

Lecture 8

Functions

Lecture 8 - Functions

- Why functions now?
 - We can already write programs with input, conditions, loops and lists.
 - Programs are becoming longer and harder to read.
 - Functions help us split a program into smaller reusable parts.

```
total = sum(values)
average = total / len(values)

# Could this become a reusable function?
```

Functions are about structure and reuse.

Lecture 8 - Functions

- Objectives
 - Define and call simple custom functions.
 - Use parameters and arguments.
 - Return values to the calling code.
 - Understand local variables and scope.
 - Use functions to decompose a problem.

The goal is to design programs, not only write instructions.

Functions as Black Boxes

Using a function without knowing its internal implementation

Lecture 8 - Functions

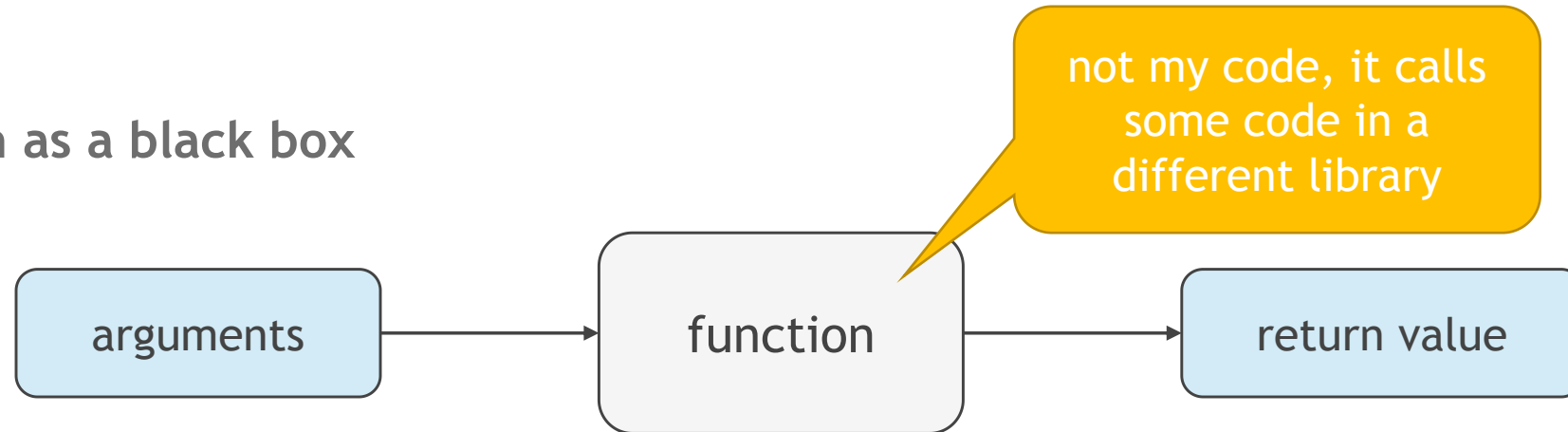
- A function is a named sequence of instructions
 - We call a function to ask it to perform a task
 - A function can receive values
 - A function can return a result
 - Built-in functions are already provided by Python.
- We need the documentation to understand what it does, and how to use it correctly.

```
price = round(6.8275, 2)  
  
# round receives two arguments  
# and returns 6.83
```

We know what `round()` does, not how it is implemented.

Lecture 8 - Functions

Function as a black box



`round(6.8275, 2) → 6.83`

As users, we need the specification: inputs (arguments) and expected output.

Defining Functions

From using functions to writing our own

Lecture 8 - Functions

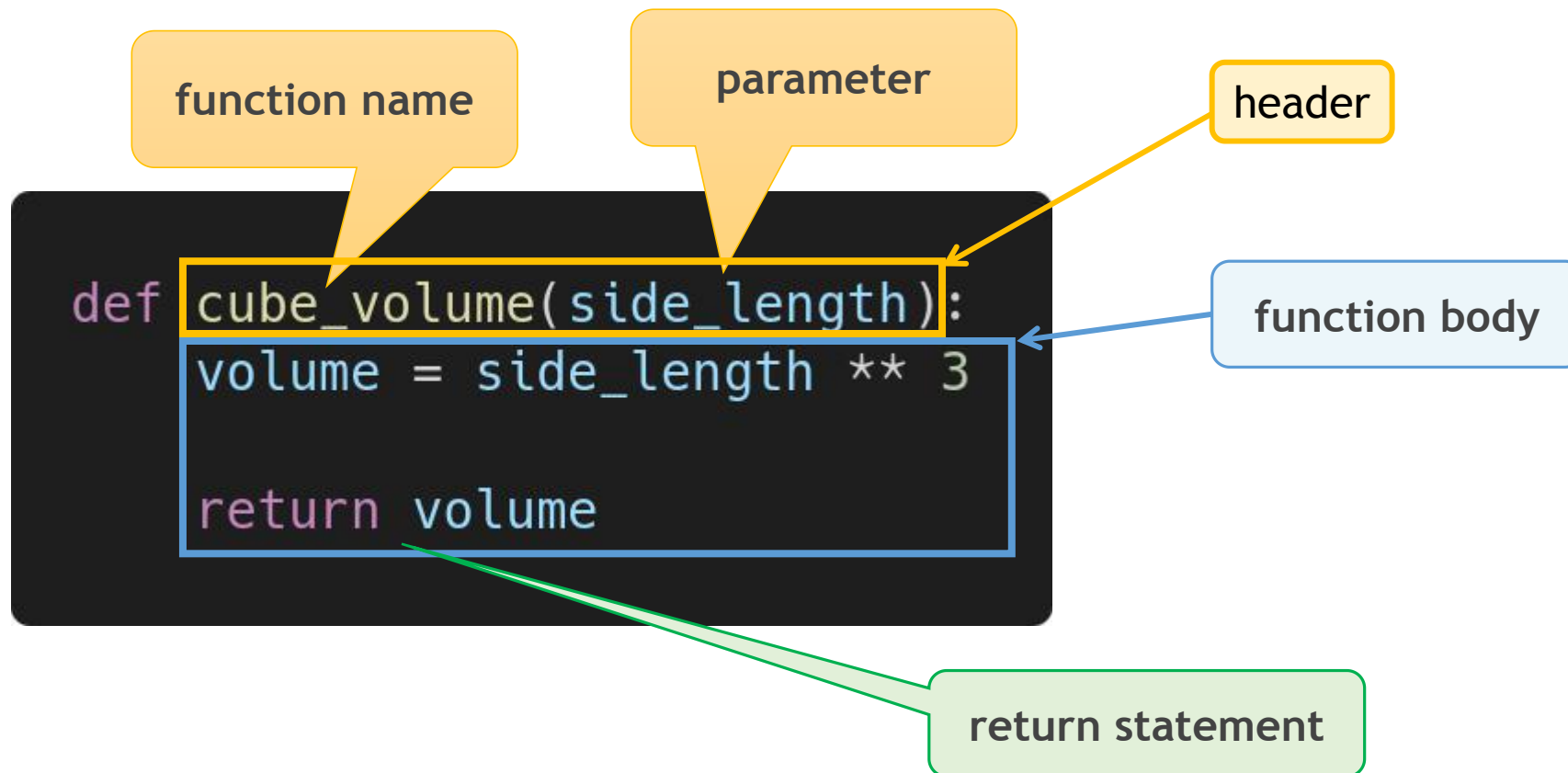
- Defining a function
 - Use the keyword `def`
 - Choose a meaningful function name
 - List the parameters between parentheses (possibly none)
 - End the header with a colon
 - Indent the function body

```
def cube_volume(side_length):  
    volume = side_length ** 3  
  
    return volume
```

`def` opens a new indented block.

Lecture 8 - Functions

Function definition: main parts



Lecture 8 - Functions

Function definition: main parts

```
def cube_volume(side_length):  
    volume = side_length ** 3  
    return volume
```

The parameter name is used as a variable inside the body. Calling a function, we initialize that parameter variable with the value passed as an argument

Lecture 8 - Functions

- A function can be defined inside your main code, or in a separate module (file) and imported with
 - `import module_name`
 - `from module_name import my_func`
- Calling a function
 - A function definition alone **does not execute** the function body
 - The function body runs only when the function is called
 - The returned value can be stored or used in an expression.

```
# definition
def cube_volume(side_length):
    return side_length ** 3

# main code
result = cube_volume(2)
print(result)      # 8
```

Define first, call later.

Lecture 8 - Functions

- Flow of execution

main code

```
# some code...  
  
result = cube_volume(2)  
  
# call enters function  
# return comes back here
```

Lecture 8 - Functions

- Flow of execution
 1. The program reaches the function call.

main code



```
# some code...  
result = cube_volume(2)  
  
# call enters function  
# return comes back here
```

Lecture 8 - Functions

- Flow of execution
 1. The program reaches the function call.
 2. Arguments are evaluated



some code...
result = cube_volume(2)

call enters function
return comes back here

main code

A grey thought bubble containing the function definition code. A red arrow points from the 'main code' box to the 'def' keyword.

```
def cube_volume(side_length):  
    volume = side_length ** 3  
  
    return volume
```

Execution temporarily jumps into
the function.

Lecture 8 - Functions

- Flow of execution
 1. The program reaches the function call.
 2. Arguments are evaluated
 3. Parameter variables are initialized



main code

```
# some code...  
result = cube_volume(2)  
  
# call enters function  
# return comes back here
```

```
def cube_volume(side_length):  
    volume = side_length ** 3  
  
    return volume
```

Execution temporarily jumps into the function.

Lecture 8 - Functions

- Flow of execution
 1. The program reaches the function call.
 2. Arguments are evaluated
 3. Parameter variables are initialized
 4. The function body is executed



```
# some code...  
  
result = cube_volume(2)  
  
# call enters function  
# return comes back here
```

main code

```
def cube_volume(side_length):  
    volume = side_length ** 3  
  
    return volume
```

Execution temporarily jumps into the function.

Lecture 8 - Functions

- Flow of execution

1. The program reaches the function call.
2. Arguments are evaluated
3. Parameter variables are initialized
4. The function body is executed
5. The return value goes back to the call point



main code

```
# some code...  
result = cube_volume(2)  
  
# call enters function  
# return comes back here
```

```
def cube_volume(side_length):  
    volume = side_length ** 3  
    return volume
```

Execution temporarily jumps into the function.

Lecture 8 - Functions

- Flow of execution

1. The program reaches the function call.
2. Arguments are evaluated.
3. Parameter variables are initialized
4. The function body is executed
5. The return value goes back to the call point
6. The main program continues



main code

```
# some code...  
result = cube_volume(2)  
  
# call enters function  
# return comes back here
```

Functions and Parameters

How to give input values to a function

Lecture 8 - Functions and Parameters

- Arguments and parameters
 - An **argument** is the value used in the function call
 - A **parameter** is the variable used inside the function definition
 - Arguments can be literals, variables or expressions.

```
cube_volume(2)      # literal  
cube_volume(size)   # variable  
cube_volume(a + b)  # expression
```

Parameter receives the argument value.

Lecture 8 - Functions and Parameters

- Functions without parameters
 - Some functions do not need external values
 - The parentheses are still required
 - They can still print or return something.

```
def print_header():  
    print("SALES REPORT")  
    print("-----")  
  
print_header()
```

No parameters, but still a function call.

Lecture 8 - Functions and Parameters

- Functions may have one or more parameters
 - separated by , in the header

Function with one parameter

```
def square(x):  
    return x ** 2  
  
print(square(5))
```

Function with two parameters

```
def total_price(price, quantity):  
    return price * quantity  
  
print(total_price(12.5, 3))
```

The number of arguments must match the function definition...
unless we define default values (see next slide)

Lecture 8 - Default Parameter Values

- A parameter can have a **default value**
→ **optional parameter**
- It is explicitly initialized in the header with:

param=value
- The default is used only when the caller **omits** that argument, otherwise it is **overwritten**
- Parameters without a default value are mandatory
- Defaults are useful for common or optional settings

```
def price_with_tax(price, rate=0.23):  
    return price * (1 + rate)  
  
price_with_tax(100)           # 123.0  
price_with_tax(100, 0.10)    # 110.0
```

Default values make simple calls shorter.

Parameters without default values must come before parameters with default values
→ **Required parameters first, optional parameters after.**

Lecture 8 - Default Values: A Common Warning

- Default arguments are **evaluated** only once, when the function is **defined**.



→ a mutable default object may be **shared** across multiple calls.

- Avoid mutable objects as default values
- Use `None` as a safe placeholder, then create the list inside the function.

Avoid

```
def add_item(item, basket=[]):  
    basket.append(item)  
    return basket
```

```
print(add_item("a"))    # ['a']  
print(add_item("b"))    # ['a', 'b']
```

Better

```
def add_item(item, basket=None):  
    if basket is None:  
        basket = []  
    basket.append(item)  
    return basket
```

Lecture 8 - Positional vs Keyword Arguments

- There are 2 ways to pass arguments to a function and initialize the corresponding parameters:
 - **Positional** arguments are matched by **order**
 - **Keyword** arguments are matched by **parameter name**
- Keyword arguments can make calls easier to read.

```
def invoice(total, tax_rate=0.23, currency="€"):
    final = total * (1 + tax_rate)
    return f"{currency}{final:.2f}"

invoice(100, 0.10, "$")
invoice(100, currency="$", tax_rate=0.10)
```

Same result, different call style.

Lecture 8 - Mixing Positional and Keyword Arguments

- When both styles are used:
 - write all **positional arguments first**;
 - then write keyword arguments;
 - keyword arguments may be in any order.
- After a keyword argument, positional arguments are no longer allowed.

```
def format_price(value, currency="€", decimals=2):  
    return f"{currency}{value:.{decimals}f}"  
  
# Valid  
format_price(12.5)  
format_price(12.5, "$")  
format_price(12.5, currency="$", decimals=1)  
format_price(12.5, decimals=1, currency="$")  
  
# Invalid  
format_price(value=12.5, "$") # SyntaxError
```

Rule: positional first, then keywords.

Lecture 8 - Choosing a Calling Style

- Use **positional** arguments for the main required values
- Use **keyword** arguments for optional settings or values that need explanation
- Avoid calls that are correct but hard to read.

```
# Harder to read
create_report("sales.csv", "out.txt", True, 2)

# Clearer
create_report(
    "sales.csv",
    output_file="out.txt",
    include_summary=True,
    decimals=2
)
```

Readability matters in function calls.

Return Statement

How to obtain the result from a function

Lecture 8 - Return Statement

- The return statement
 - return terminates the function execution
 - The returned value is sent back to the caller
 - If a function reaches the end without executing a return statement, it returns `None` by default.
 - return without a value also returns `None`.
- A function can be used inside larger expressions.

```
def add_vat(price, rate):  
    tax = price * rate  
    return price + tax  
  
item_cost = add_vat(100, 0.22)  
  
# call inside expression  
total = shipment_cost + add_vat(100, 0.22)
```

return is not the same as print().

Lecture 8 - print() and return Are Different

print()

- Displays information to the user.
- Does not send a value back to the caller.
- Useful for messages and reports to users.

```
def show_total(x):  
    print(x)  
  
result = show_total(10)  
print(result)    # None
```

return

- Sends a value back to the caller.
- The value can be stored or used in expressions.
- Useful for reusable computations.

```
def compute_total(x):  
    return x  
  
result = compute_total(10)  
print(result)    # 10
```

Use return when the caller needs the result.

Lecture 8 - Functions Without return

- A function may have no explicit return statement
- In that case, Python returns `None` automatically
- This is normal for functions used only to display or perform an action.

```
def print_summary(values):  
    print(f"Count: {len(values)}")  
    print(f"Total: {sum(values)}")  
  
result = print_summary([10, 20, 30])  
print(result)    # None
```

No return means return None.

Lecture 8 - Returning Multiple Values

- A function can return more than one value
- Python packs them into a *tuple*
- Multiple assignment can unpack the result immediately
- This is useful when results naturally

Same idea as `a, b = b, a`.

```
def min_max(values):  
    return min(values), max(values)  
  
result = min_max([12, 8, 15])  
print(result)           # (8, 15)  
  
low, high = min_max([12, 8, 15])  
print(low, high)       # 8 15
```

Lecture 8 - Multiple return Statements

- A function can have **several return statements**
- Each **branch** can terminate naturally with its own result
- As soon as return is executed, the function stops
- This often avoids unnecessary else nesting.

```
def shipping_cost(order, premium):  
    if order < 0:  
        return None           # invalid input  
  
    if premium and order >= 50:  
        return 0  
  
    if order >= 80:  
        return 0  
  
    return 4.0                 # default case
```

Each exit case is explicit.

Lecture 8 - Designing Return Values

- Make all possible exits clear
- Prefer consistent return types
- Avoid mixing computation and printing unless the purpose is clear
- Use return values to let the caller decide what to do next.

```
def average(values):  
    if len(values) == 0:  
        return None  
  
    return sum(values) / len(values)  
  
avg = average(values)  
if avg is None:  
    print("No data")  
else:  
    print(f"Average: {avg:.2f}")
```

Return the result; print at the interaction level.

Functions and Lists

Reusing algorithms on collections

Lecture 8 - Functions and Lists

- Lists as arguments
 - A function can receive a list as an argument
 - This is useful for reusable data-processing algorithms
 - Inside the function, the parameter is used like a normal list.

```
def average(values):  
    total = 0  
    for value in values:  
        total = total + value  
    return total / len(values)  
  
print(average([10, 20, 30]))
```

Now functions and loops work together.

Lecture 8 - Functions

- Designing a function
 - What information does it need?
 - What result should it return?
 - What should be done outside the function?
 - What name explains its purpose?

```
def count_positive(values):  
    count = 0  
    for value in values:  
        if value > 0:  
            count = count + 1  
    return count
```

Start from purpose, inputs and output.

Scope and Local Variables

Where variables exist

Lecture 8 - What Is Scope?

Local variables to isolate computations

- Variables created **inside** a function are **local**
 - their **scope** is limited to the function
 - They exist only during that function call
 - They are not directly visible outside
- *Scope* tells us **where** a name can be used
- **Parameters** are local variables initialized by arguments
- Code outside the function cannot directly use those local names (or, if used, they refer to a different variable with same name)

```
def compute_tax(price):  
    rate = 0.23      # local variable  
    tax = price * rate  
    return tax  
  
print(rate)          # NameError  
print(tax)           # NameError
```

price, rate and tax are local names.

Lecture 8 - Important Python Rule: Blocks and Scope

- Functions create a local scope
- In Python, `if` / `for` / `while` blocks **do not create a new local scope**
- A variable assigned inside such a block belongs to the surrounding scope
- However, the assignment may not happen if the block is not executed
→ variable never created/initialized

```
if income > 50000:  
    tax_rate = 0.30  
  
print(tax_rate)  
# Works only if the if body executed.  
# Otherwise: NameError
```

This is about variable availability, not only function scope.

Lecture 8 - Initialize Before a Conditional Block

- If a variable is needed after an `if` / `elif` / `else` block, **make sure it is always assigned**
- A common strategy is to **initialize it before the block**
- Alternatively, cover all branches with `else`

Do not rely on a branch that may not run.

```
# Safer
status = "standard"

default_discount = 0

discount = default_discount
if customer_type == "premium":
    discount = 0.15
elif customer_type == "student":
    discount = 0.10

final_price = price * (1 - discount)
```

Lecture 8 - Initialize Before a Loop

- A for or while loop may execute zero times.
 - for if empty iterator
 - while if the condition is False
- If a variable is used after the loop, initialize it before the loop.
- This is especially important for accumulators, counters and flags.

A variable needed later must exist in every path.

```
count = 0
found = False
largest = None

for value in values:
    count += 1
    if largest is None or value > largest:
        largest = value
    if value == target:
        found = True

print(count, found, largest)
```

Lecture 8 - Functions: Local Variables vs External Variables

- Inside a function, assign local variables for the computation
- Avoid relying on variables outside the function unless they are constants
- Pass needed values as parameters and return the result.

Good functions depend mostly on parameters.

```
TAX_RATE = 0.23  # constant-like global name

def total_with_tax(price, rate=TAX_RATE):
    total = price * (1 + rate)  # local
    return total

price = 100
final = total_with_tax(price)
print(final)
```

Lecture 8 - Functions

- Common mistake: expecting a parameter to modify the caller variable
 - For numbers and strings (immutable), assigning to a parameter changes **only the local variable**
 - Use `return` to send the new value back.

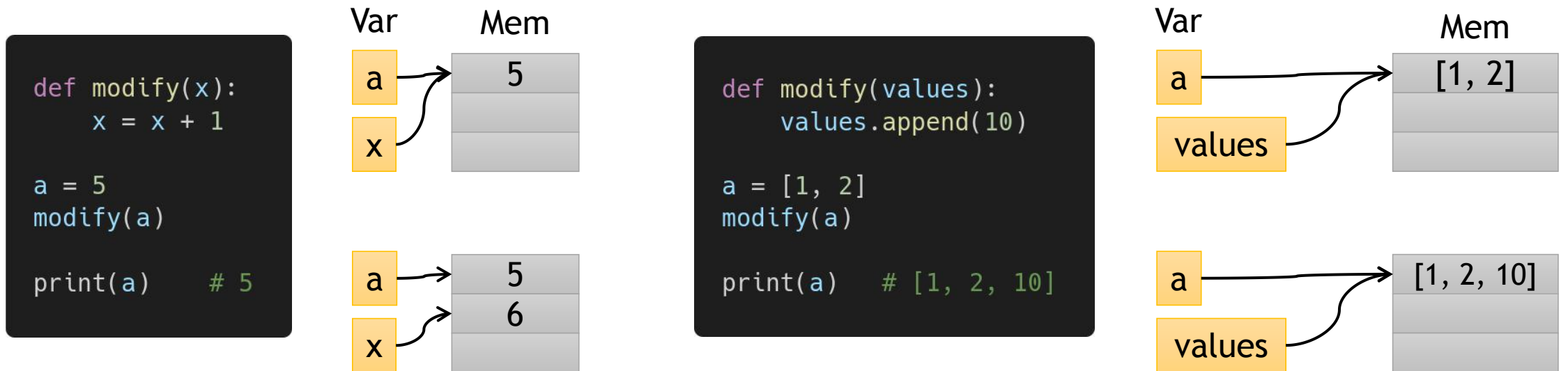
In general, it is **not** a good idea to **modify directly the parameter variables** inside a function; better to copy them and **modify a new local variable** and keep original arguments.

```
def add_tax(price):  
    price = price * 1.22  
    return price  
  
total = 100  
total = add_tax(total)
```

Return the result and assign it explicitly.

Lecture 8 - Functions

- When an object is passed as an argument, the parameter refers to **the same object** as the argument
 - If the object is **immutable**, it cannot be modified; assigning a new value to the parameter only changes what the local parameter refers to (a new object in memory)
 - If the object is **mutable**, modifications made to the object inside the function are visible outside the function (do not create a new object)



Lecture 8 - Functions

- Mutable arguments: be careful with lists
 - A list can be modified inside a function
 - This affects the same list object used by the caller
 - This can be useful, but it should be intentional.

```
def add_bonus(values):  
    values.append(100)  
    success = True  
    return success  
  
scores = [10, 20]  
success = add_bonus(scores)  
print(scores) # [10, 20, 100]
```

Lists are mutable objects.

Lecture 8 - Functions

- A safer design rule
 - Prefer functions that return a value
 - Avoid unexpected changes to arguments
 - Use clear names and comments/docstrings
 - Test functions with simple known cases.

```
def with_bonus(values):  
    result = list(values)  
    result.append(100)  
    return result
```

Make side effects explicit or avoid them.

Lecture 8 - Docstrings

- Documenting functions
 - A **docstring** is a short description (string) placed at the beginning of a function, surrounded by `"""docstring"""`
 - It explains purpose, parameters and returned value
 - It helps future readers, including yourself, but not only...
 - Not a simple comment: following certain “styles”, docstrings can be **displayed by editors** and IDEs while writing code, and they can also be used to **automatically generate documentation**

```
def area_rectangle(width, height):  
    """  
    Compute the area of a rectangle.  
  
    Parameters:  
        width: rectangle width  
        height: rectangle height  
  
    Returns:  
        The area of the rectangle.  
    """  
    return width * height
```

Lecture 8 - Functions

- Tips: testing a function
 - A function should be tested with simple cases
 - Compare the output with an expected result
 - Test normal cases and boundary cases.

```
print(average([10, 20, 30]))      # 20  
print(average([5]))               # 5  
print(count_positive([-1, 2, 3])) # 2
```

Do not test only the final program.

Problem Decomposition

From one long program to cooperating functions

Lecture 8 - Functions

- Decomposition example
 - Read data
 - Validate data
 - Compute result
 - Print report
- Each subtask can become a function.

main program looks like:

```
read_values()  
validate_values(values)  
compute_average(values)  
print_report(average)
```

One function, one responsibility.