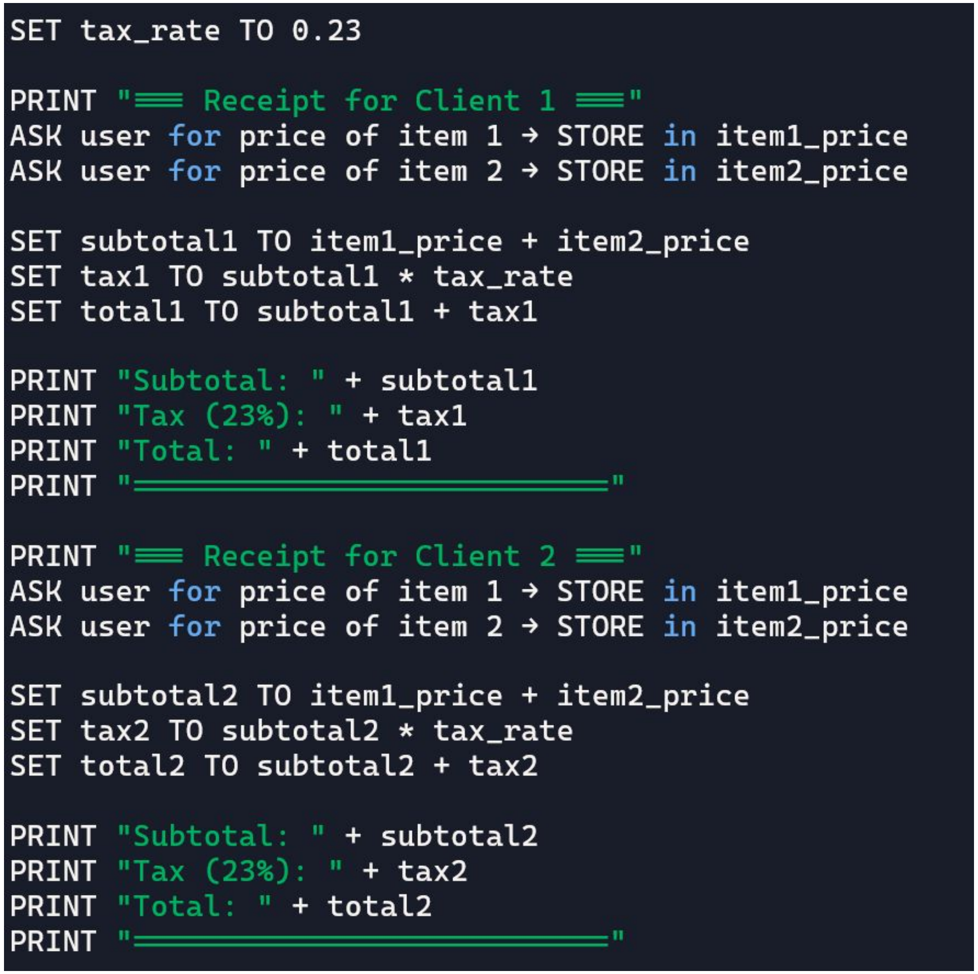
- The need for subprograms / functions
- Functions in Python:
  - Definition and invocation
- Parameters and local variables
- Execution
- Anonymous functions (Lambda expressions)
- Typical uses
- Common mistakes

Motivating Problem

- Receipt Generator

- Scenario:
  - You’re building a small program for a **local café**. Every time a customer places an order, the system needs to:
    - **Calculate the total price** (including tax).
    - Format and print a receipt.
    - Repeat this for multiple customers.

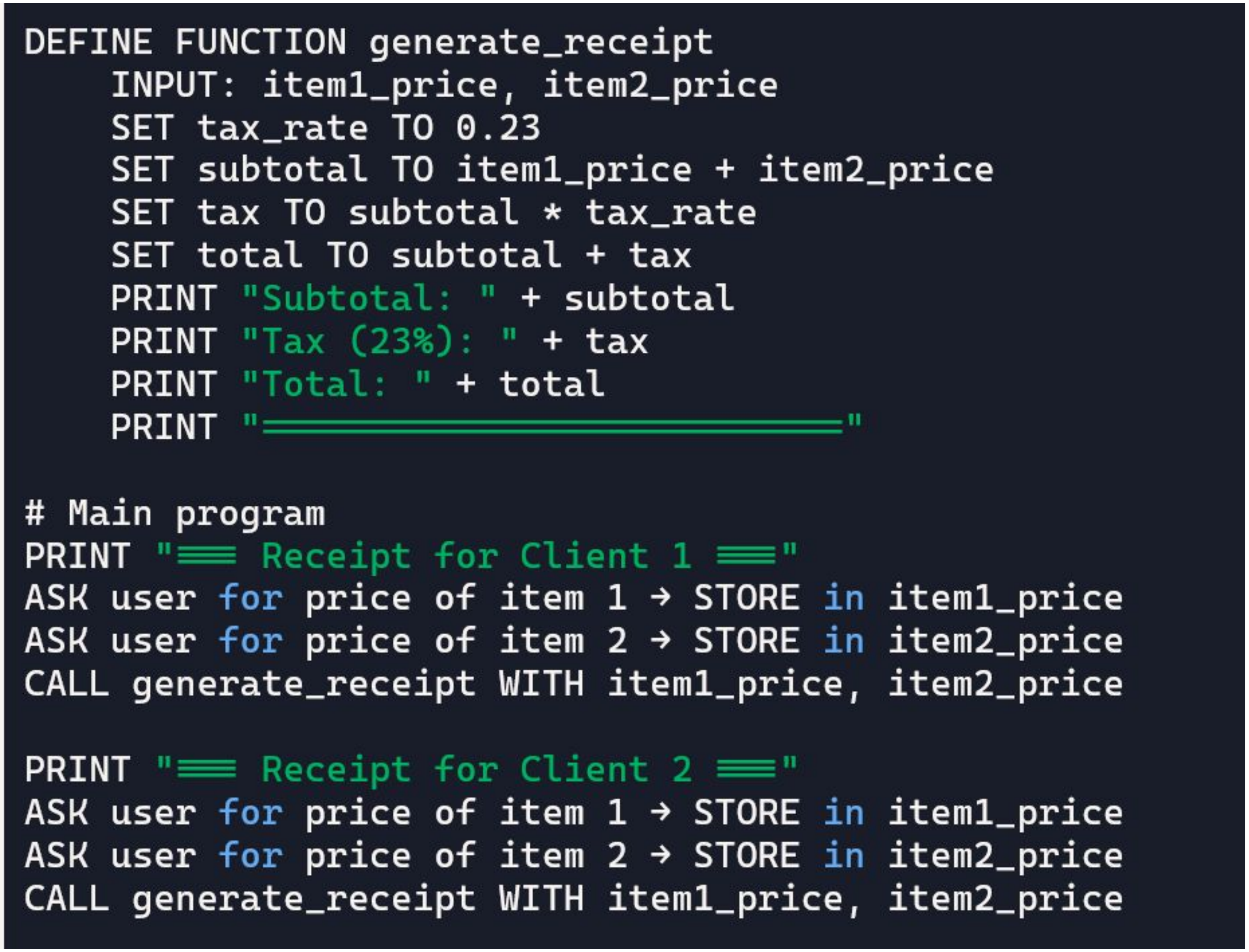
Solution 1 (using our current knowledge)



```
SET tax_rate TO 0.23

PRINT "=== Receipt for Client 1 ==="
ASK user for price of item 1 → STORE in item1_price
ASK user for price of item 2 → STORE in item2_price

SET subtotal1 TO item1_price + item2_price
SET tax1 TO subtotal1 * tax_rate
SET total1 TO subtotal1 + tax1

PRINT "Subtotal: " + subtotal1
PRINT "Tax (23%): " + tax1
PRINT "Total: " + total1
PRINT "==========================="

PRINT "=== Receipt for Client 2 ==="
ASK user for price of item 1 → STORE in item1_price
ASK user for price of item 2 → STORE in item2_price

SET subtotal2 TO item1_price + item2_price
SET tax2 TO subtotal2 * tax_rate
SET total2 TO subtotal2 + tax2

PRINT "Subtotal: " + subtotal2
PRINT "Tax (23%): " + tax2
PRINT "Total: " + total2
PRINT "==========================="
```

- Is this ok?
- Problems?

Solution 2



```
DEFINE FUNCTION generate_receipt
    INPUT: item1_price, item2_price
    SET tax_rate TO 0.23
    SET subtotal TO item1_price + item2_price
    SET tax TO subtotal * tax_rate
    SET total TO subtotal + tax
    PRINT "Subtotal: " + subtotal
    PRINT "Tax (23%): " + tax
    PRINT "Total: " + total
    PRINT "==========================="

# Main program
PRINT "=== Receipt for Client 1 ==="
ASK user for price of item 1 → STORE in item1_price
ASK user for price of item 2 → STORE in item2_price
CALL generate_receipt WITH item1_price, item2_price

PRINT "=== Receipt for Client 2 ==="
ASK user for price of item 1 → STORE in item1_price
ASK user for price of item 2 → STORE in item2_price
CALL generate_receipt WITH item1_price, item2_price
```

What we don’t know?

- How to define these new functions?
- How to use them?
- How they work internally?
- How they “communicate” with our main program?

FUNCTIONS IN PYTHON

Functions

- So far, we have been *using* the functions that are predefined in Python, such as:

```
name = input("Name? ")
print("Hello", name, "!")
root2 = math.sqrt(2)
```

Functions

- We may also **define** new functions of our own. For example:

```
def square(x):
    y = x**2
    return y
```

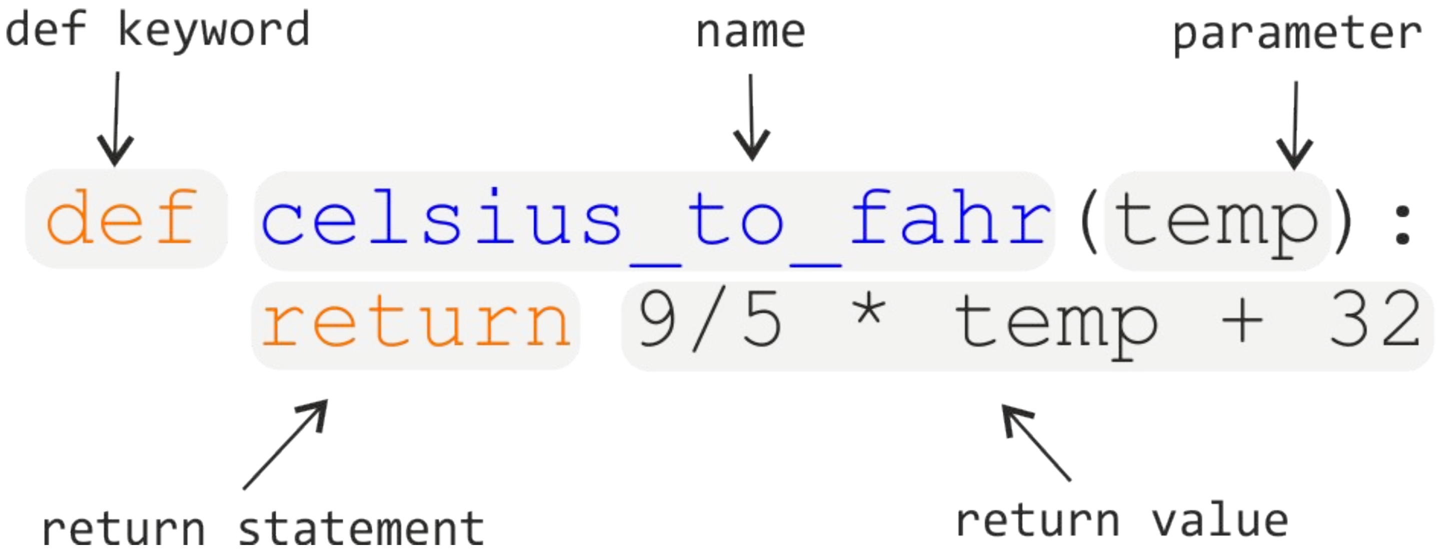
- After definition, we may **call** our function just like any other.

```
a = 10 + square(2)
b = square(a - 8)
x = 3
print(x,  square(1 - square(x-1)) + 1)
```

Play ▶️

Function definition

- A **function definition** specifies the name of a new function, a list of parameters, and a block of statements to execute when that function is called.



```
def celsius_to_fahr(temp):
    return 9/5 * temp + 32
```

Function definition

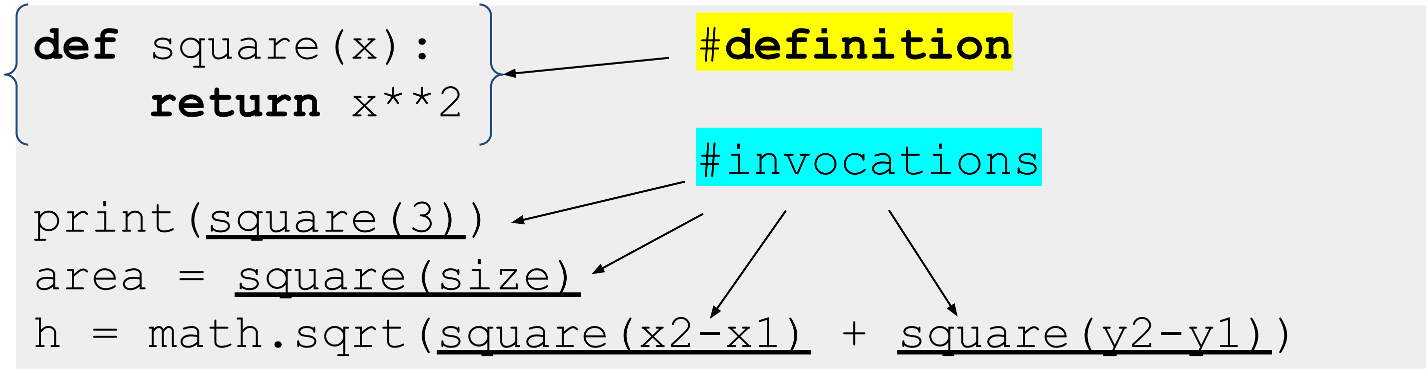
Syntax	Example

`def functionName(parameters):`
`    statements`
	`def hms2sec(h, m, s):`
	`    sec = (h*60+m)*60+s`
	`    return sec`

- The first line of the function definition is called the header
  - starts with the def keyword and ends with a colon.
- The indented block is called the body.
  - It has to be indented.
- Function names follow the same rules as variable names.

Definition vs. invocation

- Do not confuse function **definition** with function **invocation** (aka function call)!



```
def square(x):                #definition
    return x**2
                              #invocations
print(square(3))
area = square(size)
h = math.sqrt(square(x2-x1) + square(y2-y1))
```

- In a function **definition**, the statements in the body are not executed. They are **stored** for later use.
- They are **executed** only if and **when** the function is **invoked**.
- A function must be defined before being called.
- Define once, call as many times as needed.

Example

```
def hello():
    print("Hello!")

def helloTwice():
    hello()
    hello()

#calling the function
helloTwice()
```

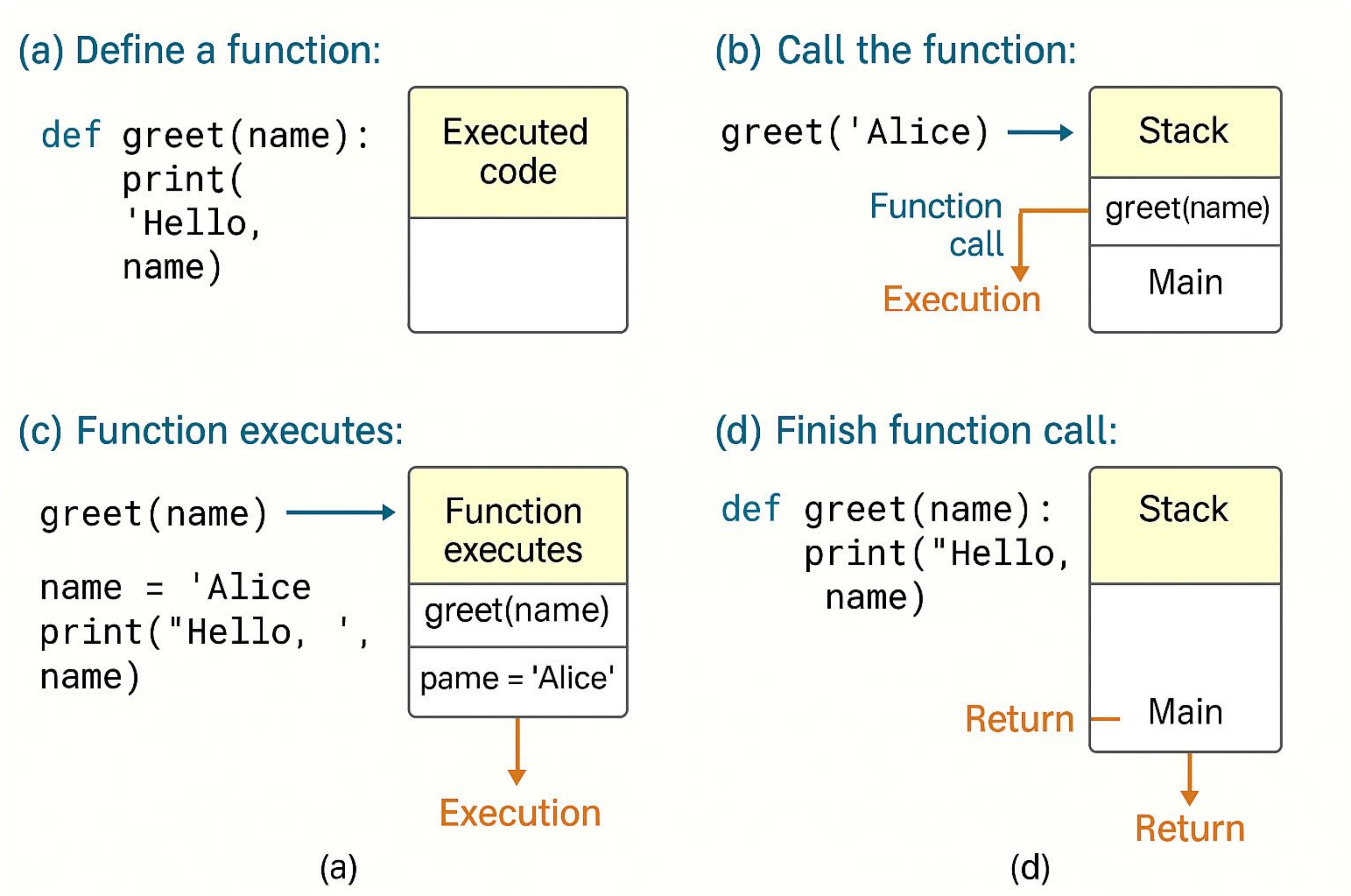
Play ▶

- This example contains two function definitions: `hello` and `helloTwice`.
- Then, `helloTwice` is called (invoked).
- When `helloTwice` runs, it calls `hello` twice.

Flow of execution

- Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.
- A **function definition** simply **stores the statements** in the function body for later use. The body is **not executed** at this time.
- A **function call** is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to **execute the body** of the function and then **returns** to pick up where it left off.
- Before executing the body, argument values are **assigned** to function parameters.

Execution of a function



```
def greet(name):
    print(
    'Hello,
    name)
```

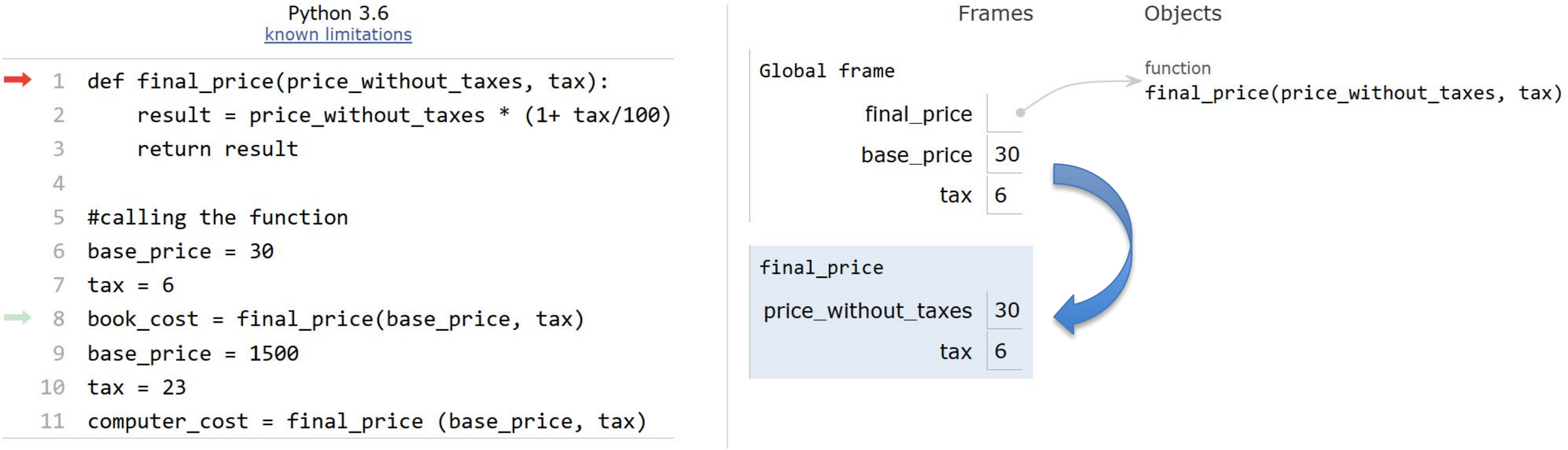
```
greet('Alice)
```

```
greet(name)

name = 'Alice
print("Hello, ',
name)
```

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

Execution of a function

- Phases of function execution:
  - stack frame creation, parameter passing, execution of block of code, and return result



Python 3.6
known limitations

```
def final_price(price_without_taxes, tax):
    result = price_without_taxes * (1+ tax/100)
    return result

#calling the function
base_price = 30
tax = 6
book_cost = final_price(base_price, tax)
base_price = 1500
tax = 23
computer_cost = final_price (base_price, tax)
```

Frames

Global frame

final_price	
base_price	30
tax	6

final_price

price_without_taxes	30
tax	6

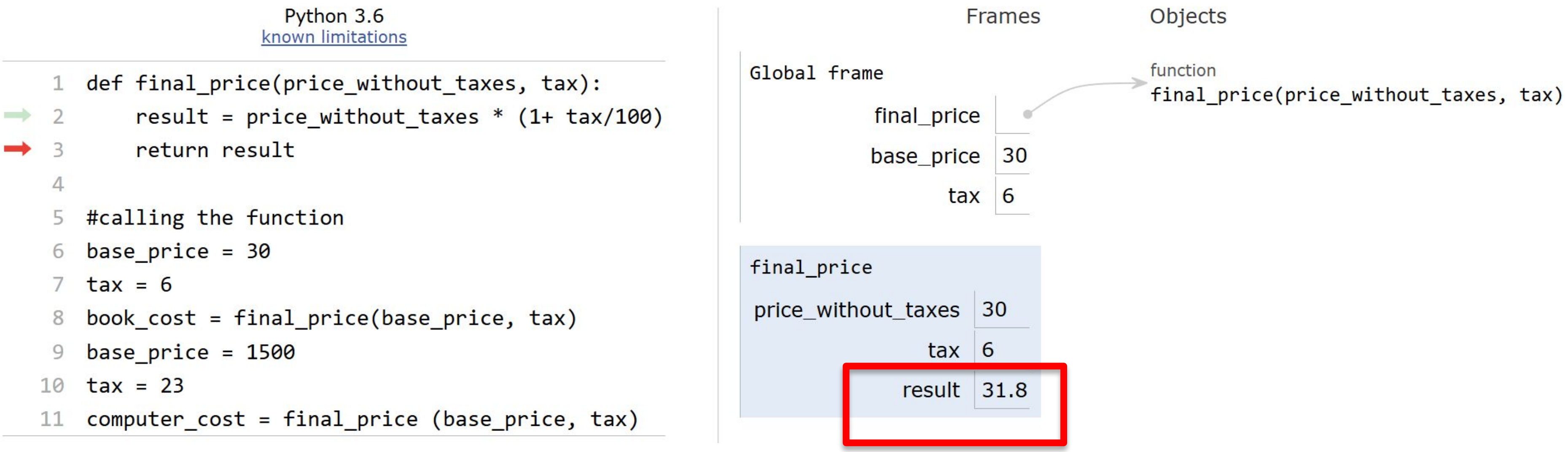
Objects

function
final_price(price_without_taxes, tax)

Play ▶

Execution of a function

- Phases of function execution:
  - stack frame creation, parameter passing, execution of block of code, and return result



Python 3.6
known limitations

```
def final_price(price_without_taxes, tax):
    result = price_without_taxes * (1+ tax/100)
    return result

#calling the function
base_price = 30
tax = 6
book_cost = final_price(base_price, tax)
base_price = 1500
tax = 23
computer_cost = final_price (base_price, tax)
```

Frames

Global frame

final_price	
base_price	30
tax	6

final_price

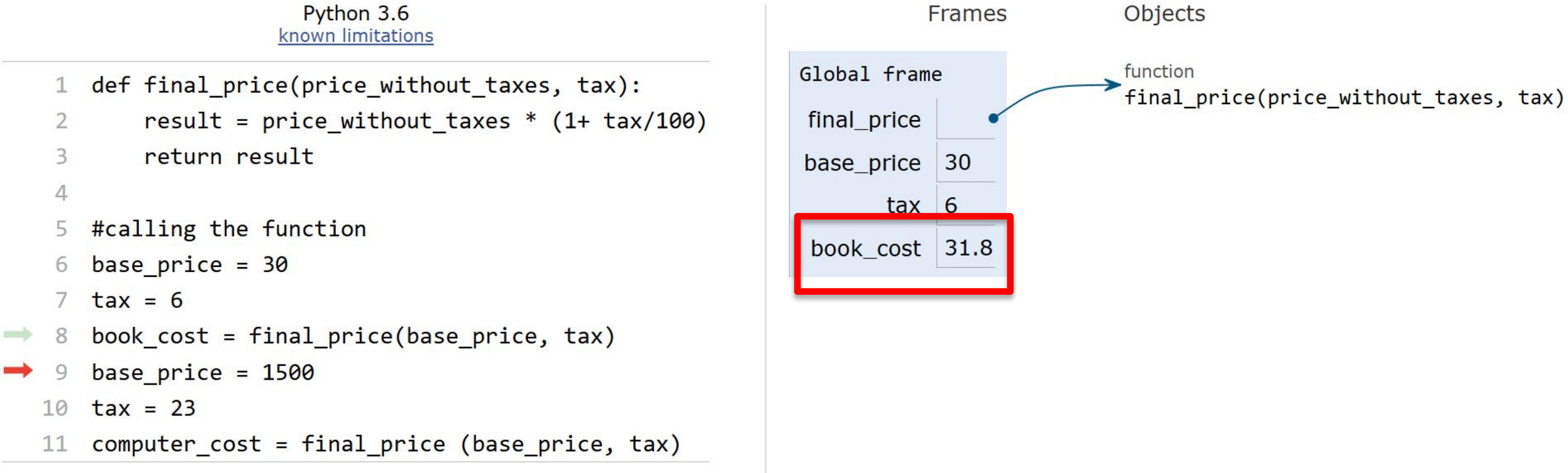
price_without_taxes	30
tax	6
result	31.8

Objects

function
final_price(price_without_taxes, tax)

Execution of a function

- Phases of function execution:
  - stack frame creation, parameter passing, execution of block of code, and return result



```
def final_price(price_without_taxes, tax):
    result = price_without_taxes * (1+ tax/100)
    return result

#calling the function
base_price = 30
tax = 6
book_cost = final_price(base_price, tax)
base_price = 1500
tax = 23
computer_cost = final_price (base_price, tax)
```

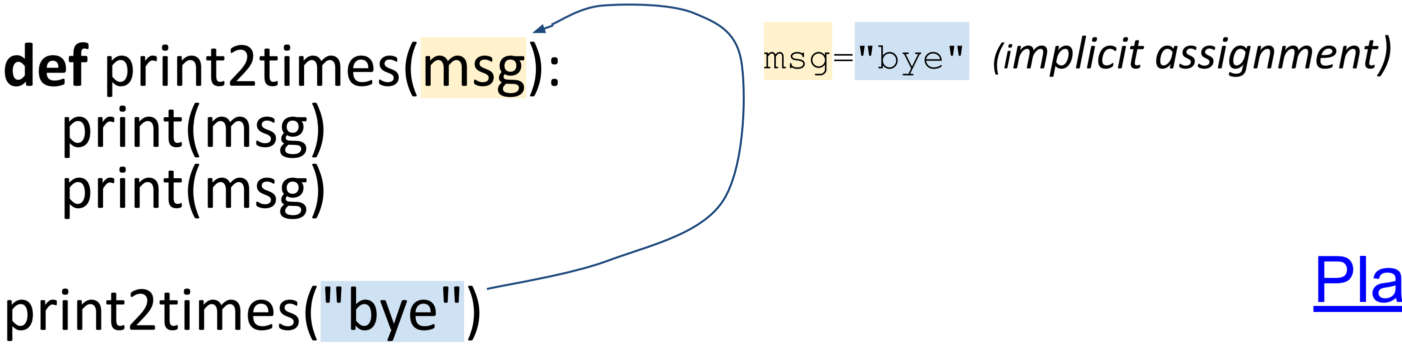
Global frame	
final_price	
base_price	30
tax	6
book_cost	31.8

function
final_price(price_without_taxes, tax)

Play ▶️

Parameters and arguments

- Some functions require arguments.
  - For example, when you call math.sin() you pass a number as an argument.
- Some functions take more than one argument:
  - math.pow() takes two, the base and the exponent.
- When a function is called, the arguments are *values* assigned to *variables* called parameters in the definition.



```
def print2times(msg):
    print(msg)
    print(msg)

print2times("bye")
```

`msg="bye"` *(implicit assignment)*

Play ▶

Parameters are local variables

- Parameters are local variables, too.
- You may modify parameters, but the effect is local!

```
def double(x):
    x *= 2       # you may modify parameters
    return x

x = 3
y = double(x)    # <=> double(3)
print(x, y)      # What's the value of x and y?
```

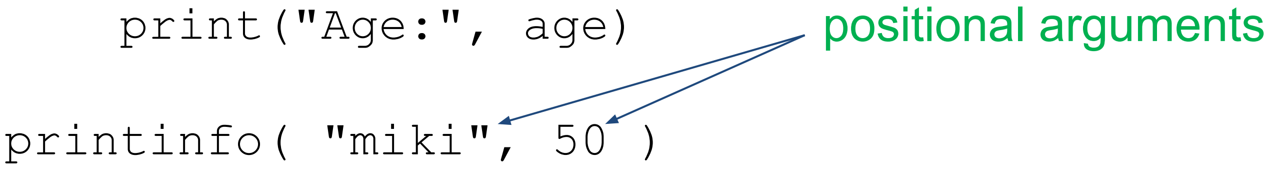
Play ▶

- When the function is called, the parameter receives (just) the *value* of the argument.
- This form of argument passing is called *pass by value*.

Positional and keyword arguments

- In a function call, **positional arguments** are assigned to parameters according to their position.

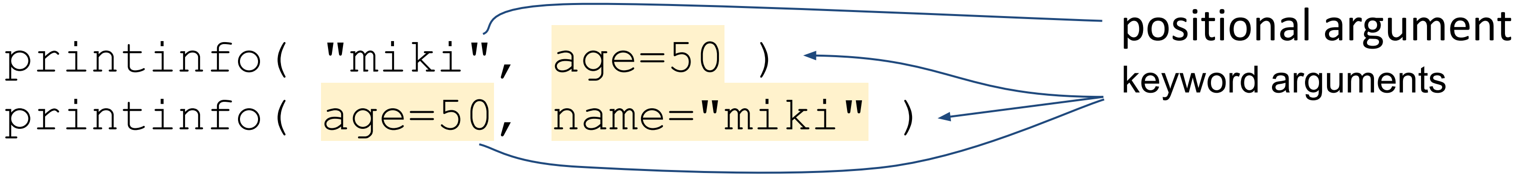
```
def printinfo( name, age ):
    print("Name:", name)
    print("Age:", age)
```



```
printinfo( "miki", 50 )
```

positional arguments

- With **keyword arguments**, the values are assigned to parameters identified by name.



```
printinfo( "miki", age=50 )
printinfo( age=50, name="miki" )
```

positional argument

keyword arguments

- With keyword arguments you don’t have to remember the order of parameters, just their names.
- When mixed, positional must precede keyword arguments.

Return values

- Some functions, such as `abs` or `math.sin`, produce results, which may be used in expressions or stored in variables.
- Other functions, like `print`, perform an action but don’t return a value.
  - They are called void functions.
  - *Actually, they return the special value `None`.*
- The *return statement* can only be used inside a function.

```
return expression
```

- When executed, it exits the function and returns the value of the expression to wherever the function was called from.
- A return statement with no expression ⇔ **`return`** `None`
- If execution reaches the end of the body ⇔ **`return`** `None`

Global vs. local variables

- Variables defined inside a function have a *local scope*.
  Local variables are accessible and changeable only inside their function.
- Variables defined outside functions have a *global scope*.
  Global variables are accessible everywhere.
- But when you *assign* to a name inside a function, you create a new local variable even if an identical global name exists.
  In summary: local names mask global names.

```
def add(a, b):
    total = a + b   # Here total is local variable
    print("Inside:", total)
    return total

total = 0             # This is a global variable
print(add(10, 20))    # Call add function
print("Outside:", total)
print(a, b)          # ERROR!
```

Play ▶

Default argument values

- A function definition may specify **default argument values** for some of its parameters.

```
def printinfo(name, age=35):
    print("Name:", name)
    print("Age:", age)
```

- When calling the function, if a value is not provided for that argument, it takes the default value.

```
printinfo("miki", 50)
printinfo("miki")          # here, age is 35!
printinfo(name="miki")     # same here
```

- This is used for optional arguments in some functions.

```
print(1, 2, 3)
print(1, 2, 3, sep='->')
print(1, 2, 3, sep='->', end='\n-FIM-\n')
```

Variable-length arguments

- You can define a function to accept a variable number of arguments.
- These so-called variable-length arguments are assigned to a special parameter in the function definition.

```
def printinfo(arg1, *args):
    print(arg1)
    for var in args:
        print(var)

printinfo(70, 60, 50)  #the last two are passed as a tuple
printinfo(10)
```

arg1=70

args=(60, 50)

- The asterisk (*) indicates the parameter that receives the values of all (positional) variable arguments.
- Advanced topic. Not required.

Anonymous functions / Lambda expressions

- A ***lambda expression*** is an expression whose result is a function.
- **Usage:** `lambda parameters : operation`
- You may store it in a variable and use it later, for example.

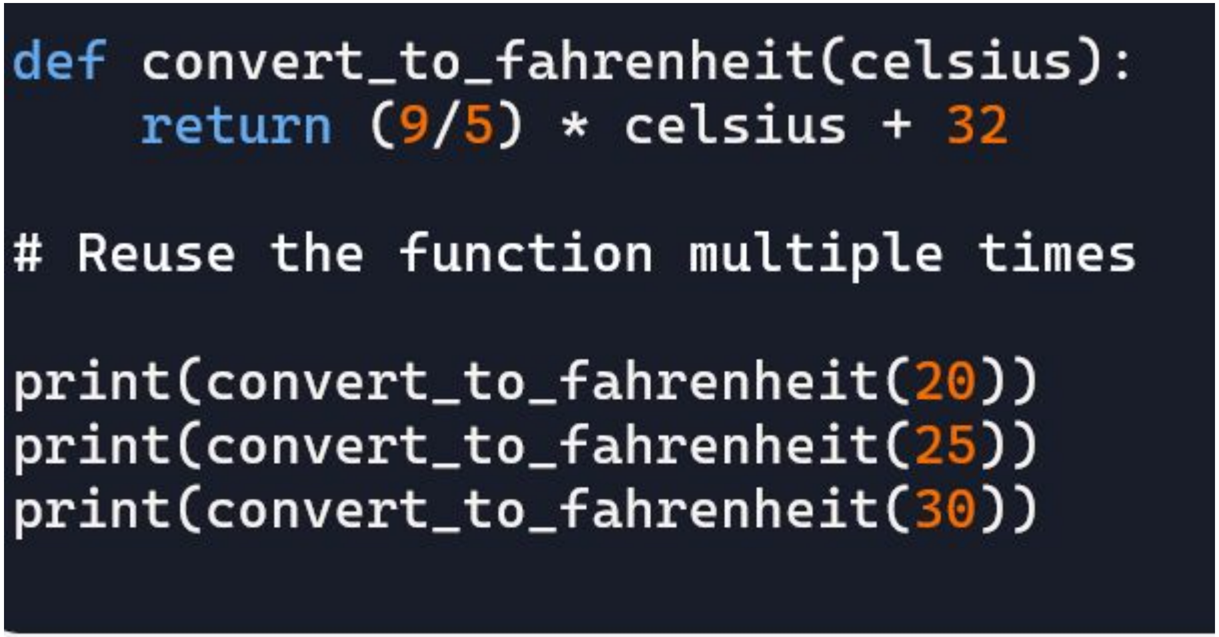
```
add = lambda a, b: a + b ← # lambda expression

# Now you can call add as a function
print("Total: ", add(10, 20))   # Total: 30
```

- They're also known as *anonymous functions*.
- They cannot contain statements, only a single expression.
- They're most useful to pass functions as arguments to other functions.
  - *We'll see examples later in the course*

TYPICAL USES OF FUNCTIONS

🔁 Avoiding Repetition

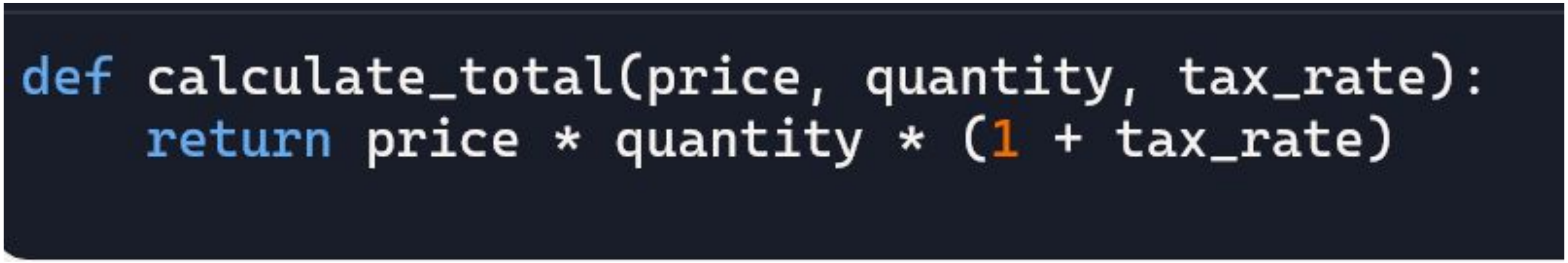
- **Use case:** When the same logic appears multiple times.
- **Examples:**
  - Converting temperatures, calculating tax, formatting dates.



```python
def convert_to_fahrenheit(celsius):
    return (9/5) * celsius + 32

# Reuse the function multiple times

print(convert_to_fahrenheit(20))
print(convert_to_fahrenheit(25))
print(convert_to_fahrenheit(30))
```

📦 Encapsulating Logic

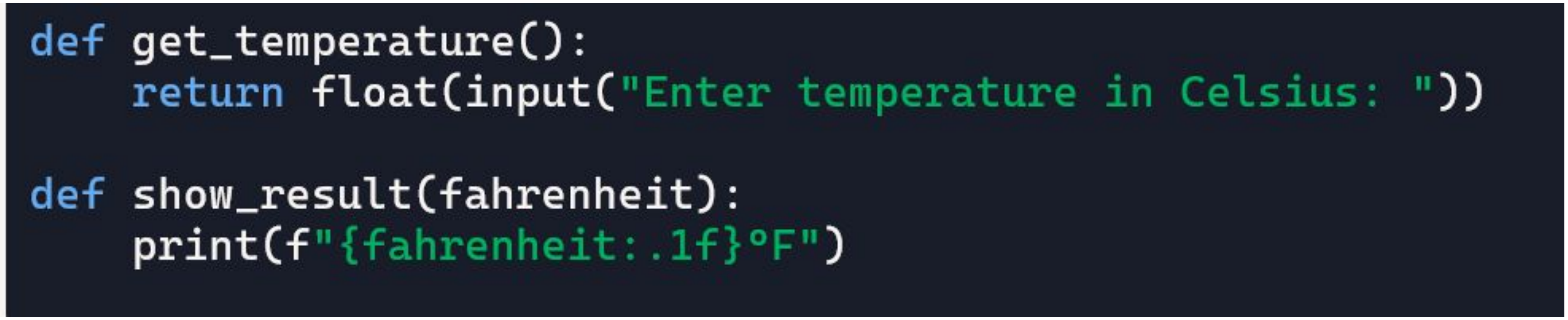
- **Use case:** Wrapping complex operations into a single, reusable unit.
- **Example:**
  - Calculating invoice totals, validating user input, processing text.



```python
def calculate_total(price, quantity, tax_rate):
    return price * quantity * (1 + tax_rate)
```

📥 Handling Input and Output

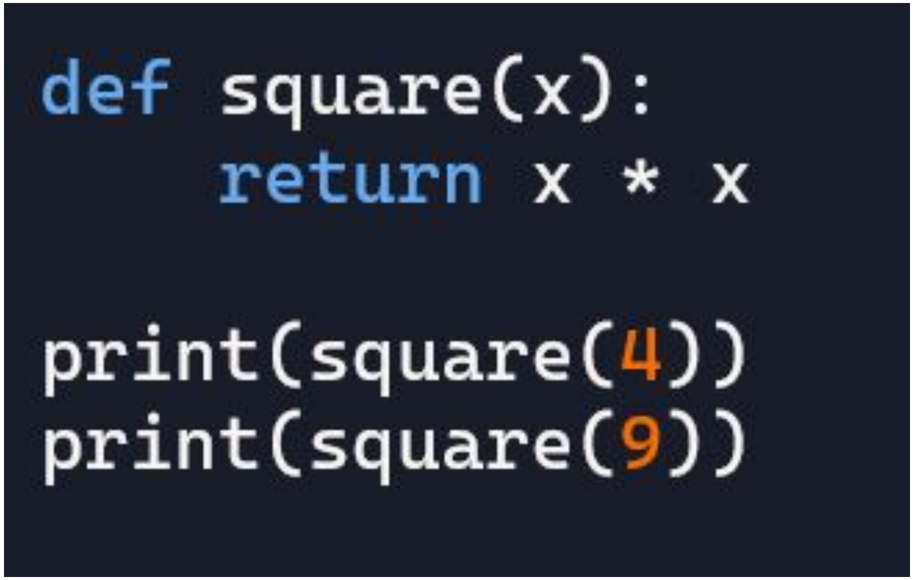
- **Use case:** Separating user interaction from processing logic.
- **Example:**
  - Reading values, displaying formatted results.



```python
def get_temperature():
    return float(input("Enter temperature in Celsius: "))

def show_result(fahrenheit):
    print(f"{fahrenheit:.1f}°F")
```

🧮 Performing Calculations

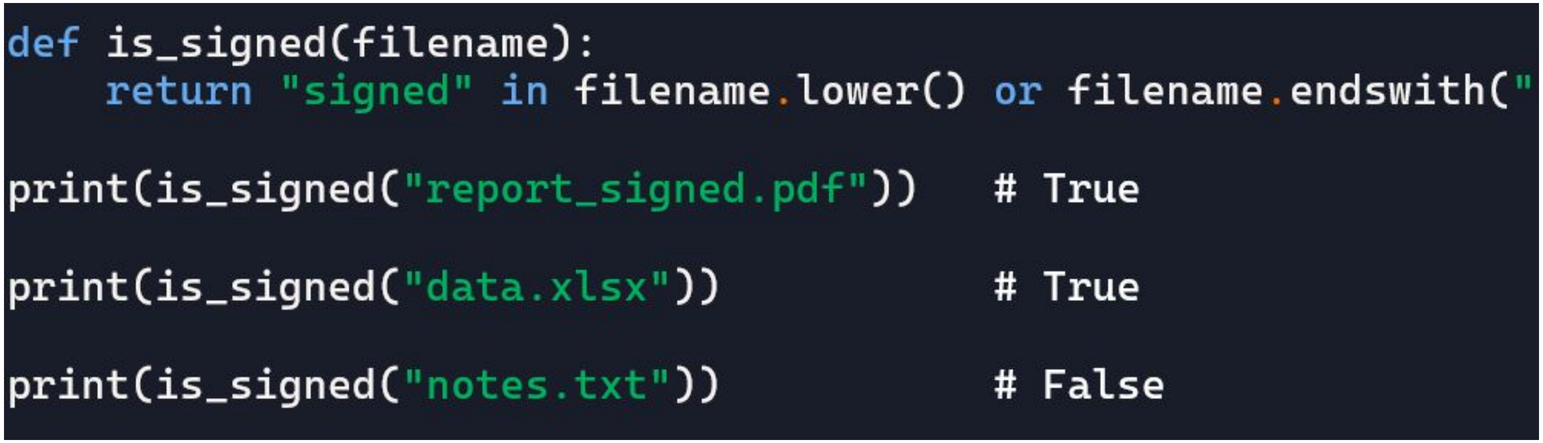
- **Use case:** Mathematical operations, statistics, or aggregations.
- **Example:**
  - Summing a list, computing averages, applying formulas.



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

print(square(4))
print(square(9))
```

🔍 Filtering or Transforming Data

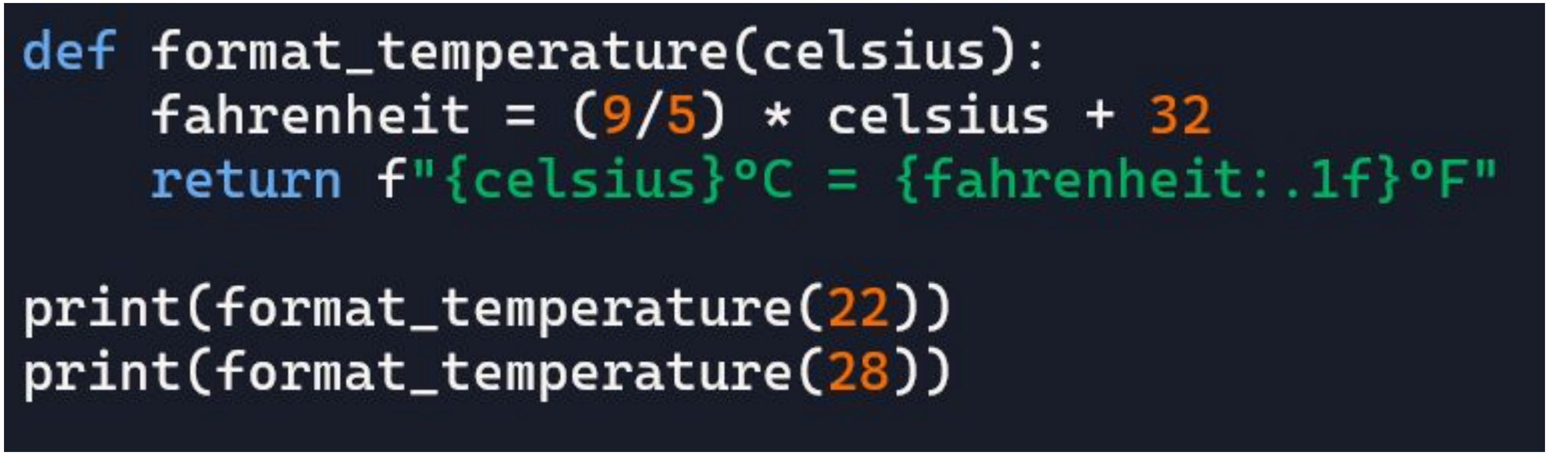
- **Use case:** Cleaning, modifying, or selecting data from a collection.
- **Example:**
  - Filtering filenames, transforming strings, extracting keywords.



```python
def is_signed(filename):
    return "signed" in filename.lower() or filename.endswith("

print(is_signed("report_signed.pdf"))   # True

print(is_signed("data.xlsx"))           # True

print(is_signed("notes.txt"))           # False
```

🧰 Organizing Code for Reuse

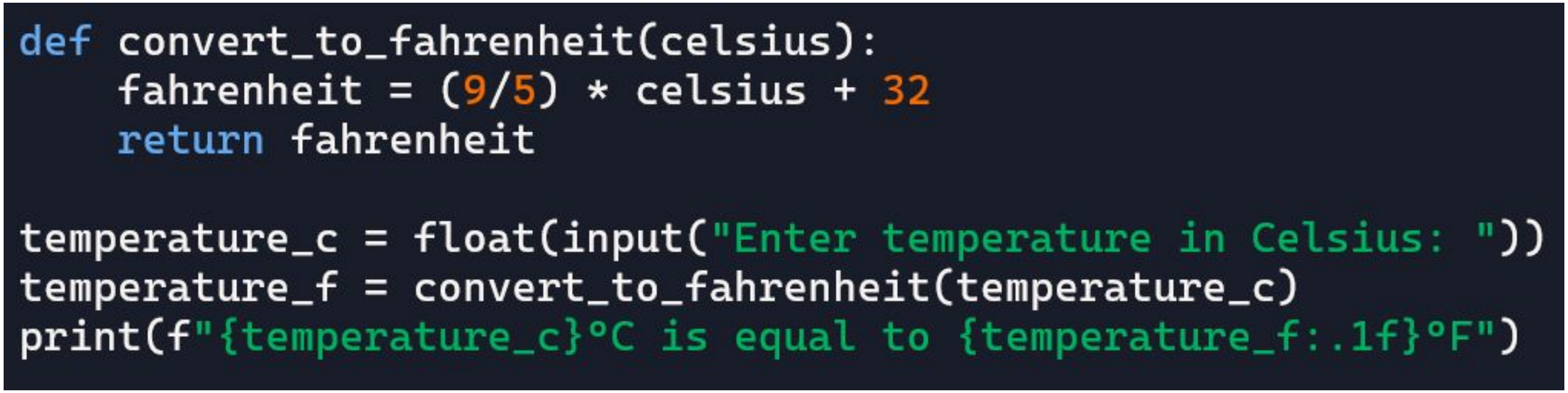
- **Use case:** Creating modules, libraries, or helper utilities.
- **Example:**
  - A file with reusable functions for string formatting or date handling.
    - For example utils.py



```
def format_temperature(celsius):
    fahrenheit = (9/5) * celsius + 32
    return f"{celsius}°C = {fahrenheit:.1f}°F"

print(format_temperature(22))
print(format_temperature(28))
```

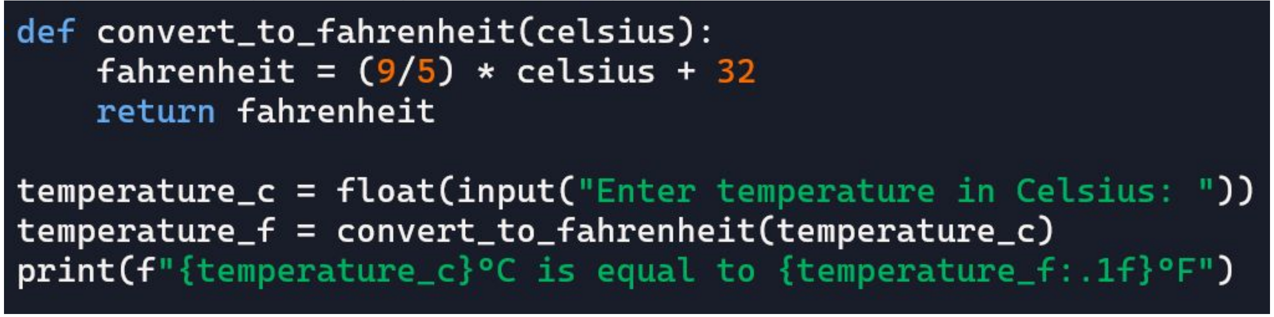
Example: Celsius to Fahrenheit Converter

- Let's look in more detail to our convert_to_Fahrenheit function …



```python
def convert_to_fahrenheit(celsius):
    fahrenheit = (9/5) * celsius + 32
    return fahrenheit

temperature_c = float(input("Enter temperature in Celsius: "))
temperature_f = convert_to_fahrenheit(temperature_c)
print(f"{temperature_c}°C is equal to {temperature_f:.1f}°F")
```

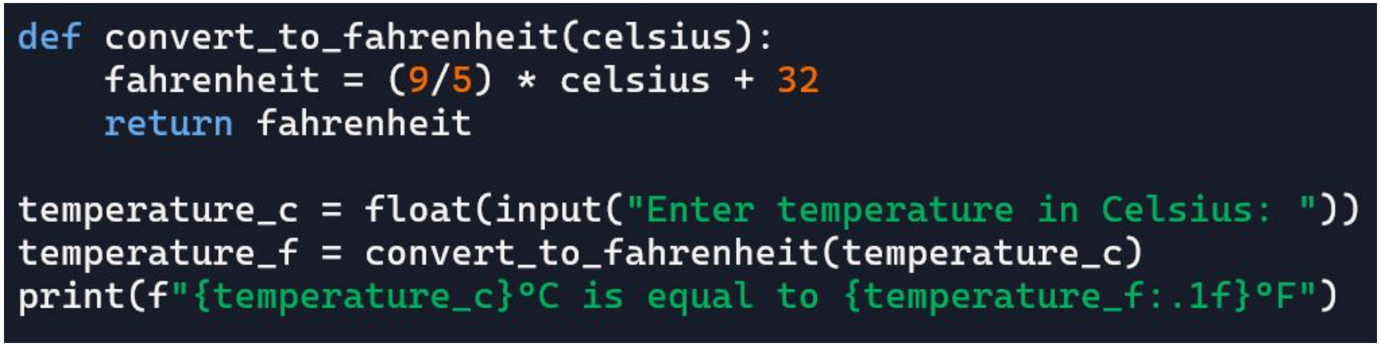
Step-by-Step Breakdown



```python
def convert_to_fahrenheit(celsius):
    fahrenheit = (9/5) * celsius + 32
    return fahrenheit

temperature_c = float(input("Enter temperature in Celsius: "))
temperature_f = convert_to_fahrenheit(temperature_c)
print(f"{temperature_c}°C is equal to {temperature_f:.1f}°F")
```

- **def convert_to_fahrenheit(celsius):**
  - **def** is the keyword that defines a function.
  - **convert_to_fahrenheit** is the name of the function.
    - You choose this name to describe what the function does.
  - **(celsius)** is a parameter.
    - It's a placeholder for the value you'll pass into the function when you use it.
  - **fahrenheit = (9/5) * celsius + 32** line performs the actual conversion using the standard formula.
  - **return fahrenheit,** tells Python to **send back** the result of the calculation to wherever the function was called.

Step-by-Step Breakdown (cont.)



```
def convert_to_fahrenheit(celsius):
    fahrenheit = (9/5) * celsius + 32
    return fahrenheit

temperature_c = float(input("Enter temperature in Celsius: "))
temperature_f = convert_to_fahrenheit(temperature_c)
print(f"{temperature_c}°C is equal to {temperature_f:.1f}°F")
```

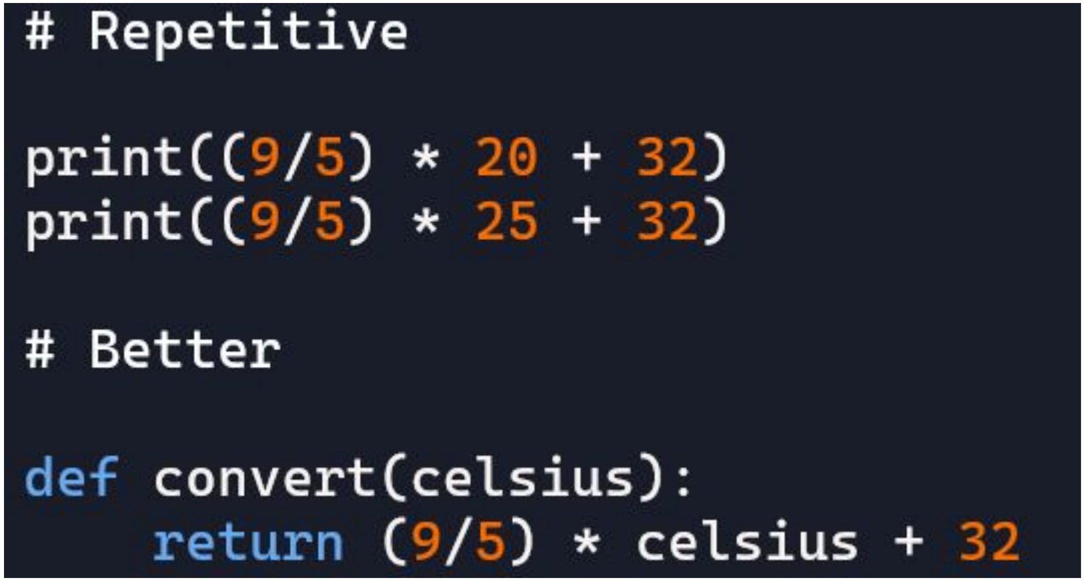
- **temperature_c = float(input("Enter temperature in Celsius: "))**
  - This line asks the user to enter a temperature.
  - input() gets the value as a string, and float() converts it to a number with decimals.

- **temperature_f = convert_to_fahrenheit(temperature_c)**
  - This is where the function is **called**.
  - You pass temperature_c into the function, and it returns the converted value.
  - That result is stored in temperature_f.

- **print(f"{temperature_c}°C is equal to {temperature_f:.1f}°F")**
  - This prints the final result using an **f-string**

COMMON MISTAKES

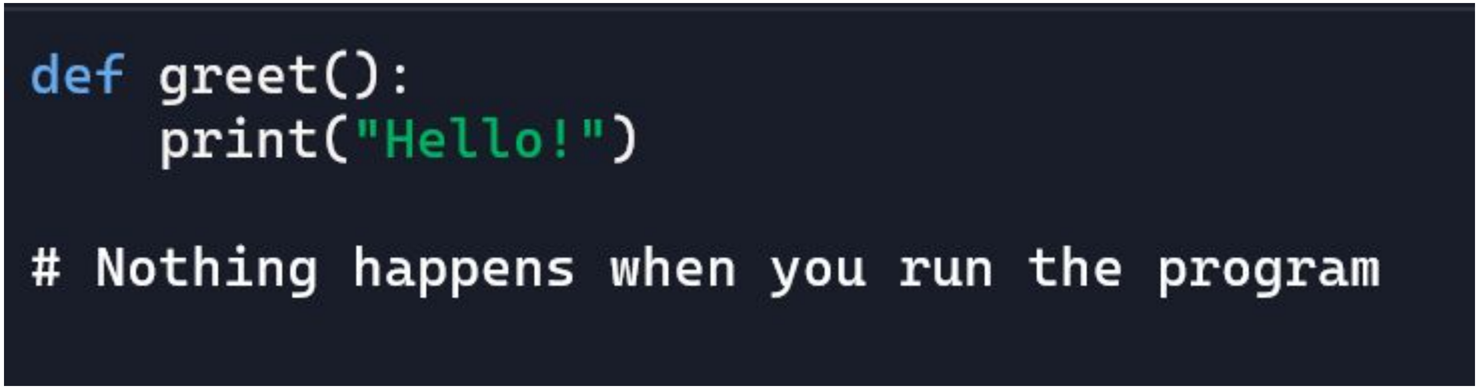
# ❌ #1 — Not Using Functions at All



```python
# Repetitive

print((9/5) * 20 + 32)
print((9/5) * 25 + 32)

# Better

def convert(celsius):
    return (9/5) * celsius + 32
```

- Problem: Repeating code blocks manually
- Fix: Identify repeated logic and wrap it in a function

# ❌ #2 — Defining but Not Calling

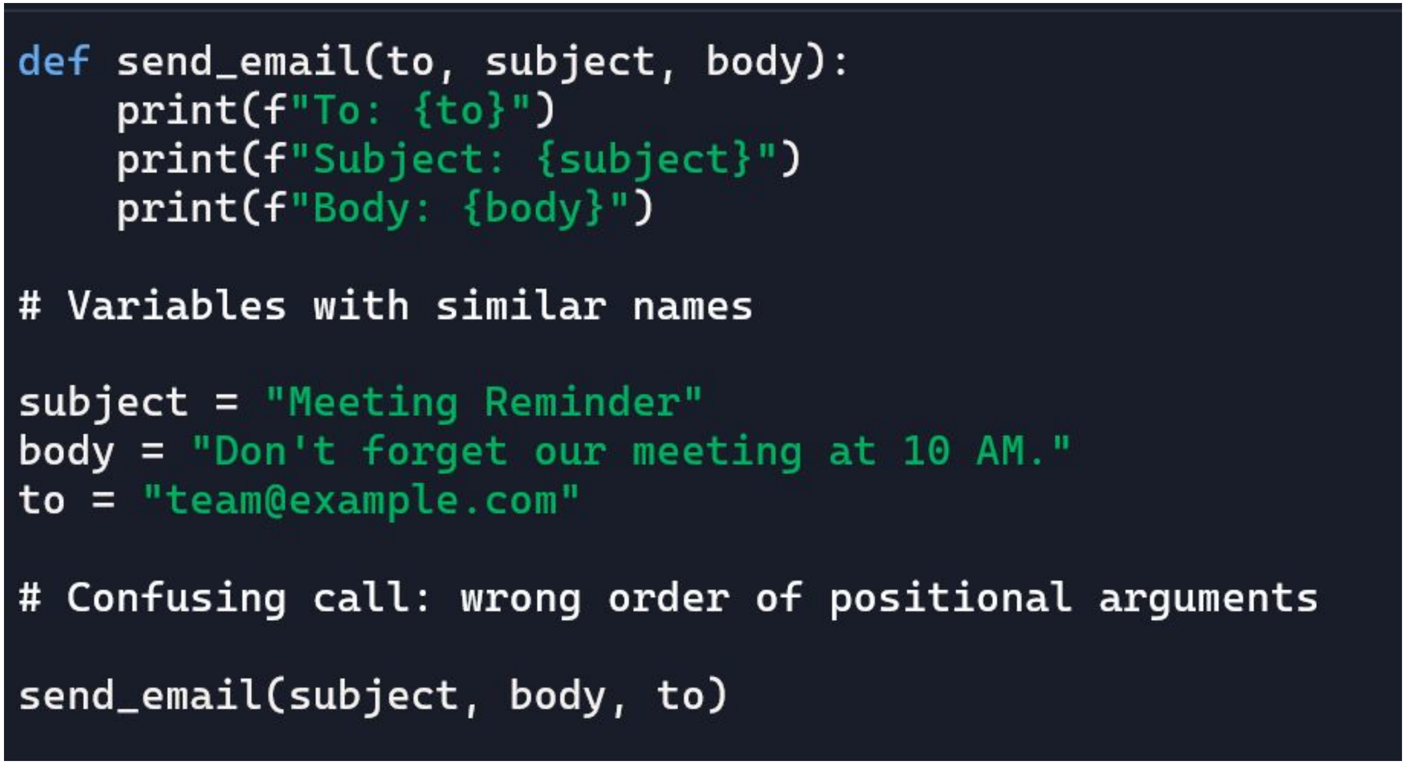


```python
def greet():
    print("Hello!")

# Nothing happens when you run the program
```

- Problem: Function is written but never used

- Fix: Always call the function to activate it

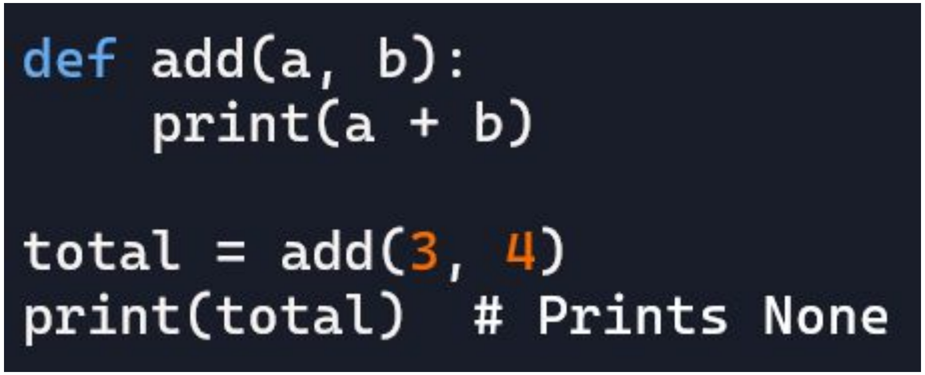
# ❌ #3 — Confusing Parameters and Arguments



```python
def send_email(to, subject, body):
    print(f"To: {to}")
    print(f"Subject: {subject}")
    print(f"Body: {body}")

# Variables with similar names

subject = "Meeting Reminder"
body = "Don't forget our meeting at 10 AM."
to = "team@example.com"

# Confusing call: wrong order of positional arguments

send_email(subject, body, to)
```

- Problem:
  - The function expects: to, subject, body
  - But the call passes: subject, body, to — wrong order!
- Fix:
  - Respect order of parameters
  - Or Use **keyword arguments** to make the call explicit

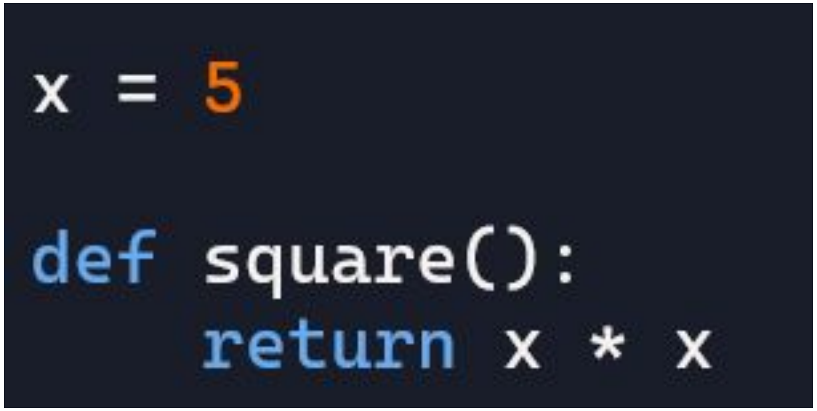
# ❌ #4 — Forgetting to Return Values



```python
def add(a, b):
    print(a + b)

total = add(3, 4)
print(total)  # Prints None
```

- Problem: Function prints but doesn’t return
- Fix:
  - Use return to send back results
  - And don’t print inside the function

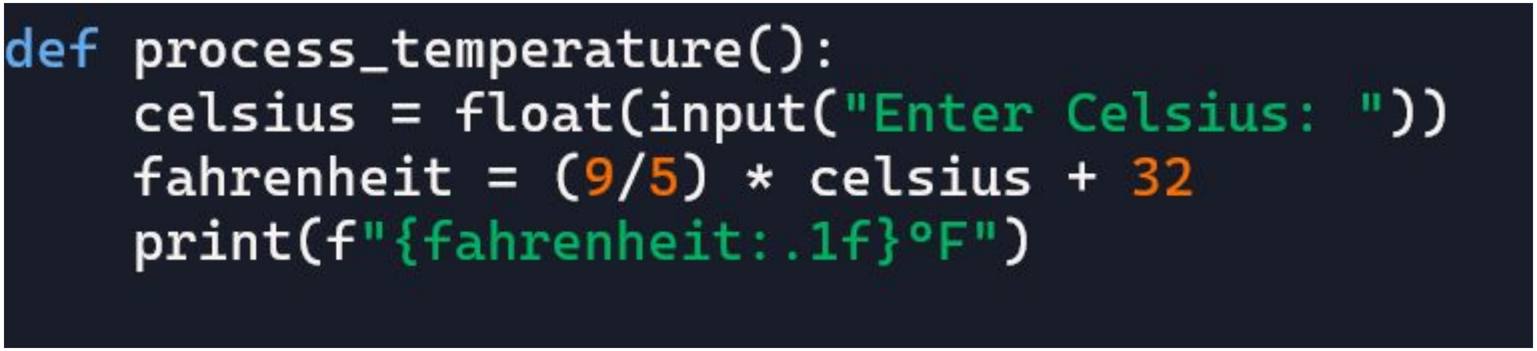
# ❌ #5 — Using Global Variables Instead of Parameters



```python
x = 5

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

- Problem: Function depends on external variables
- Fix: Pass all needed data as parameters

# ❌ #6 — Overcomplicating Functions



```python
def process_temperature():
    celsius = float(input("Enter Celsius: "))
    fahrenheit = (9/5) * celsius + 32
    print(f"{fahrenheit:.1f}°F")
```

- Problem: One function does too much
- This function will be easy to reuse or test?
  - What if we already have the value for celsius?
  - What if we want Fahrenheit for further calculations and not to be shown ?
  - Testing implies user actions ☹
- Fix: Use single responsibility principle
  - Process function only processes

Single Responsibility Principle (SRP)

- **A function should do one thing and do it well.**

- When defining a function:
  - **Avoid mixing concerns**:
    - Don't include input/output operations (like print() or input()) inside a function that’s meant to process data.
  - **Keep it pure**:
    - The function should take arguments, perform its computation, and return a result—without side effects like reading from or writing to the console, files, or network.

- Why This Matters
  - Reusability:
    - A pure processing function can be reused in different contexts—whether the data comes from a file, user input, or another function.
  - Testability:
    - It’s easier to write unit tests for functions that don’t depend on external input/output.
  - Composability:
    - You can combine processing functions more flexibly when they don’t have embedded I/O logic.

Discussion

- Why functions (and the modularity they provide) are important in real-world projects?

Modularity and Functions Matter in Real-World Projects

- **Clarity & Focus**:
  - Each function does one job—simplifies understanding and debugging.
- **Reusability**:
  - Write once, use anywhere—saves time and effort.
- **Testability**:
  - Isolated logic makes unit testing straightforward.
- **Maintainability**:
  - Easier updates without breaking other parts.
- **Scalability**:
  - Modular code grows with your project.
- **Team Collaboration**:
  - Clear boundaries enable parallel development.
- **Cognitive Ease**:
  - Developers focus on one piece at a time.

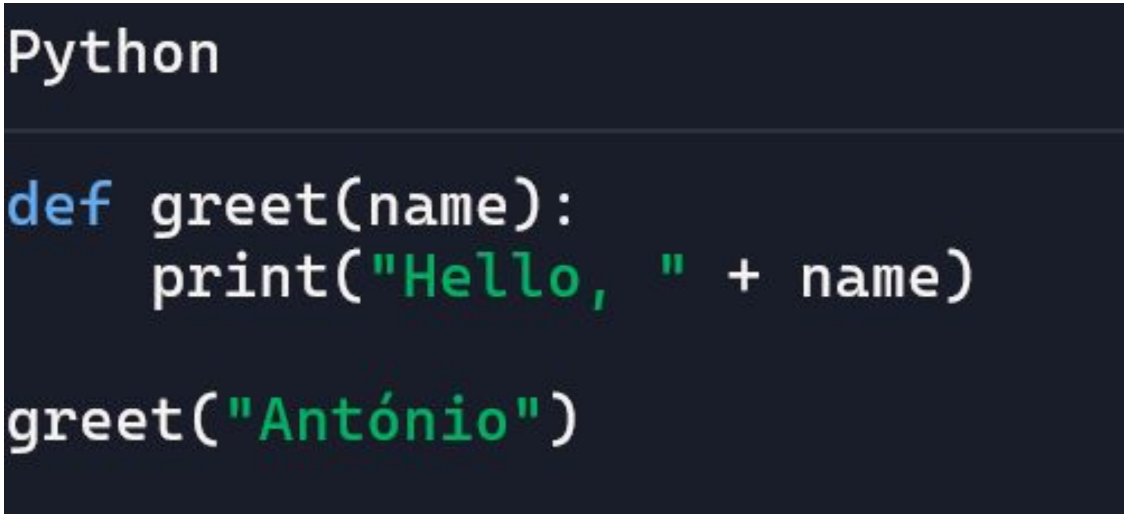
- Modular design turns complexity into manageable building blocks.

Tips

- Use small, focused functions
- Reuse and clarity is very important
- Practice with real-world examples
- Debug with print statements and test cases

FUNCTIONS IN OTHER LANGUAGES

🐍 Python

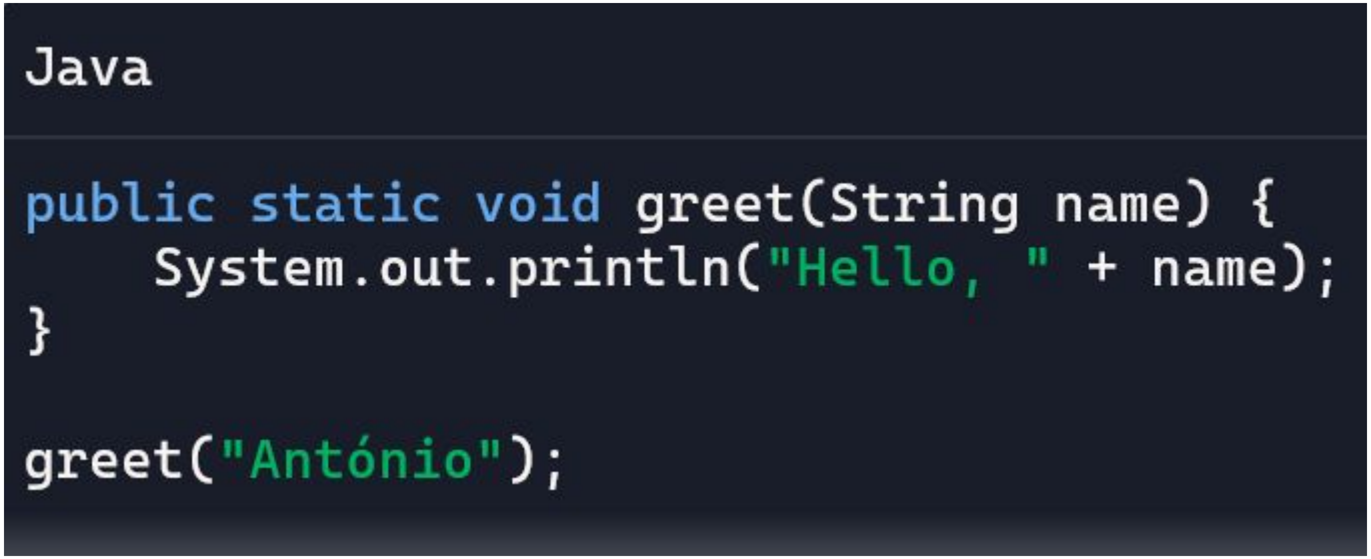


Python

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

greet("António")
```

- **Indentation-based blocks** (no braces).
- **Functions are first-class objects** (can be passed around).
  - More on this topic later…
- **No type declarations** (unless using type hints).

☕ Java

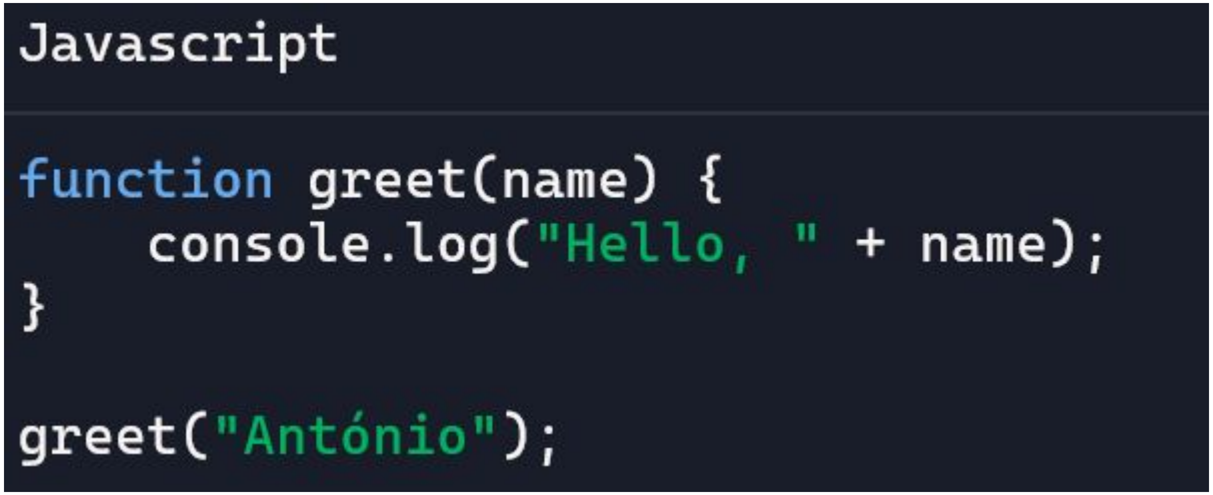


Java

```java
public static void greet(String name) {
    System.out.println("Hello, " + name);
}

greet("António");
```

- Requires **explicit type declarations**.
- Uses **braces** **{}** for blocks.
- Must be inside a **class**.
- static needed for calling without an object.

🌐 JavaScript

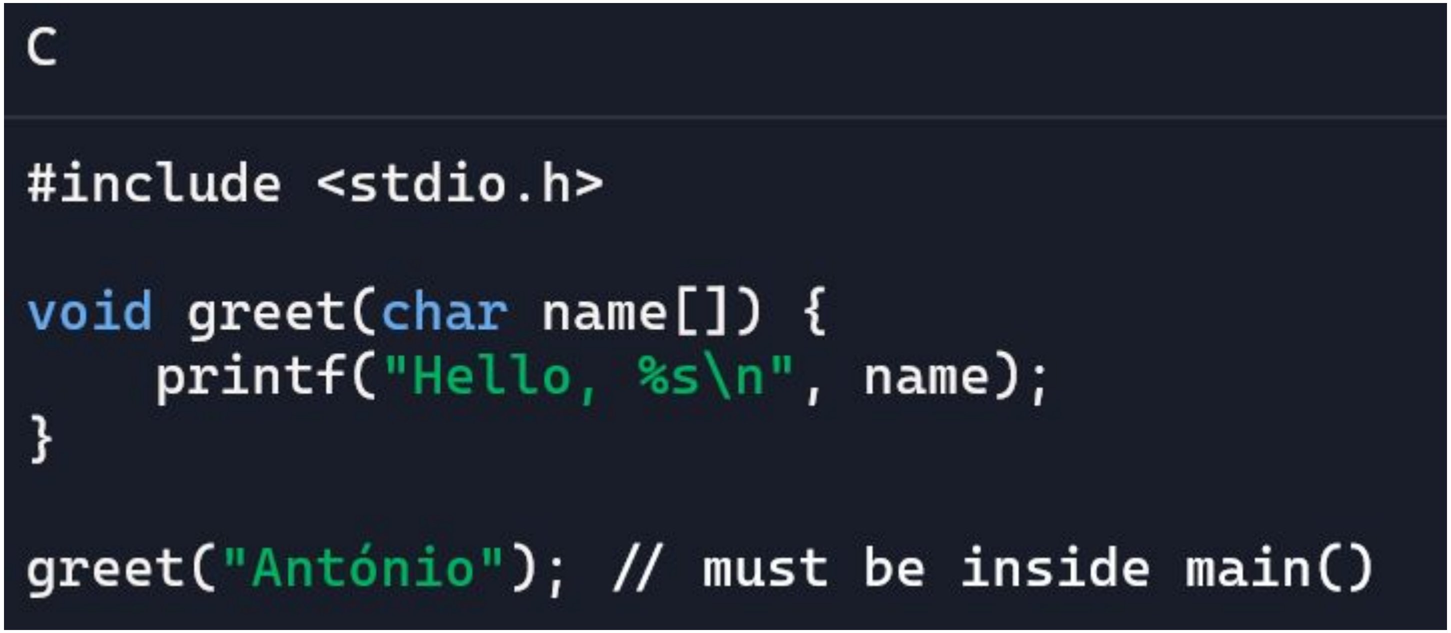


Javascript

```javascript
function greet(name) {
    console.log("Hello, " + name);
}

greet("António");
```

- Flexible with **types** (dynamic typing).
- Supports **anonymous functions** and **arrow functions**.
- Functions can be nested or assigned to variables.

🧬 C

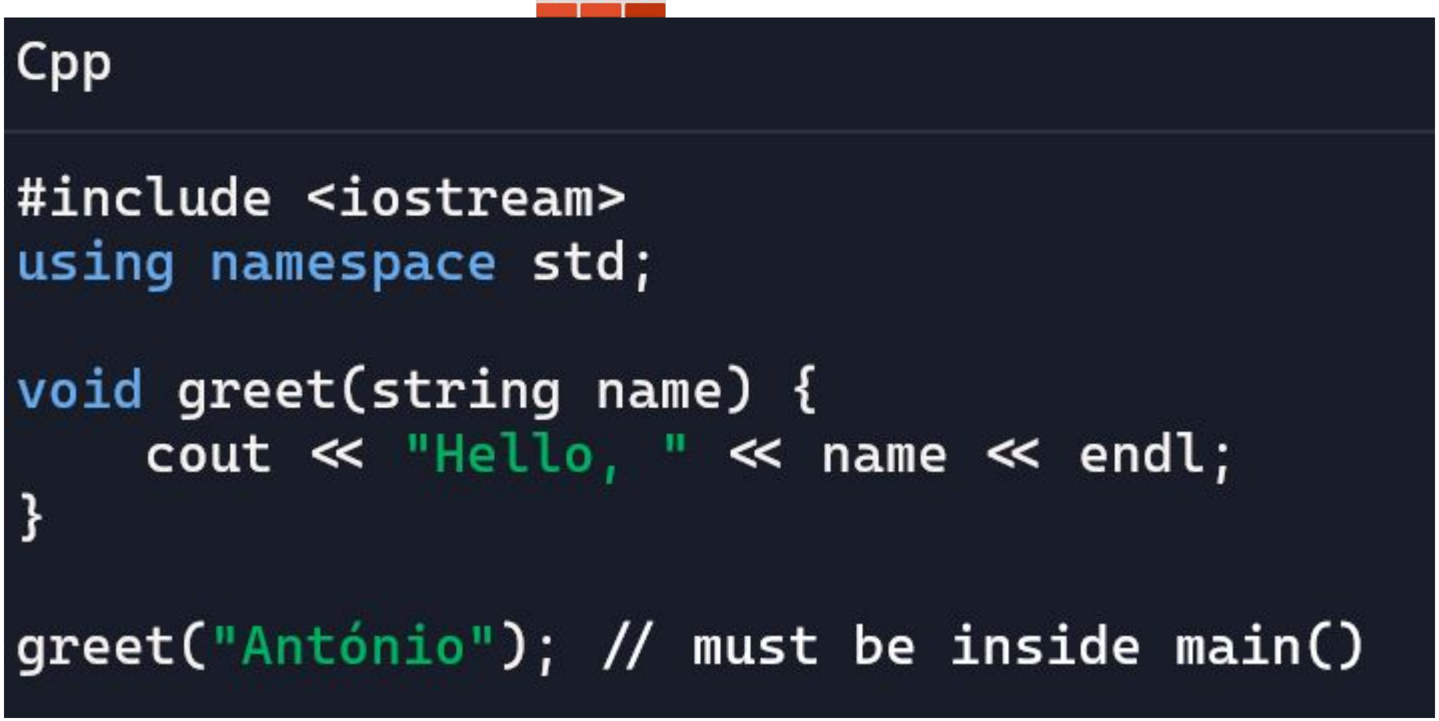


```
C
```

```c
#include <stdio.h>

void greet(char name[]) {
    printf("Hello, %s\n", name);
}

greet("António"); // must be inside main()
```

- Requires **type declarations** for parameters and return.
- No native string type—uses char[].
- Functions must be declared **outside** **main()** and called from within.

🧱 C++

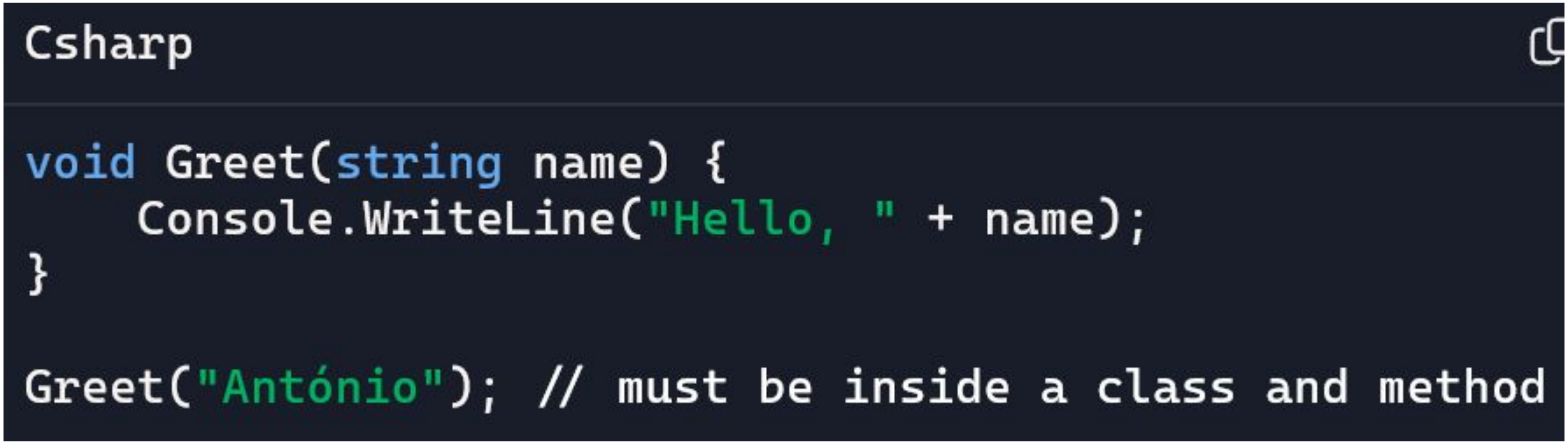


Cpp

```cpp
#include <iostream>
using namespace std;

void greet(string name) {
    cout << "Hello, " << name << endl;
}

greet("António"); // must be inside main()
```

- Similar to C, but supports `string` and **function overloading**.
- Can use **default arguments**.
- Must be called from within a function like `main()`.

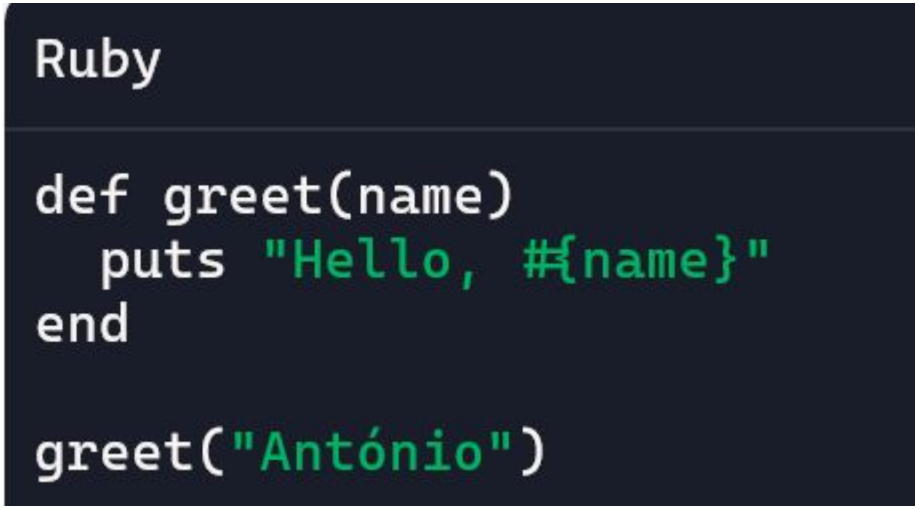
🧪 C#



```csharp
void Greet(string name) {
    Console.WriteLine("Hello, " + name);
}

Greet("António"); // must be inside a class and method
```

- Similar to Java in structure.
- Must be inside a **class**.
- Uses **PascalCase** naming convention by default.

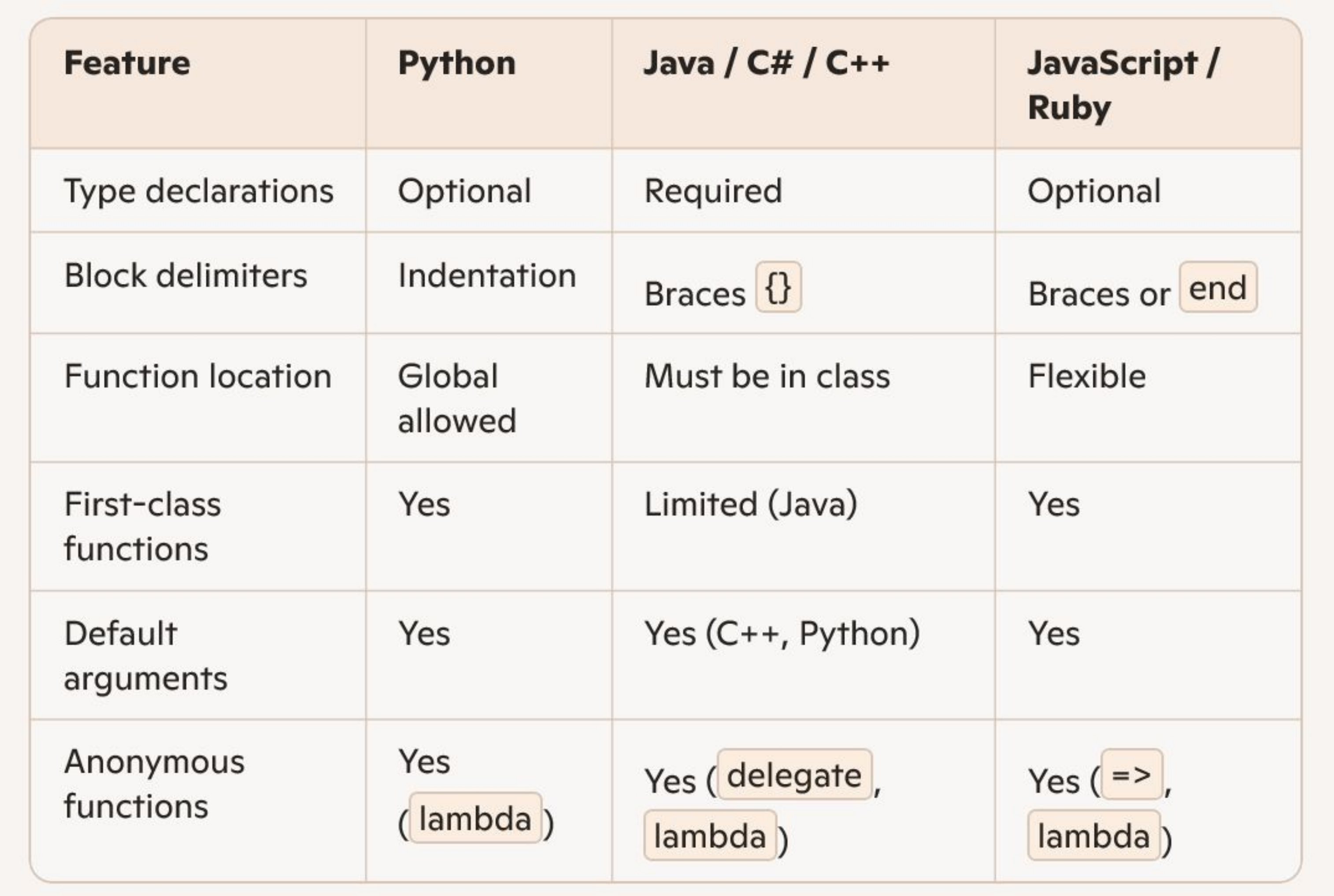
🧮 Ruby


Ruby

```ruby
def greet(name)
  puts "Hello, #{name}"
end

greet("António")
```

- No parentheses required for calls.
- Uses end to close function.
- Highly flexible and expressive.

Key Differences from Python



Feature	Python	Java / C# / C++	JavaScript / Ruby
Type declarations	Optional	Required	Optional
Block delimiters	Indentation	Braces `{}`	Braces or `end`
Function location	Global allowed	Must be in class	Flexible
First-class functions	Yes	Limited (Java)	Yes
Default arguments	Yes	Yes (C++, Python)	Yes
Anonymous functions	Yes (`lambda`)	Yes (`delegate`, `lambda`)	Yes (`=>`, `lambda`)

Exercises

- Do these codecheck exercises.

- Answer this review quiz.

- What was the muddiest point in class?