Module 03 : Writing functions and Debugging

Content

- Using and writing functions
- Building *abstractions*: why functions *matter*
- Debugging
- Responsible use of GenAI in coding: asking AI for help, evaluating output critically

Programs without abstractions

- So far, we only have used *built-in* functions provided by Python (`print()`, `len()`, etc.)
- We have written programs by putting all our instructions in the *global space* (without any *structure*)
- Some block of instructions are or *could* be *repeated*
- It is hard to tell what each part of the program is doing
- The program becomes hard to *understand* and to *modify*

Functions

- A **function** (also called a procedure) is a self-contained **block of code designed to perform a specific task**
- Instead of rewriting the same code *multiple times*, you *group* instructions under a **single name** and **execute** (call) them whenever needed.
- Functions make it possible to *isolate and group a set of sometimes complex instructions* (the *how*) behind a *name*, and more generally behind an *interface* (the *what*).

> This type of programming is called *procedural or modular programming*.

Function syntax

To write your own function, use the `def` operator:

Syntax:

```
def function_name (list of arguments):
    body of the function (instructions)
```

Example

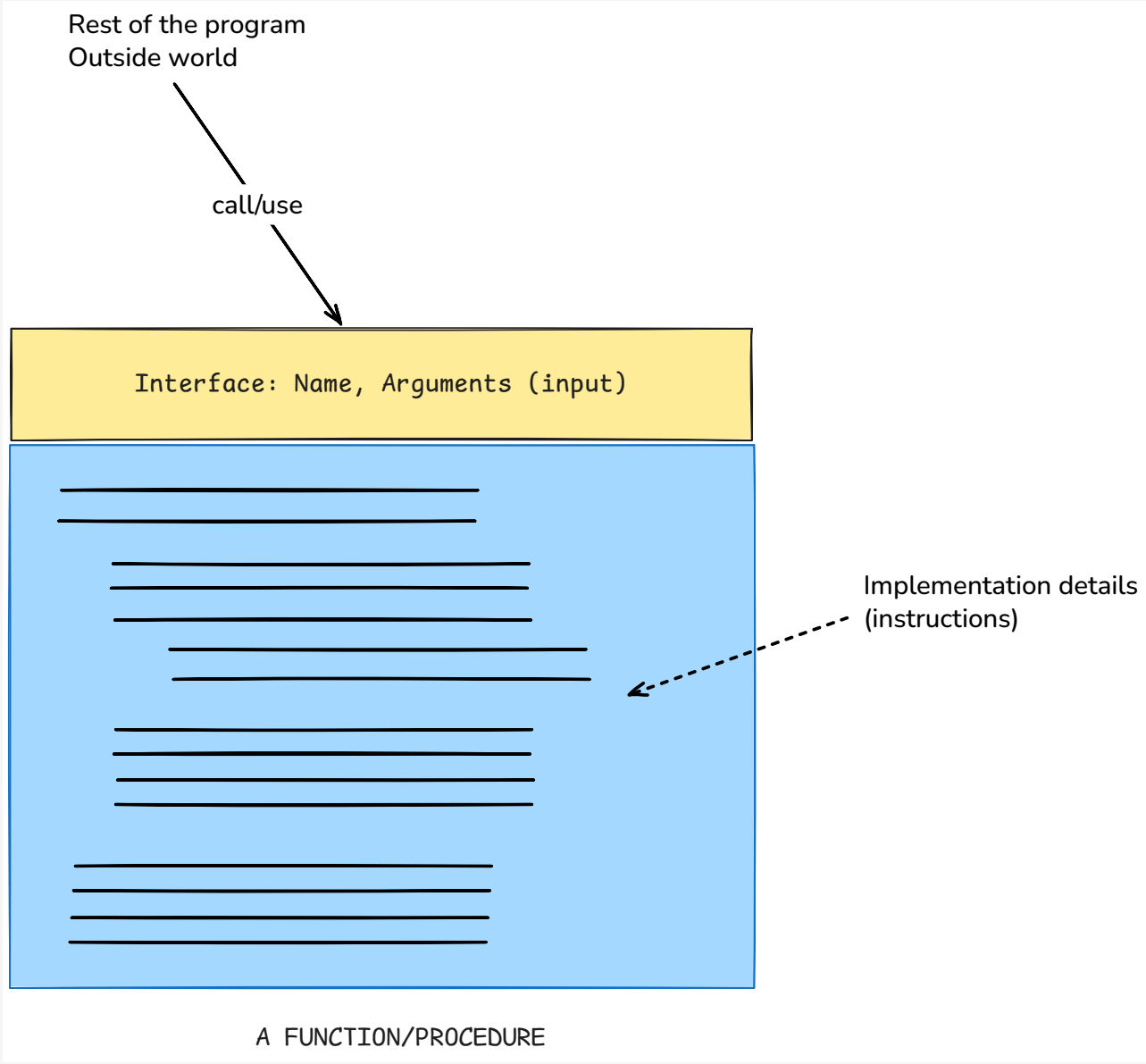
```
# A function called 'calculate_total_price' takes two arguments
def calculate_total_price(base_price, tax_rate):
    total = base_price * (1 + tax_rate)
    return total
```

Anatomy of a function

A function is made of *two parts*:

- The **signature** (or **interface**) consists of its *name* and *arguments*. This is the part **exposed to the rest of the program**.
- The **body** is the *encapsulated* set of instructions that allows the function to *do its work*

Before *using* a function you will need to *define it*.



Calling a function

Once defined, a function can be *called* (executed) by using its name followed by parenthesis:

```
# call calculate_total_price
final_price = calculate_total_price(100, 20)
```

Function arguments and parameters

- Arguments are **data passed into the function** (*input*) to work with
- When a function is called, **arguments passed are copied into the parameters**
- Inside the function body, **parameters** act as **local variables**

```
# The function takes two arguments (input) in order to do its work
def calculate_total_price(base_price, tax_rate):
    total = base_price * (1 + tax_rate)
    return total

# 100 is copied into 'base_price' parameter, 20 is copied into 'rax_rate'
final_price = calculate_total_price(100, 20)  # Returns 120.0
```

Return value and execution flow

A function **always return** something (*output*), usually the result of the procedure, with a `return` statement. The *return statement* :

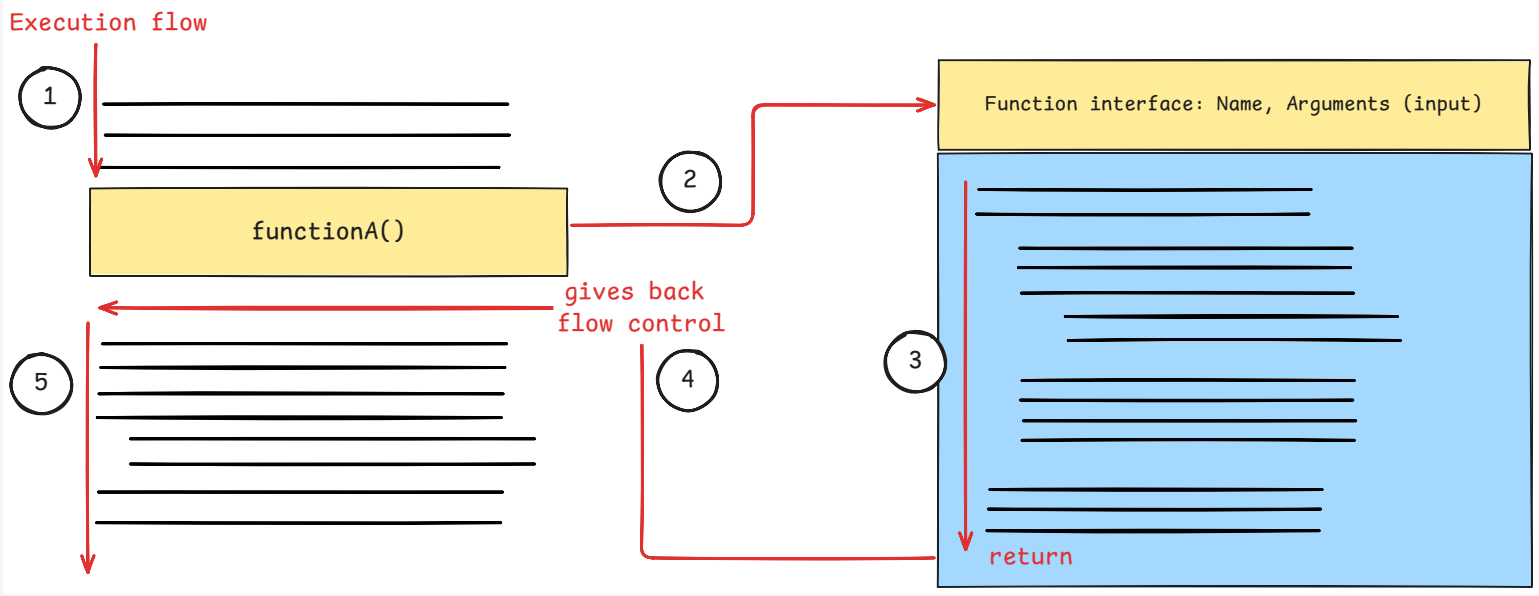
- immediately **stops the execution of a function**
- **passes control flow back to the caller**
- **sends a result**, the return value of the function, to be used in the rest of the program.

```
print("Let's compute the final price...")    # 1

final_price = calculate_total_price(150, 20) # 2 Function returns and passes control back

print(f"Final price is ${final_price:.2f}"   # 3
```

Execution flow when calling a function



Function *without* `return` statement

If you omit the `return` statement in a function (or use a bare return with no value), the function return `None`

```
def some_procedure():
 print("do stuff")
 print("do stuff")
 print("finished")

result = some_procedure()

print(result) # None!
```

Return stop the execution of the function

```
def some_procedure():
 print("do stuff")
 print("do stuff")
 print("finished")

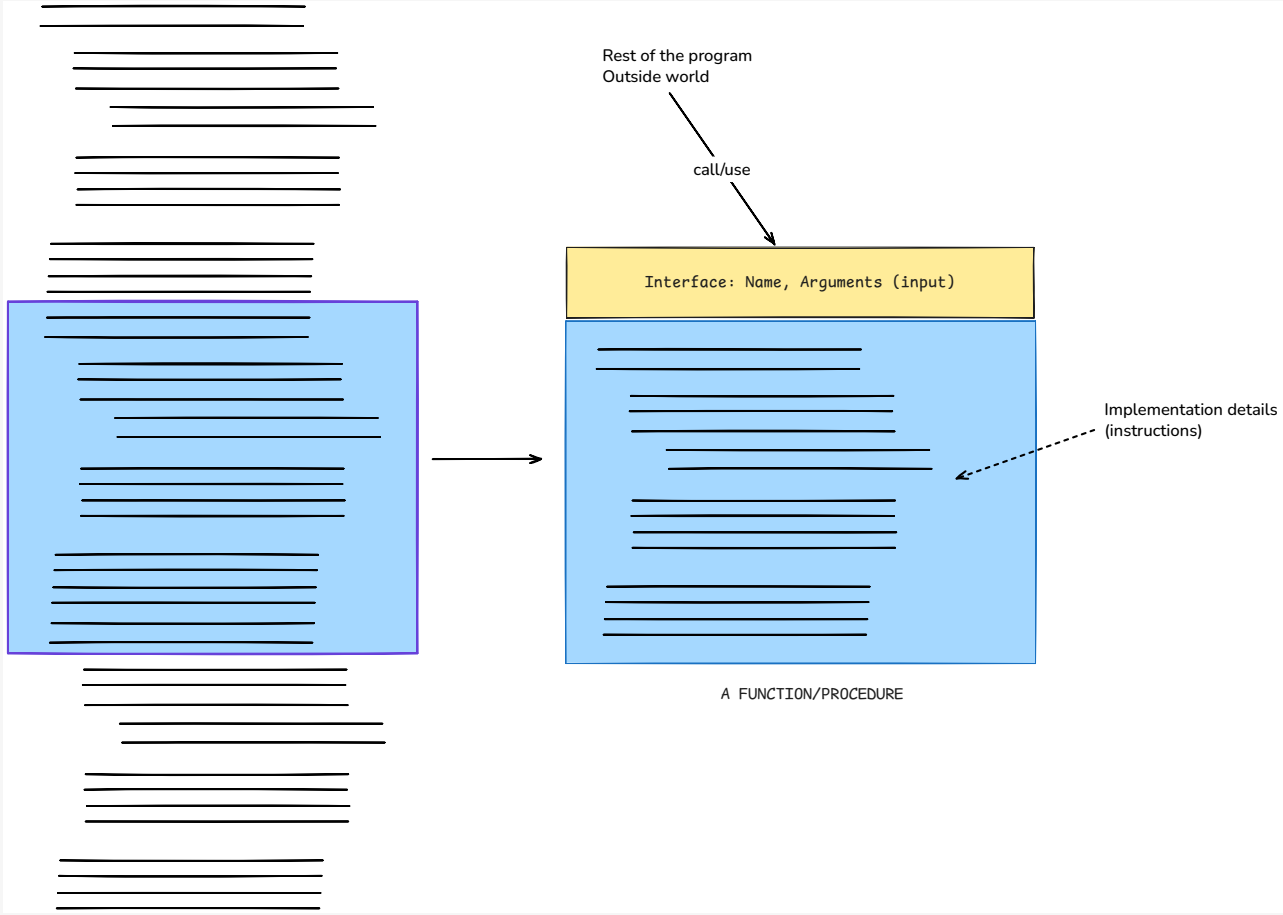
 #Stops and return
 return "done";

 # This will never be executed!
 print("do another thing")

some_procedure()
```

Why writing and using functions *matter*?

- Functions allow **breaking down** a program into multiple **parts**
- Each function acts like a **black box** that **exposes a service** through an *interface* while *hiding its internal details*. They *hide the implementation details* from you and other parts of the program.
- By *writing a program with functions*, one can **reason about the system** to *understand* and **evolve it**. Using functions makes your programs easier to write, read, test, and fix.



You don't know! You don't *want* to know!

Have you ever wondered *how* `print()` *actually* works?

No! And that's entirely *the point*! It's (maybe?) interesting, but it's not *your problem*. You just want to use it because you *need* to solve *your current problem*.

Just like with your car: you don't need to know *how* cars work to *go* where you *need* to be, you only need to know how to drive!

Practice : Simple functions

1. **Write a function** called `display_message()` that prints one sentence telling everyone what you are learning about in this chapter. Call the function, and make sure the message displays correctly.
2. **Write a function** called `favorite_book()` that accepts one parameter, `title`. The function should print a message, such as "One of my favorite books is Alice in Wonderland". **Call the function**, making sure to *include a book title as an argument* in the function call.

> From Prof. Karim ZKIK, PhD, Associate Professor

More on functions

Named and positional arguments

When calling a function, you pass values into its *parameters*. Python allows you to pass these values either :

- by **position**. Order *matters* in positional arguments.
- by **name** (explicit parameter matching).

```
def add(x, y, z):
 return x + y + z

# Positional argument (implicit), order matters!
add(1, 2, 3)

# Named arguments (also called 'keywords')
add(y=2, z=1, x=3)

# Mixing
# You can combine both styles in a single function call,
# but positional arguments must always come BEFORE named arguments.

add(2, z=3, y=2)
```

Default arguments

In Python, functions can have **default arguments**, which are parameters with **predefined values**. This means you don't *always* need to pass every argument while calling a function:

- If you provide a value, Python uses it
- If you skip it, the default value is used automatically.

Syntax:

```
    def function_name(param1=value1, param2=value2, ...):
        # function body
```

Default arguments, example

```
# tax_rate as the value 20 by default
def calculate_total_price(base_price, tax_rate = 20):
    total = base_price * (1 + tax_rate)
    return total

# 20 is used as default value (no value provided)
final_price = calculate_total_price(100)  # Returns 120.0

# 30 is provided and used instead of default value
final_price = calculate_total_price(100, 30)  # Returns 130.0
```

> Rule: Non-default parameters must come **before default parameters** in the function definition

Scope

Functions, like if/else block or for/while loop, has a *scope*.

The *body* of the function is a *a code block*, variables declared in it are *local* to the function: their lifecycle ends (they are destroyed) when the function returns.

```
x = 5

def setX(num):
    # Local variable x is not the same as the global variable x (global scope)
    x = num # => 43
    print (x) # => 43
    return # local x is destroyed and no longer exists

setX(43)
print(x) # 5
```

Arbitrary number of arguments

A function can also take an *arbitrary* number of arguments.

The `*args` syntax collects any number of extra positional arguments into a *tuple*.

```
# A function that takes an arbitrary number of positional arguments
def varargs(*args):
    return args

print(type(varargs(1, 2, 3)))

# Example
def sum_all(*args):
    # args is a tuple of all positional arguments passed
    return sum(args)

print(sum_all(), sum_all(1,2), sum_all(10, 20, 30, 40))
```

Arbitrary number of named arguments

The `**kwargs` syntax collects any extra keyword (*named*) arguments into a *dictionary*.

```
# Arbitrary number of named arguments
def keyword_args(**kwargs):
    return kwargs

keyword_args(1, 2)          # TypeError: keyword_args() takes 0 positional arguments but 2 were given
keyword_args(foo=1, bar=2) # {'foo': 1, 'bar': 2}

def print_user_profile(**kwargs):
    # kwargs is a dictionary of named arguments
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_user_profile(name="Jane", age=30, role="CEO")
```

Practice : Functions and arguments

1. **Write a function** called `make_shirt()` that accepts a size and the text of a message that should be printed on the shirt. The function should print a sentence summarizing the size of the shirt and the message printed on it. **Call the function** once using **positional arguments** to make a shirt. **Call the function** a second time using **keyword arguments**.
2. **Modify** the `make_shirt()` function so that shirts are large by default with a message that reads "I love Monty Python". Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.
3. **Write a function** called `describe_city()` that accepts the name of a city and its country. The function should **return** a simple sentence, such as "Reykjavik is in Iceland". **Give the parameter** for the country a default value. **Call your function** for three different cities, at least one of which is not in the default country, and **store each sentance** in a list. **Print** all sentances.

> From Prof. Karim ZKIK, PhD, Associate Professor

Correction : Functions and arguments 1/2

```
def display_message():
 print("I'am learning how and why to write functions in Python")

display_message()

def favorite_book(title):
 print(f"One of my favorite books is {title.title()}")

favorite_book("Comment parler des livres que l'on a pas lu ?")
```

Correction : Functions and arguments 2/2

```
def make_shirt(size="L", text="I love Monty Python!"):
 print(f"Size: {size} Text: '{text}'")

make_shirt('M', "Best boss")
make_shirt(size='M',text="Best boss")

make_shirt('M')
make_shirt()
make_shirt(text="Stupid", size="S")

def describe_city(name, country="France"):
 return f"{name.title()} is in {country.title()}"

descriptions = [describe_city("Paris"), describe_city("London", "England"), describe_city("Pnom Penh", "Cambodia")]

for desc in descriptions:
 print(desc)
```

Advices on writing *good* functions

- It is recommended to **use the infinitive form** in English. A function *acts*, it manipulates and transforms data. **A verb is therefore a good choice for a function name**. For example, `search`, `create_new_game`, or `cancel_order` are good choices.
- It is also important to name arguments well so that the user of your function easily understands what they correspond to. **Never use single-character names** like `a` or `i`, unless the function's purpose is completely obvious. Today we have *auto-completion tools*, so we have *no excuse* for not using long yet meaningful names for our variables and functions!
- The more arguments a function accepts, the higher its *arity*, making it harder to remember the *order* in which to provide their values and *what* each argument is used for. This adds *complexity* and requires extra *mental effort* from the person using it. You must therefore **define and name function's arguments carefully** to provide the **simplest possible signature** for the problem it is meant to solve.

What is a *bug*?

A *bug* is the *difference* between *expected/assumed behavior* (in your *head*) and *actual/real* behavior of the program.

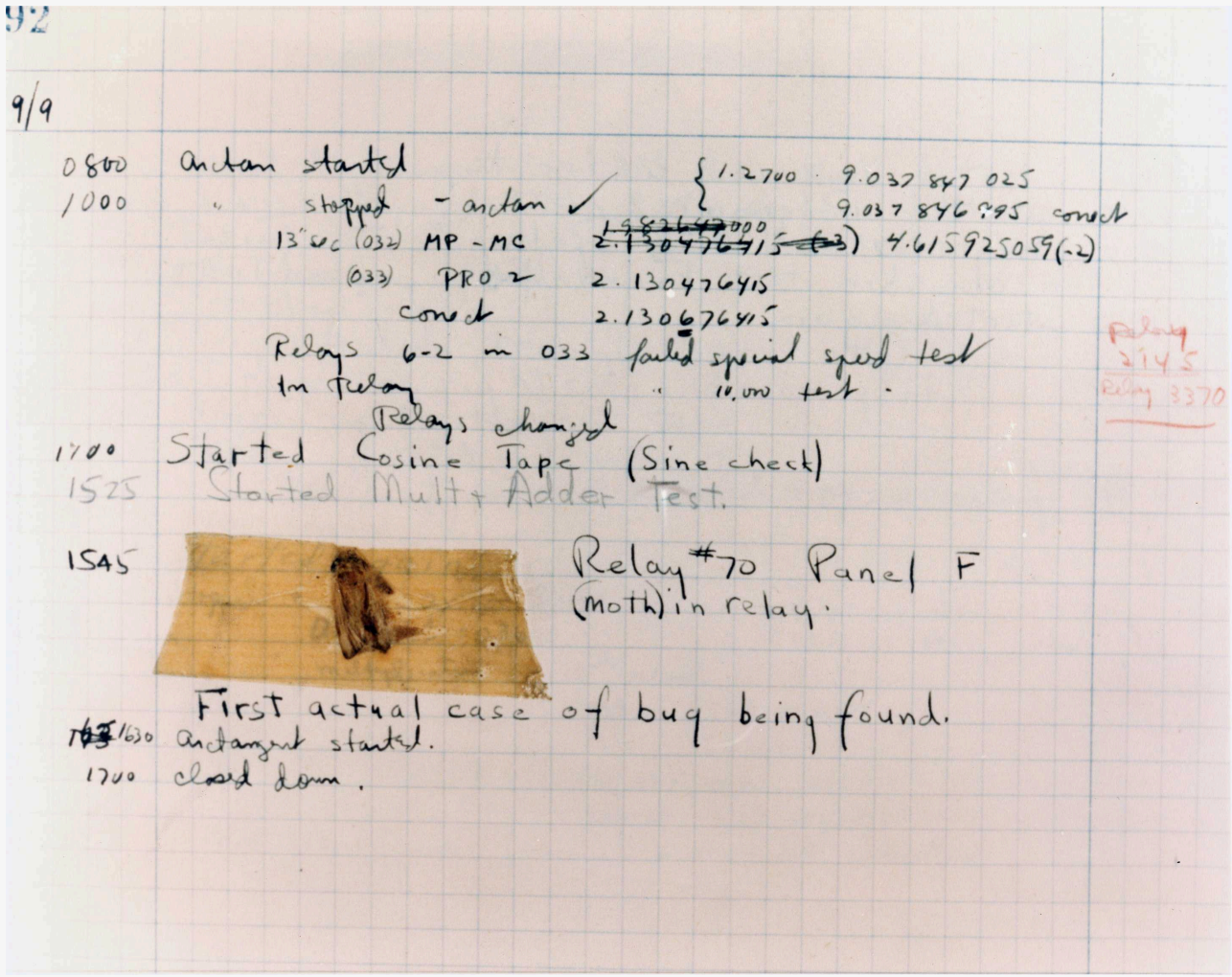
It is a *mismatch* or an expectation-reality *gap*.

It results in a *defect* or *fault* in the program.

Origin of *bugs*?

> Moth found trapped between points at Relay #70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947.

> But, is this legend true? In 1878, Edison wrote about "Bugs as being little faults and difficulties"



Debugging

> "Everyone knows that *debugging is twice as hard as writing a program in the first place*. So if you're as clever as you can be when you write it, how will you ever debug it?" Brian W. Kernighan

Debugging is the process of *finding*, *analyzing*, and *removing errors* (or "bugs") in a computer program so that it runs *correctly* (as expected).

Debugging workflow

1. Read error messages if there are any
2. Reproduce the issue in a *isolated* environment (in a dedicated program or in the Python console). Isolate the code to trigger the error consistently so you can see what causes it.
3. Narrow down which line or section of code is producing the wrong output or crash.
4. Modify the code to correct the logic without breaking other parts of the program.
5. Run the program again to verify the bug is gone and *no new issues were introduced*.

Basic debugging, from simple bugs to hardest ones 1/2

Simple bugs:

- Syntax errors (typos, broken code structure) or Runtime Error (divide by 0, open a file that does not exist, etc.)
- Type errors
- Invalid keywords
- Using out of scope or undeclared variables
- Improper bound bounds (index overflow), etc.

How to debug: **Read the error messages** thrown by the interpreter and **fix** accordingly.

Read the error messages

```
def sum(*numbers):
    sum = 0
    for x in numbers
        sum += x
    return sum

sum(1,2,3)
```

Error:

```
  File "...program.py", line 5
    for x in numbers
                    ^
SyntaxError: expected ':'
```

- Check the **line** number (go to line 5)
- Read the error: `SyntaxError`, expected ':'. I forgot a semicolon!

Basic debugging, from simple bugs to harder ones 2/2

Harder bugs:

- The program executes *without errors* but produces *wrong results*. It does not behave *as expected*.
- Bugs occurring *only* in special conditions or context (not every time)
- Unexpected variable mutations
- Logical flaws (incorrect conditions or logic)

How to debug hard bugs

> "The most effective debugging tool is still **careful thought**, coupled with **judiciously placed print statements**." Brian W. Kernighan

- **Think** and **print** values to see what is stored in them and to *follow the execution flow*
- **Explaining your code out loud** to an object (or colleague) to spot where logic breaks down is quite useful!

Practice : Debugging

Here is a small program with *bugs*, supposed to calculate the average of grades that are greater than or equal to 10/20 from a list.

Debug this code.

```
grades = [12, 8, 15, 17, 9, 14]

def compute_average(grades)
    total = 0
    count = 0

    for grade in grades:
        if grade <= 10:
            total = total + grade
            count = count + 1

    average = total / grade
    return average

result = compute_average(grades)
print("Average of notes greater or equal than 10 : " + result)
```

Responsible use of GenAI in our programming sessions

- Use GenAI to **explain** you some piece of code
- Use GenAI to (re)**explain you a concept**, a **syntax** or **documentation** and ask it for **code examples**
- Write your program. **Once it is finished** and it works, **submit it to an AI for review**. It will offer advices on how you can improve your program to learn new techniques
- Write **specific prompt** to focus on few concepts at the same time. E.g. "Write this program... using *only* lists and the method append". **Restrict the scope** to reinforce targeted skills.

Key takeaway: **Don't use GenAI to write code for you**! Use it as a learning assistant to explore concepts and enhance your skills.

Data Camp

- Course : Introduction to Functions in Python :
  - Chapter 1 : Writing your own functions
  - Chapter 2 : Default arguments, variable-length arguments and scope

More practice

Solve the following problems of the problem sheet:

- Problem 10
- Problem 11
- Problem 12

Going Further/Extra information

More advanced techniques for debugging: Use a *debugger*

While inserting `print()` statements works for simple programs, it can become inefficient for complex or larger applications.

Step-debugging lets you *pause code execution*, *inspect memory states in real time*, *step through instructions line-by-line*, and *alter variables on the fly*.

Python comes with a built-in interactive debugger called `pdb`. It requires no external installations and works directly in any terminal.

Debugger usage, example

You can create *breakpoints* to jump from to one point and pause the execution

```
# Import debugger
# pdb.set_trace() hardcodes a breakpoint directly into the code
import pdb; pdb.set_trace()

def calculate_total(prices, tax_rate):
    subtotal = sum(prices)
    breakpoint()  # Execution pauses here; drops you into the pdb shell
    total = subtotal * (1 + tax_rate)
    return total

calculate_total([10, 20, 30], 0.05)
```

Copy/paste the code and **type** the following commands in the interactive debug session:

`l` : to see where you are, `c` : to continue to next breakpoint, `p subtotal` : to print the content of the variable, `n` : next instruction and then `exit`