Module 01 - Programming, with Python

Why learning (even a little) programming is good for you

- Learn **how to solve problems**, how to solve **YOUR** *business* **problems**!
- Develop *new skills*:
  - Build up *intuition* and *analytic thinking* about **processes** (reusable in many contexts!)
  - **Automate** repetitive, *error-prone* tasks and business procedures (saving time!)
  - **Analyze** real world datasets efficiently, allowing **good decision making**
  - Use and master existing tools/apps, in more advanced ways (e.g. spreadsheets)
- You culture! Better understand how things work
- Programming is a *creative activity*, like drawing or writing
- It's **fun** (believe me)!

> Software is *everywhere*, not a day goes by without you interacting with software

What programming will teach you

Learn to get better at:

- **Designing** :
  - *Analyze*: decompose big problems in smaller solvable problems
  - Find the *right problem* to solve
  - Give *shape* to the *good* solution to the right problem
  - Think about *edge cases*
- **Communicating your ideas**
- **Testing your ideas**: Prototyping, Modeling, Quick *feedback loop*
- **Handling failures!**: The world *is* a *mess*, there will be failures! Anticipate what can **go wrong**
- **Specifying** a solution with *good abstractions* (e.g. writing *better prompts* for a LLM!)

> Later, every time you encounter a *new* problem, *ask yourself* this question: *"How would I solve it by writing a program?"*

Programming vs coding

> "People confuse **programming** with **coding**. *Coding* is to programming what's typing is to writing." (Leslie Lamport)

What is *coding* ? (Oxford Dictionary)

> "To *convert* into a code. To express in an indirect way. To *encode*"

- *Translation* in a specific symbol system
- To reify *preexisting* careful toughts (crafting a story, developing an argument, designing a step-by-step procedure, etc.)

What is *programming* ? (Oxford Dictionary)

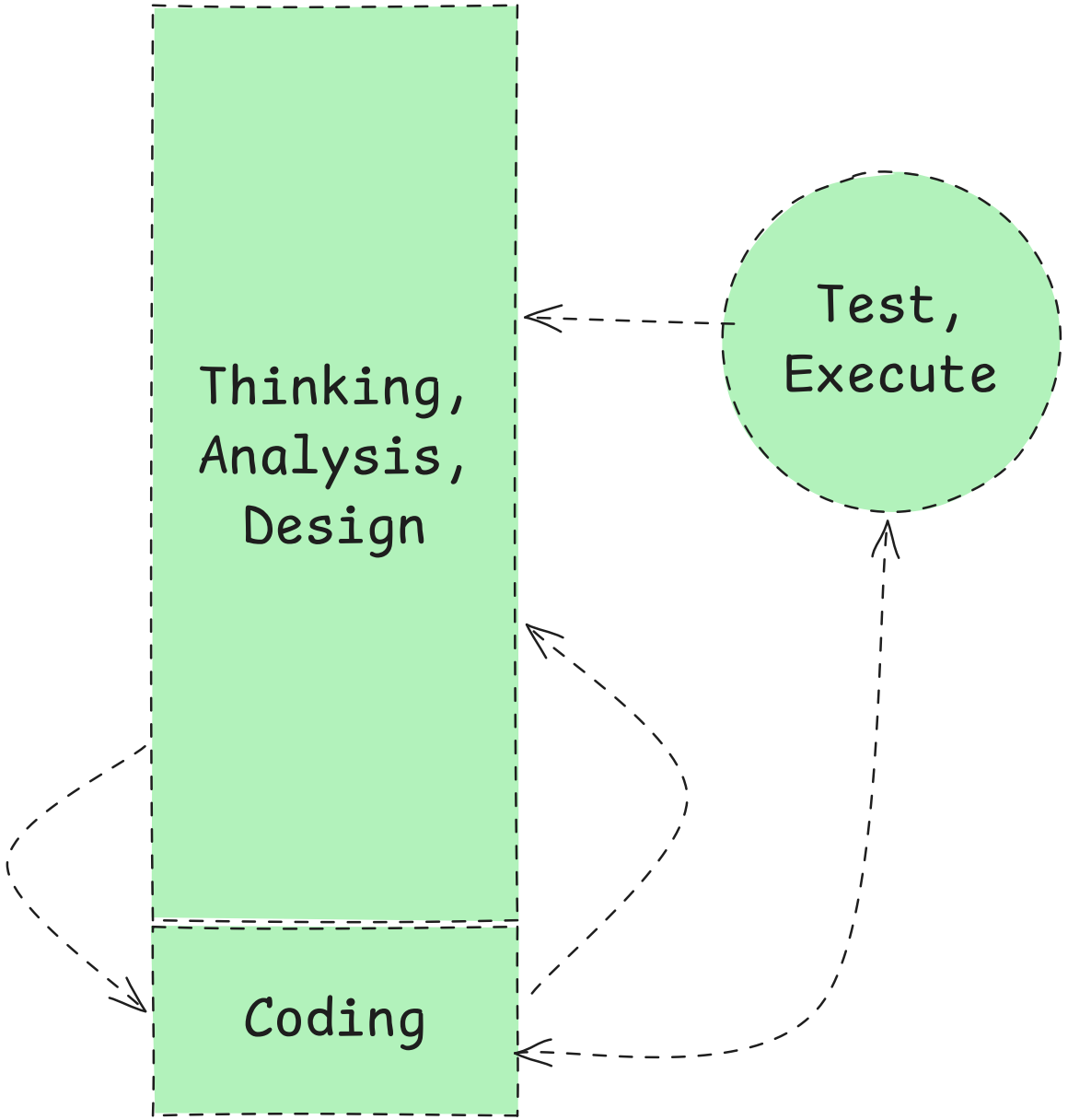
> "To *apply* a set of *instructions* to control the *behavior* of a computer"

- It's **not** about syntax or language or code
- It's about control, logic and **behavior**

Programming vs Coding

- **Programming**: Find and describe an automated **process** that **solves a given problem**. *How to* knowledge.
- **Coding**: **Translate** a process into a **language** the computer can understand

> Coding is *cheap* (especially in the era of gen AI), *programming* isn't! Or is it? While coding, *many things* happen in your brain! Coding can not really be reduced to "typing characters on a keyboard". Nevertheless, writing code is a part of a broader activity we call *programming*



Programming

Programming without coding

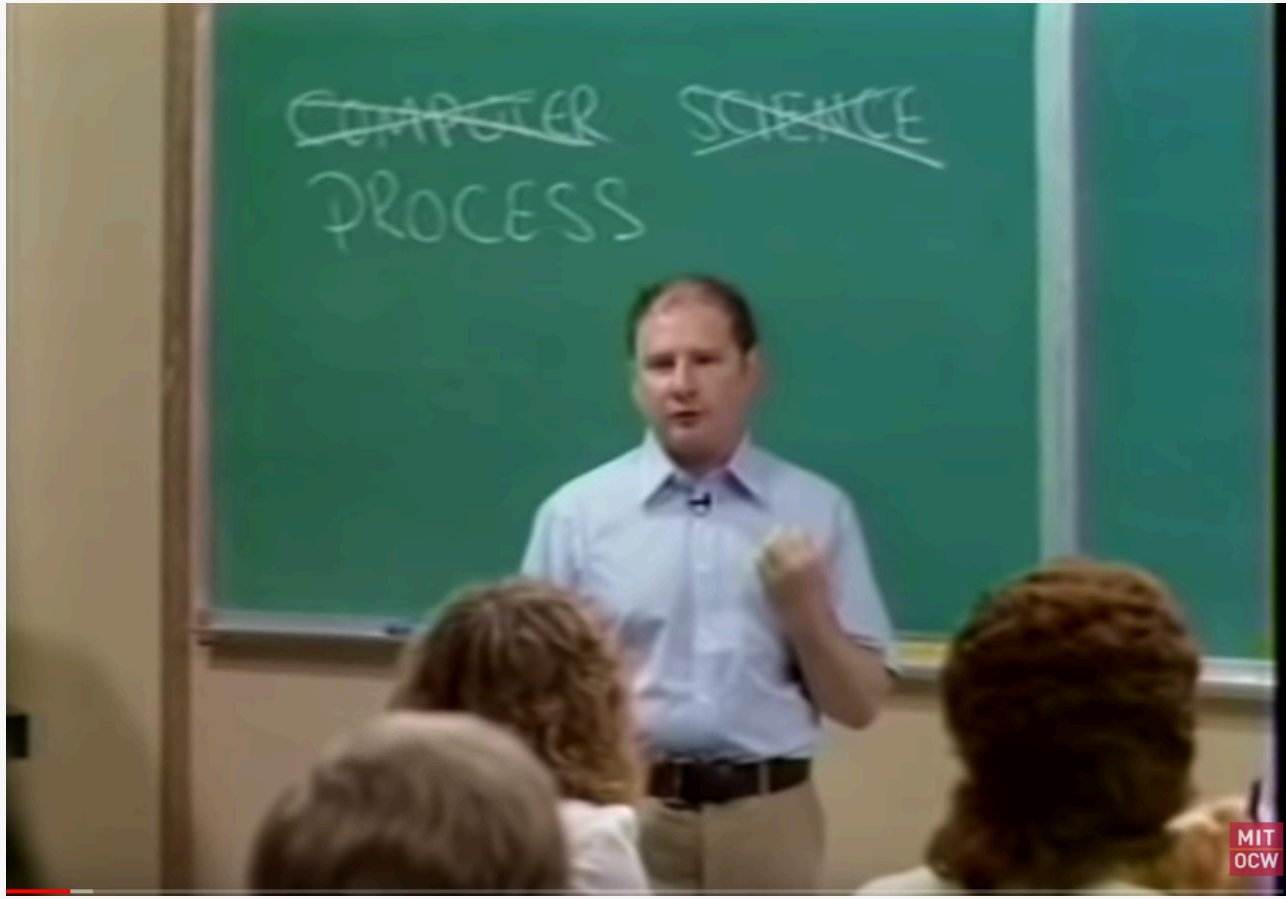
- You can *program* **without** *coding* (and maybe you already did) !
- Visual programming/*NoCode* tools:
  - Automation platforms/Workflows: Make, Zappier, n8n, etc.
  - Applications: Scratch, Airtable, etc.
  - Spreadsheets (Excel, Libre Office, etc.)

> There is (and always be!) code underneath...But it's not *your* problem!

Programming is not (really) about computers

> "Computer science is a terrible name for this business... First of all, **it's not a science**... It's also **not really very much about computers**" (Harold Abelson, MIT 6.001 Structure and Interpretation, 1986). Watch the first episode of this course to have a wonderful introduction to programming

Programming is about *processes*, building *how-to knowledge*.



Example of a program

We want to solve this problem:

Find the youngest person into a group.

Two steps work

1. Find the correct process
2. Program a computer to execute it

Find the process

We have to *design* (or *find*) a **step-by-step process** (instructions) to solve this problem.

Find the youngest person program, first try

1. **For each** person in the group:
    i. **Ask** for their birth date and their name.
    ii. **If** their birth date is **more recent** than the previous youngest person's (or if it's the first person we ask), **consider this person the youngest** person in the group.

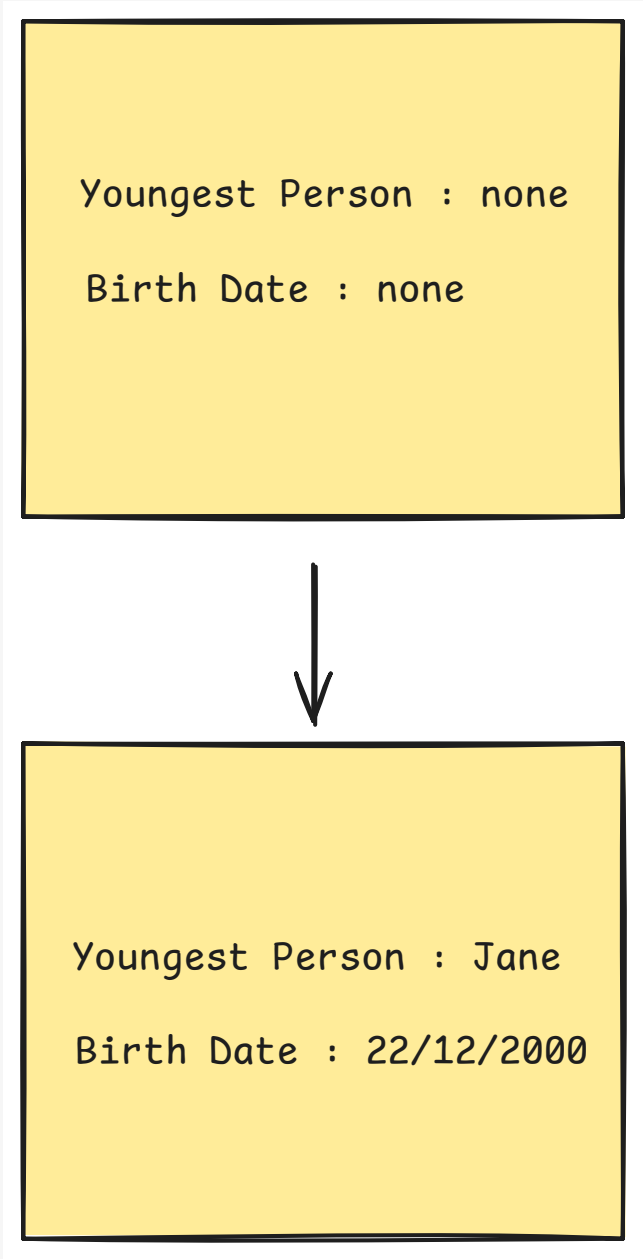
First analysis

- We have to **remember** "somehow" whose birth date is the most recent and *which person* is the youngest *during* the process, in order to give the correct answer at the end: we need **memory**.
- We need to **name** *data* to identify them correctly.

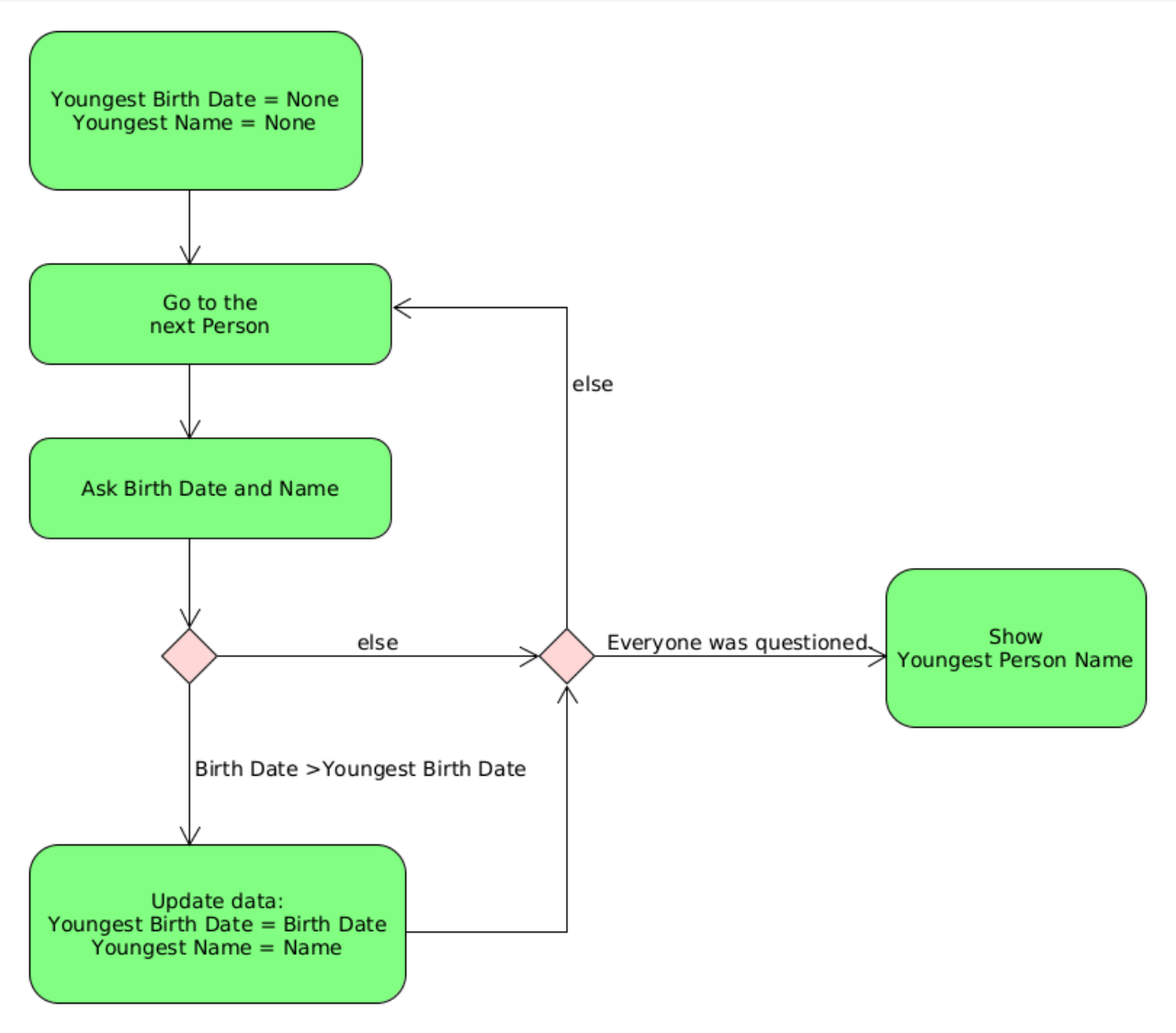
Let's do this again **using a piece of paper** as memory, to keep track of the correct answer.

Find the youngest person program, second try

1. **Take a piece of paper** and **write** *none* for "Youngest Person" and "Birthdate".
2. **For each** person in the group:
    i. **Ask** for their birth date and their name.
    ii. **If** their birth date is **more recent than the one written on the paper** (or if it's the first person), **update** the paper: **erase** the old info and **write** down **this new person's name and birth date**.
3. **Show** the name written on the paper.



Find the youngest person program, a graphical view



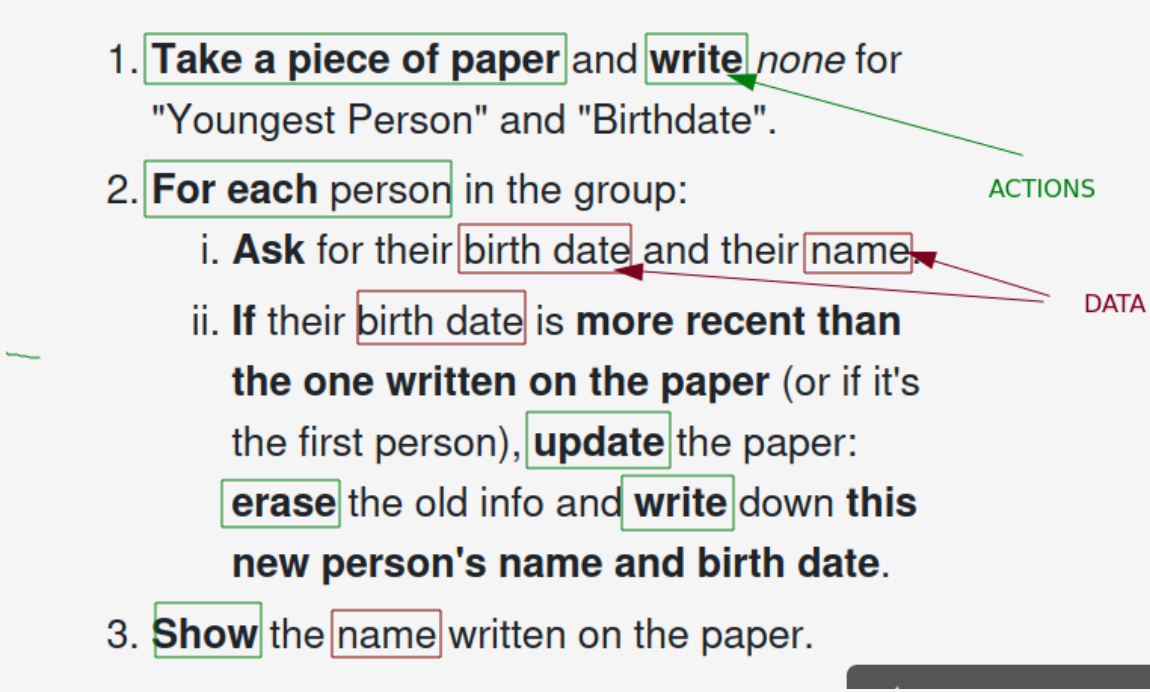
Analysis

- This is what we can call *Pseudo-code* or *Structured english* (For each, If, Update, Erase, etc.). This program can *not* be (yet!) executed by a computer, because computers does not speak english
- Our piece of paper = **memory** (read and write to it)
- `Youngest Person` and `Birthdate` are what we call **variables**:
  - They have a **name** (unique identifier)
  - They **store some data** *during* the process

Our procedure has two parts: Data and Instructions

Notice that our program is made of **two different parts**:

- **Data**: "Birth Date", "Name" are data. It's facts, *information*, *values*. It doesn't *do* anything.
- **Actions/Instructions**: "For each", "if", "update the paper", "ask for" are instructions, *actions* to perform. Instructions are **executable**.



Turn our procedure into a program

Once figured it out, we can **code** (encode, translate) this *procudre* **into a programming language**, like the *Python language* (or any other language!), to **automate** its **execution** with a computer.

An automated procedure is what we call a **program**.

> Any *program* can be *encoded* into different programming languages

Why turning our procedure into a program?

Because we want:

- The process to be **automated**. Once initiated, the process runs completely *autonomously without requiring any human intervention*.
- To execute this procedure *as many times as needed*, adapting to *any group size or volume*, no human error!
- To let *the computer do all the work*, so we can focus on high-value tasks.

> In short: time savings, no human error and flexibility.

What could go wrong in the manual procedure?

- Misspell names or birth dates
- Lack of data (someone forgot…)
- Incomplete process: I forgot to ask someone
- *Edge cases*: two persons could be born on the same day! Need to ask for more precise information? Choose one? Print both?
- Etc.

> Programming is like life: if things *can* go wrong, (given enough time) they *will* go wrong!

How to turn our procedure into a *program*?

- We need to load our *procedure* (data and instructions) *into* the computer (somehow!)
- But computers *do not* understand english (or any human language)!
- We need to *translate* it to a *code* (symbol system) a computer can *understand*.
- What is the *language* of the machine? How do *we tell* the computer to execute our procedure?
- How do *we program* the computer?

But first, what is (really) a computer?

How do *we program* the computer?

To answer this question, a few words about computers:

- *What* they are
- *How* they work

Properties of a modern computer

- What your need to know about computers, they are:
  - **Stupid**: They can only do **simple operations** with **binary data**: Read, Write, Add, Subtract, Multiply, Compare,... This is what we call *computing*
  - **"Fast"** (compared to us): They can do it **billions of times per second** (GHz).

> *Note: They are still "slow" compared to biological systems like cells, where billions of complex chemical reactions happen in parallel.*

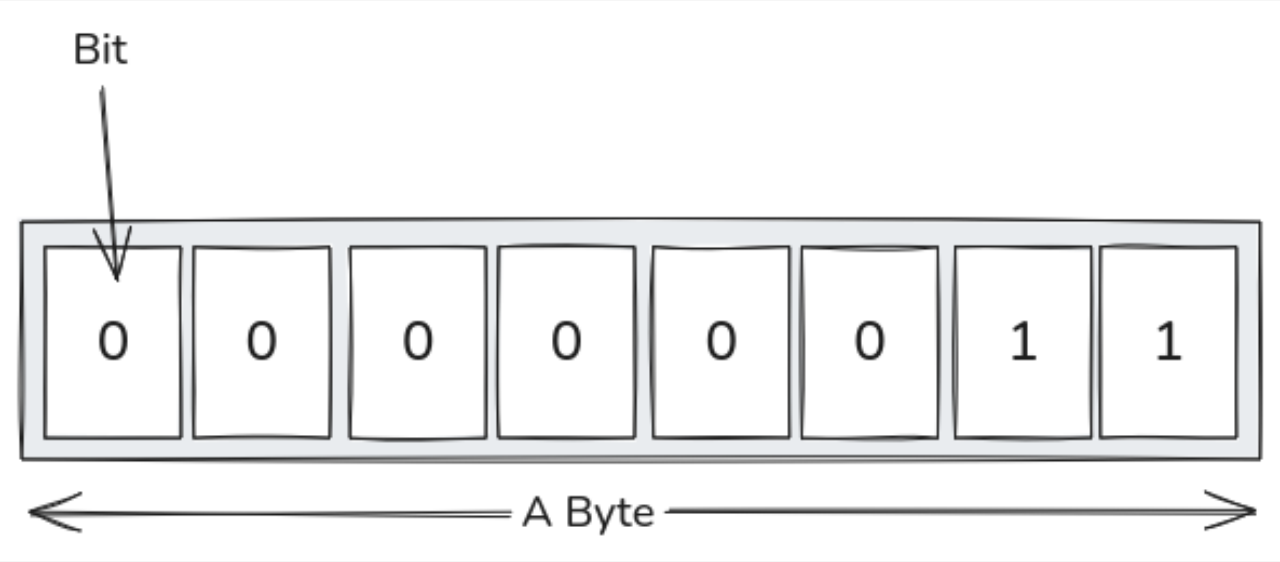
What is a computer ?

- Computers are **electronic machines** able to execute very simple operations
- Computers only understand *binary code*, called **Machine code**
- Machine code is binary patterns that encodes signals for the hardware to *activate specific circuits* and *execute specific actions* (Read, Write, Add, Compare, etc.). This correspondence is called the **Instruction Set Architecture (ISA)**
- Each machine (CPU) has its own *Instruction Set*.

> Ex: `00010101 10110101` machine code for a given machine could result in the action "write the value 5 to memory" or "compare 72 to 16".

Bits don't Byte

- **Bit**:
  - Short for *binary digit*. The **fundamental block of information**. A bit can only be one of two values: 0 or 1.
  - With a base-2 symbol system, called *binary*, you can *say/encode anything*!
- **Byte** (computer):
  - Group of **8 bits**
  - The **basic unit of memory** computers work with (read/write)



> 1 *byte* gives $2^8 = 256$ unique *codes* to represent 256 different values (from 00000000 to 11111111), for example numbers from 0 to 255.

Computer architecture

- Designed in the **1940s** by John von Neumann
- Von Neumann Architecture is based on a core principle: **data and programs are stored in the same main memory** (in binary)

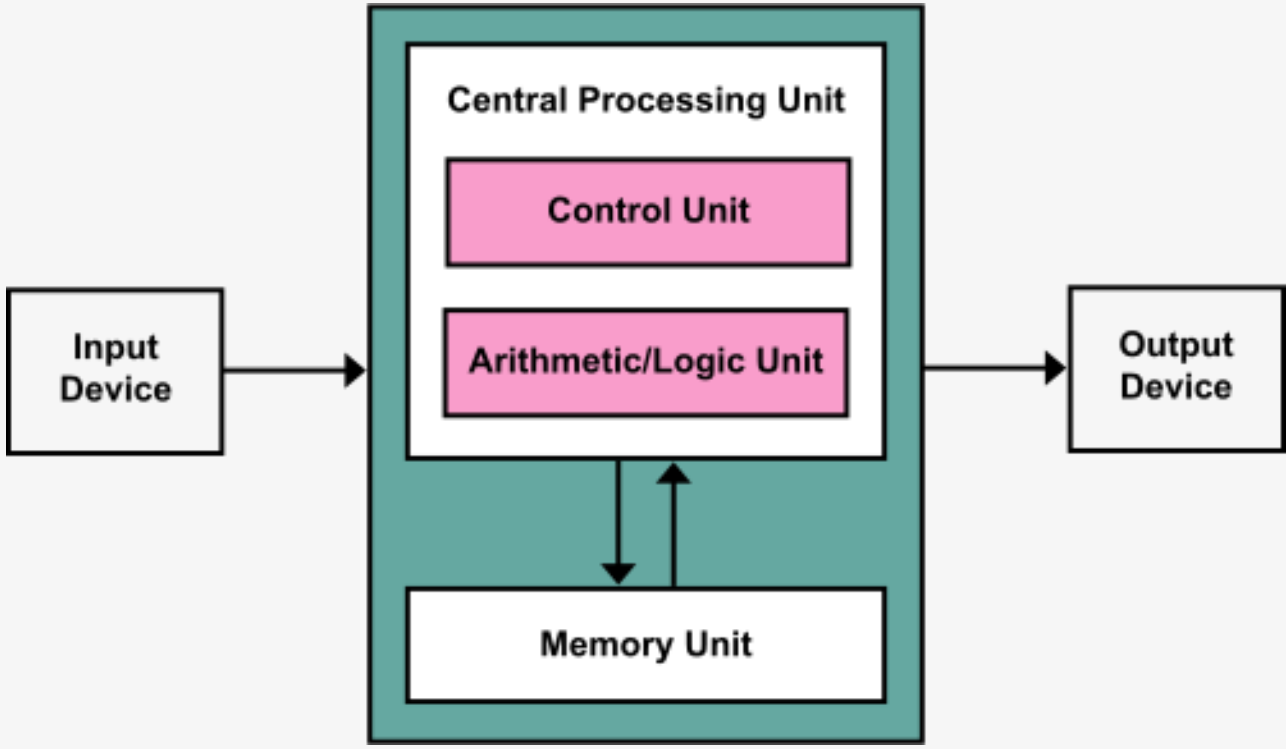
Von Neumann architecture **remains the standard computer architecture used today!**

> Prior, machines required **physical rewiring** to switch programs (One problem to solve = One machine). The Von Neumann architecture allows computers to run any program simply by loading new binary instructions into memory! This is *Soft*ware!

Fundamentals of Von Neumann Architecture

Computers has four major **components**:

- **Central Processing Unit (CPU)**: Decodes and **executes** instructions. **Read/Write from/to memory**
- **Random Access Memory (RAM)**: Giant *array* of numbered bytes (each byte has an address). **Stores** instructions and data
- **Input and Output devices (I/O)**: Used to **interact** with the *outside world* (keyboard, screen, hard drive, sensors, network card, etc.).

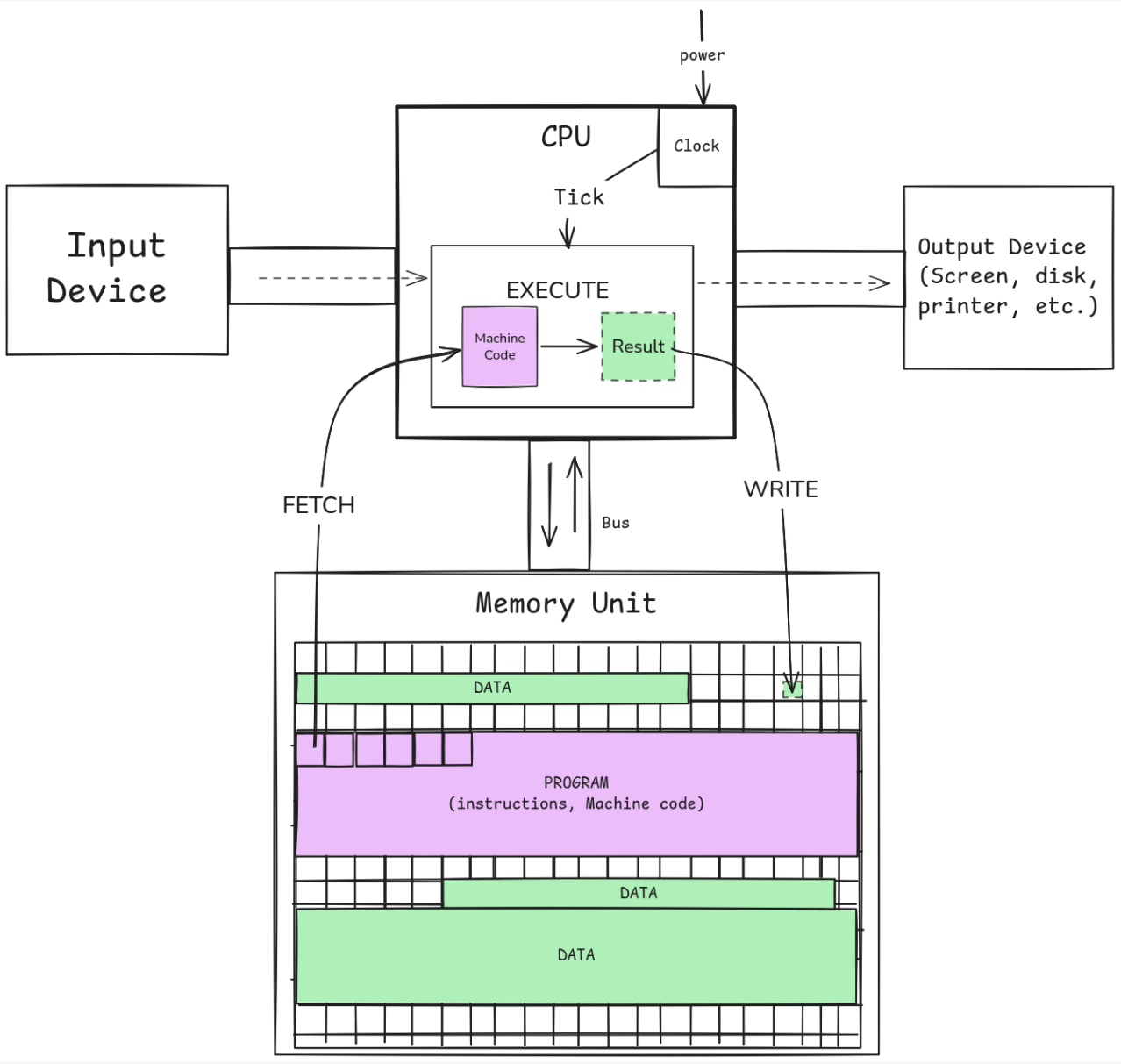


Execution cycle (simplified)

1. **Fetch**: CPU reads next **machine code instruction**, from a program loaded in memory
2. **Decode** what kind of instruction
3. **Execute**: sequence of actions to *perform* the instruction (ex: Add, Compare, etc.), defined by the instruction set of the CPU.
4. **Write**: write the result back to memory
5. Go back to step **1**.

At the end, **memory changes** (bytes are *written*, new *values* are created): program execution

> The CPU clock sets the pace, ticking *billions of times per second* (GHz) to trigger each step of the *fetch-decode-execute* cycle.



What is program execution at low level (machine level)

- A *program* consists of **a sequence of binary instructions** that drives the **memory** from an **initial state** to a **desired target state**
- The desired target state **is the result** of the program (e.g. write the correct name of the youngest person on the screen!)

How to store our program (our anything) in computer memory?

Computer memory is a *giant* array of *bytes* (8-bit boxes)

How to put *things into computer* memory?

We need to *encode* things in *binary*, using well defined *encoding systems*.

Encoding things in binary, some examples

```
#Integers
1           -> 00000001                                     (1 byte integer)
2           -> 00000010
255         -> 11111111

#Decimal (floating point number)
3.14        -> 01000000 01001000 11110101 11000011     (32-bit Float: Sign + Exponent + Mantissa)

#Text: different encoding systems exist
'A'         -> 01000001                                     (in ASCII)
'😀'        -> 11110000 10011111 10011000 10000000     (in UTF-8)

#Pixel: every pixel consists of 3 bytes for Red, Green and Blue colors:
Pure Red    -> 11111111 00000000 00000000               (RGB, 3 bytes: R=255, G=0, B=0)
Pure Blue   -> 00000000 00000000 11111111               (RGB, 3 bytes: R=0, G=0, B=255)

#Image: A big array of pixels
11111111 00000000 00000000 11111111 00000000 11111111 00000000 00000000 11111111 00000000 00000000
00000000 11111111 00000000 00000000...

#Movie : A big array of images...
```

> Everything (text, photos, programs, etc.) is ultimately just (m|b)illions of zeros and ones (binary data) being manipulated by hardware components

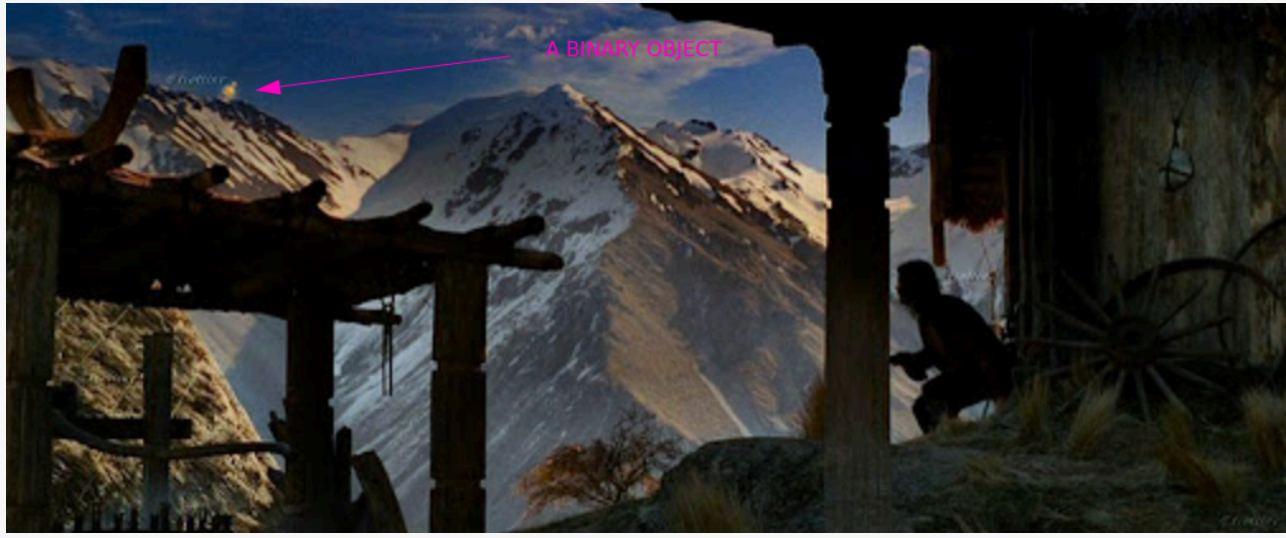
How to store a bit (for real!): a few examples of binary objects

- A *bit* is an *abstract concept* representing a choice between two mutually exclusive states: 0 or 1
- Computers need **real binary objects** to have a memory and **store binary data and machine code**
- A bit can be **stored** by a *digital device* or other *physical system* that exists in either of *two possible distinct states*.

Beacons of Gondor

Important binary objects

- Unlit (0): Gondor is not in trouble apparently
- Lit (1): Gondor *is* in big trouble!



Candle

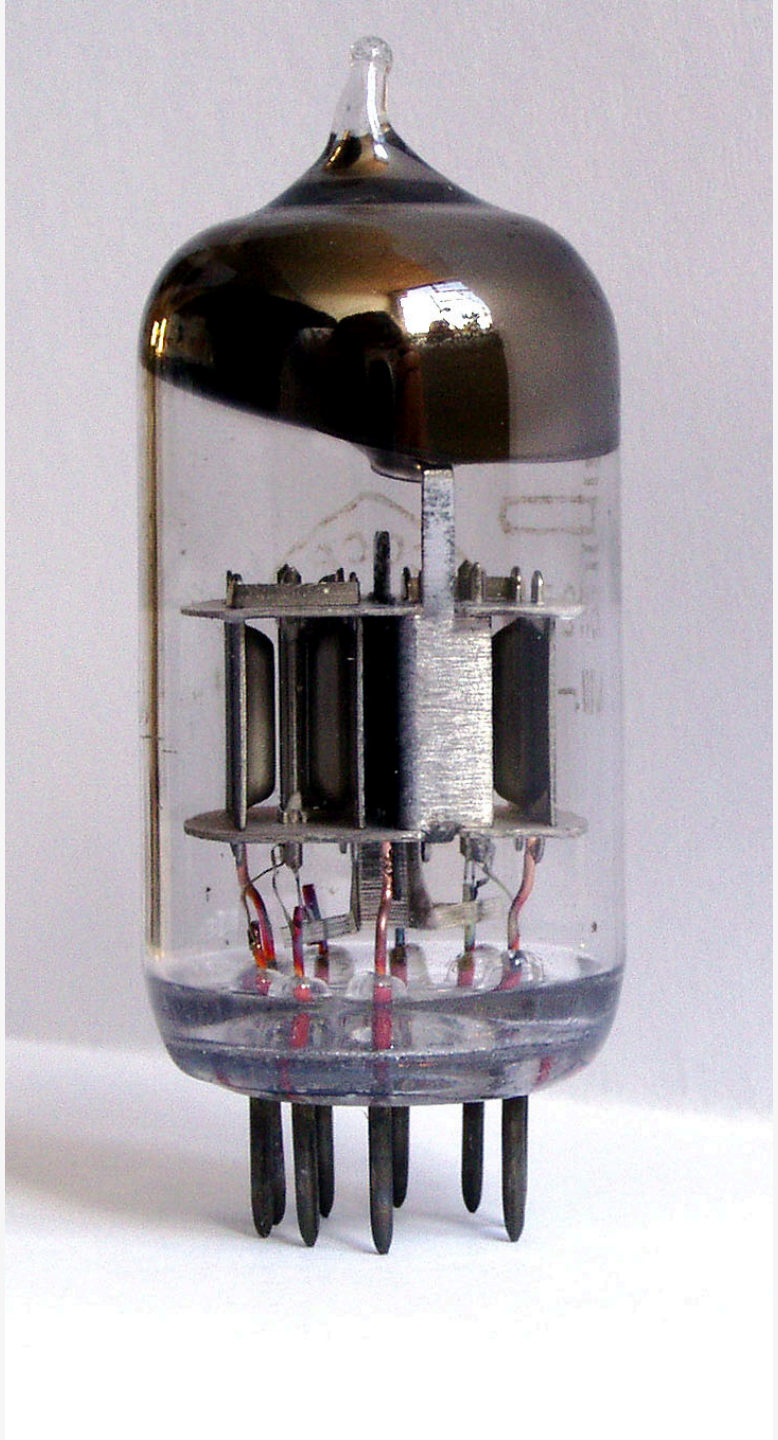
- Lit (1), Unlit (0)

> For many obvious reasons, computers don't use candles for storing bits



Vacuum tube

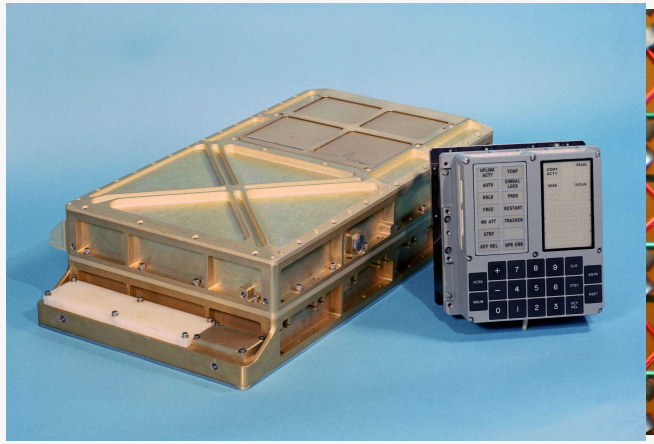
- Conducting (1), Non-conducting (0)
- Read/write access time: ~**10 microseconds**
- Bit size: ~**10 cm**
- Hard to produce, Unreliable (short lifetime), Expensive to manufacture (around $100 per unit in the 1940s!), High power consumption!

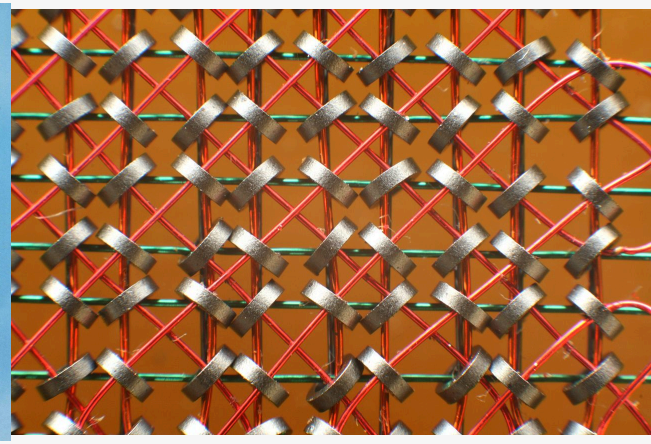


Magnetic tore (e.g. used by Apollo Guidance Computer)

- Magnetic Field direction: Up (1), Down (0)
- Read/write access time: ~**10 microseconds**
- Bit size: ~**1 mm**
- More reliable, Lower power consumption, Less fragile than vacuum tubes, Handle Radiations!

> Total Apollo Guidance Computer memory was **4 KB**. AGC is *real* software engineering



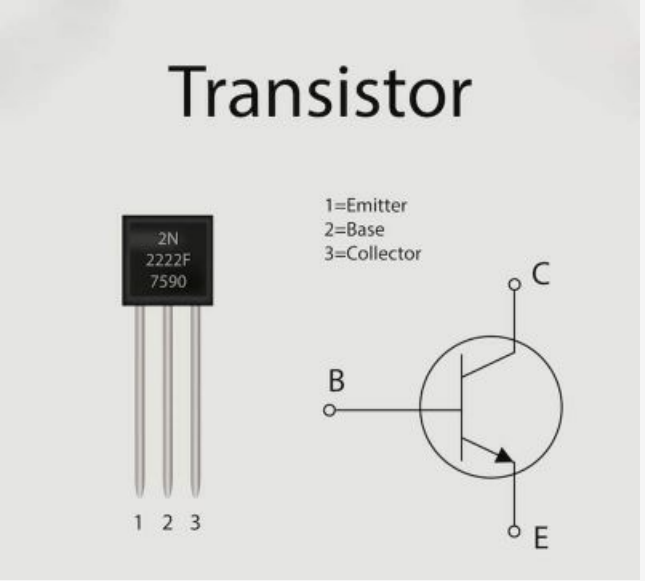


Transistor (used in modern computers!)



Typical Dynamic RAM module (Corsair DDR2-533) :

- Charged(1), discharged(0)
- Based on **billions** of tiny binary objects (memory cells 1T1C)
- Read/write access time: ~**10 ns**
- Bit size: ~**10 nanometers** !



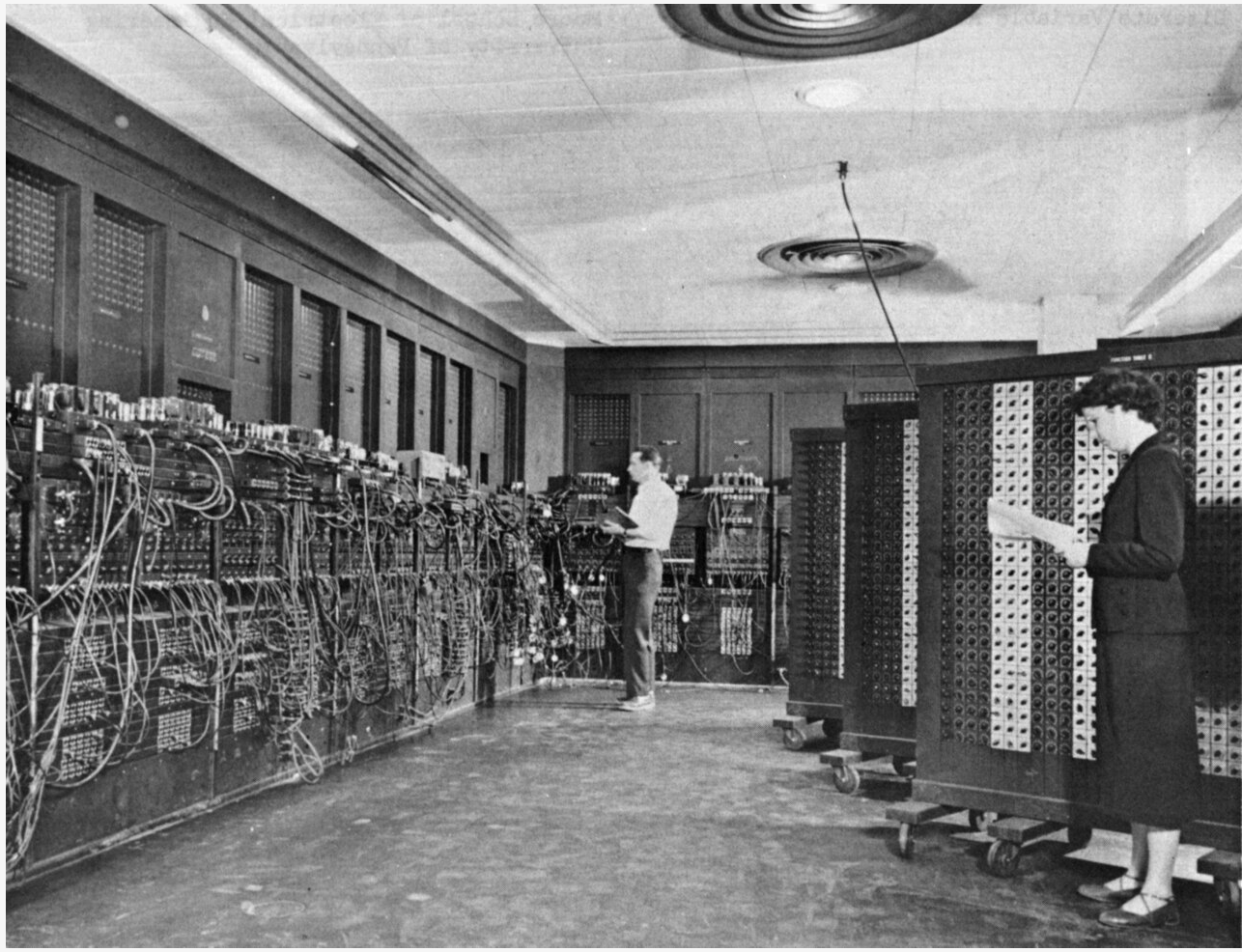
Reminder

- What do the following prefixes mean: **Kilo**, **Mega**, **Giga**?
  - Kilo: 10^3 (thousand)
  - Mega: 10^6 (million)
  - Giga: 10^9 (billion)
- What is a **Hertz**? Unit of frequency, 1 Hz = 1 cycle per second. E.g., a clock that ticks once per second = 1 Hz.

40's/50's - *Mainframes* : ENIAC

ENIAC, **1945-1950** (USA)

- **First fully electronic computer!**
- Could solve, in principle, **any numerical computing problem**
- Follows the **Von Neumann architecture**
- Weights **27 tons** and spans **170 square meters**! (uses 18,000 vacuum tubes!)



Evolution of Computing: From the 1940s to Today 1/2

Machine	Year	Processor / GHz	RAM	Storage	Price (EUROS)
ENIAC	**1945**	**~0,00001 GHz**	**20 Kb**	Magnetic drum **20Kb**	~ > **500 000 EUROS**
PDP-11	1970	~0,001 GHz	4 Kb to 256 Kb	Magnetic tape 10 to 40 Mb	~ 250 000 EUROS
Apple II	1977	0,001 GHz (1 MHz)	4 Kb to 48 Kb	Tape / 140 Kb to 5 Mb	~ 1300 EUROS
ComMbdore 64	1982	0,001 GHz (1 MHz)	64 Kb	Tape / 170 Kb to 5 Mb	~ 550 EUROS
IBM PC	1981	0,00477 GHz (4,77 MHz)	16 Kb to 640 Kb	Hard drive 10 to 20 Mb	~ 1500 EUROS
IMac G3	1998	0,233 to 0,5 GHz	32 to 512 Mb	4 to 20 Gb	~ 1300 EUROS
BlacKberry	2003	0,065 to 0,12 GHz	2 to 16 Mb	16 to 64 Mb	~ 500 EUROS
IPhone	2007	0,412 GHz	128 Mb	8 to 16 Gb	~ 600 EUROS
Lenovo ThinkPad Mbderne	**2025**	**3 to 5 GHz** (multi-core)	**16 to 64 Gb**	512 Gb to **>4 To**	**2500 EUROS**

Evolution of Computing: From the 1950s to Today 2/2

Today, a modern mainstream laptop, such as a Lenovo ThinkPad:

- is roughly a **million times *faster*** (billions of instructions per second);
- has **ten million times *more* RAM**;
- **costs about 1 million times *less*** to produce;

than a computer like the 1945 ENIAC (the first fully programmable computer)!

> Welcome to the *Space Age*!

What has changed over 80 years

- **Machine power**: today, they execute billions of instructions per second.
- **Memory capacity**: multiplied by millions, giving users the "impression" of *almost* infinite memory*.

What has BARELY/NOT changed over 80 years

- **Fundamental architecture**: **modern computers still follow the Von Neumann model** (a single memory for both data and instructions, a processing unit and a control unit);
- **How we program machines**: our programming **languages** and **concepts** remain **nearly unchanged**, even though hardware has evolved drastically.

How to encode our procedure ? 1/2

- Our procedure must be **encoded in Machine code** (binary) to be *loaded* and executable by the computer
- But, we **have the choice**, at which *level* of *abstraction* we want to code our procedure, we can use either:
  - Machine code directly! Ouch...
  - **Lower level languages**, like Assembly languages, close to machine code
  - **Higher level languages** to write *human friendly code*, and use tools **to translate it to machine code**.

How to encode our procedure ? 2/2

```
^  -> Python, PHP, Lisp, Java, R, Fortran, etc.
|  ----------------------
|  Higher level languages (closer to humans)
|  ----------------------
|  - C Language : on the border between lower and higher level languages
|  ----------------------
|  Lower level languages  (closer to hardware)
|  ----------------------
|  - Assembly languages (last human readable programming language before binary)
|  - Machine code (just bits)
```

Same program, different codes, an example

Let see the **same program** written in different codes:

- **Machine Code**: binary instructions the CPU can process directly
- **Assembly Code**: last human readable programming language before binary
- **C code**: at the interface between lower and higher level languages
- **Python code** (what we will use!): an example of higher level language

Example of Machine code (lowest level language)

```
01111111 01000101 01001100 01000110 00000010 00000001
00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000011 00000000
00111110 00000000 00000001 00000000 00000000 00000000
01010000 00010000 00000000 00000000 00000000 00000000
00000000 00000000 01000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 10011000 00110110
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 01000000 00000000
00111000 00000000 00001101 00000000 01000000 00000000
00011111 00000000 00011110 00000000 00000110 00000000
```

Example of Assembly Code (low level language)

```
.LC0:
 .string "c=%d"
 .text
 .globl main
 .type main, @function
main:
.LFB0:
 .cfi_startproc
 pushq %rbp
 .cfi_def_cfa_offset 16
 .cfi_offset 6, -16
 movq %rsp, %rbp
 .cfi_def_cfa_register 6
 subq $16, %rsp
 movl $2, -4(%rbp)
 movl $21, -8(%rbp)
 movl -4(%rbp), %eax
 imull -8(%rbp), %eax
 movl %eax, -12(%rbp)
 movl -12(%rbp), %eax
 movl %eax, %esi
 leaq .LC0(%rip), %rax
 movq %rax, %rdi
 movl $0, %eax
 call printf@PLT
 movl $0, %eax
 leave
 .cfi_def_cfa 7, 8
 ret
 .cfi_endproc
```

Example of C code ("higher" level language)

```c
#include<stdio.h>

int main(){
    int a = 2;
    int b = 21;
    int c = a * b;
    printf("c=%d", c);
}
```

Example of Python code (higher level language)

```
a = 2
b = 21
c = a * b
print(f"c={c}")
```

Which programming language to choose?

As always, *it depends* on what you want to do!

There are more than 8000 programming languages available today. Almost all of them allow you to write *any* kind of program.

However, programming languages differ in their *properties*, *advantages*, and *application domain*. Programming languages can be:

- High-level or Low-level
- Interpreted or compiled
- General purpose or domain-specific

Lower level languages

Hardware details are exposed (CPU architecture, registers, memory addresses). Memory is managed explicitly. Perfect for embedded systems. Examples: Assembly languages, C/C++ (from a modern perspective).

- **Pros**:
  - **Performance and optimization**: Direct control over execution speed and resource usage (**Manual Memory Management**)
  - What you write is (almost) directly what the hardware executes
  - Interact directly with hardware interfaces
- **Cons**:
  - Need to compile to every CPU architecture (ISA)
  - Must **understand computer (CPU) architecture**.
  - **Manual Memory Management**

Higher level languages

Abstract away hardware details. Focus on **problem-solving logic** rather than *how* the computer works (CPU, RAM). Examples: Java, JavaScript, Lisp, **Python!**, etc.

- **Pros**:
  - "Closer" to the problem to solve, **stronger abstractions** (loops, data structures, procedures/functions, modules, etc.), more *human friendly*
  - **Portability**: *Write once, run anywhere* (cross-platform*).
  - **Automatic Memory Management** via *Garbage Collection* and automatic allocation.
- **Cons**:
  - Slower execution speed and higher RAM usage.
  - Hardware is hidden behind layers of abstraction.

> Python IS higher level language

Interpreted and Compiled languages

- **Compiled languages** : the entire high-level program is **transformed directly** into *Machine Code* (`01001000...`) by a **compiler** *prior* to running. Produces an executable file tailored directly to the CPU instruction set, yielding maximum execution performance and determinism. Examples: C, C++, Fortran, Go, Rust
- **Interpreted languages** : a *pre-compiled program*, called an **interpreter**, **reads and translates code line by line** into intermediate code (*bytecode*), **executing instructions dynamically** *on the fly*. Offers flexibility and ease of development, but adds a translation layer above hardware execution and requires to load the interpreter in memory. Examples: Lisp, PHP, JavaScript, Python

> Python IS an interpreted language

Python

In this course, we will **use** the Python Programming language to write our programs.

We need to learn its **language syntax**. It will be our *vessel* to explore programming.

> Syntax changes *slightly* from one language to another, **concepts and fundamentals don't!** Without knowing, by learning Python, you will be learning many other languages *at the same time*!

Python in short

- Created in 1989 by Guido Van Rossum(Netherlands). Version 1 released in **1991**
- **Version 3+ (today)**, released in 2008
- Qualities :
  - *Easy* to read, to *learn*
  - *General purpose*
  - *High level*, *Interpreted* (Cross-platform!), *Dynamic*
  - *Multi-paradigms* (procedural, functional, object-oriented)
  - *Huge community*, documentation and learning materials!
  - Lot! of specialized libraries/modules: GUI (PyGTK), **Mathematics/Computing** (NumPy, SciPy), Machine Learning (TensorFlow), Deep Learning (Pytorch), **Data Visualisation** (Matplotlib), Data Analysis (Pandas), etc.

> Python is an excellent language to learn programming!

What you can do with Python?

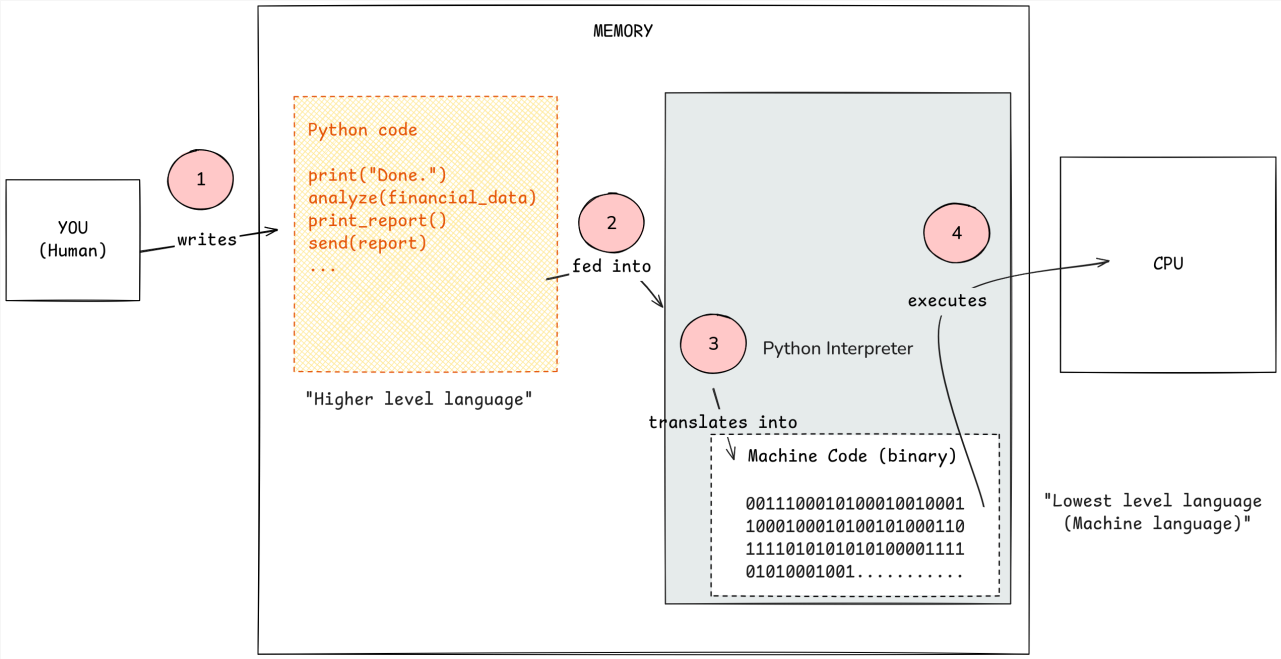
Python is a **general purpose** programming language. It's used:

- For **teaching/learning programming** (schools)
- By the scientific community (Physics, Biology, etc.) (with Fortran, R)
- For **Machine/Deep learning**, **Data Sciences** and **Artificial Intelligence** (advanced and complex libraries)
- For writing **scripts** and **automating** any kind of tasks
- To develop **complete web applications**, **desktop applications** and **video games**
- In Excel spreadsheets!

> It's a good choice!

Python is an interpreted language

A special program, called the **Python Interpreter** will *translates* and *executes* our Python code *into* machine code



Setting up our programming environnement 1/2

What we need:

- **The Python interpreter**, to run our programs!
- Package manager to install useful packages later on
- Rich **text editor** to write and read code
- A **terminal**

Setting up our programming environnement 2/2

We will use the PyCharm, an **Integrated Development Environment (IDE)**, maintained by Jet Brains (Community Version).

1. **Download** and **install** PyCharm on your machine.
2. **Launch** PyCharm
3. **Create a new script** `script.py` and **run** it. See the result in the terminal.



Why use PyCharm ?

- Community version is **free**
- *Beginner friendly*, no configuration, everything at *one place*
- **Cross-platform** (Windows, macOS, GNU/Linux)
- Manage and install the **Python Interpreter**, **Package manager** and **useful/powerful packages/modules** (for later) to write (even more) interesting programs!
- **Smart code editor** with **autocompletion** and color highlight syntax
- Jupyler Notebook Integration
- Useful developer tools: debugger, terminal, etc.

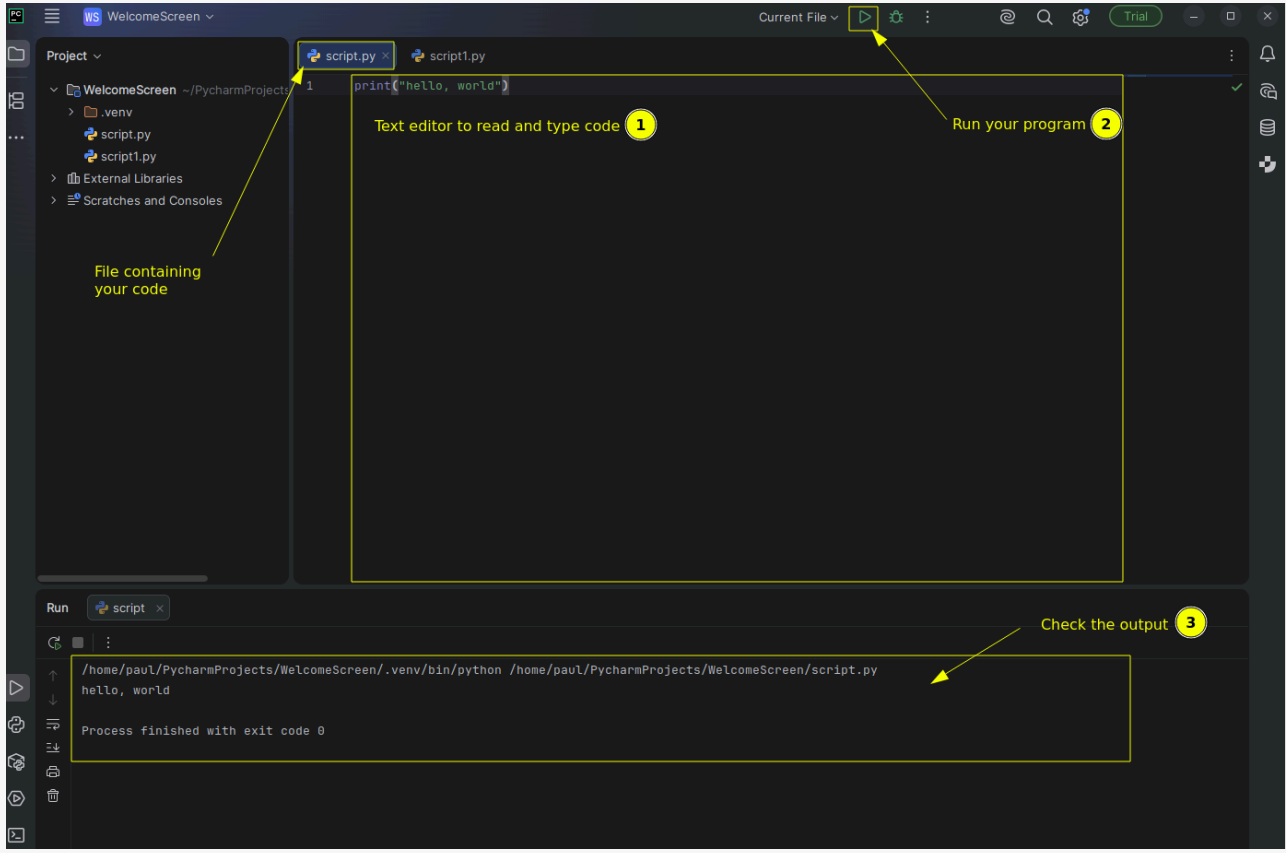
> PyCharm is a sophisticated program! We will only use basic functionalities for this course.

Programming in PyCharm

- Programs are written in files called *scripts*, with a `.py` extension (by convention).
- To run a program, we only have to call the **Python interpreter** (*Play button* in PyCharm):

```
# In the terminal, inspect Python version installed (3+)
python -V
Python 3.14.7
# Run a program written in my_program.py
python my_program.py
```

> Be sure to be able to **master the full cycle**: write/read code, run your program, see the output



Our "Find the youngest person in the group" program... in Python

Disclaimer:

- Try to get the *big picture of the following code* (the different *parts* and the role of each *section*)
- There are a *lot of details*! It's completely normal if you *don't* understand everything for now, *don't worry*!

At the end of the next module **you will be able to write such programs yourself**!

Run "Find the youngest person in the group" program

1. **Copy/Paste** the program in a *new script* called `find_youngest.py`
2. **Run** it in PyCharm and **check** the result in the terminal

```
group = {
    "Alice": "1995-04-12",
    "Bob": "2002-11-23",
    "Charlie": "1988-08-05",
}

youngest_name = None
youngest_date = None

for name, date_str in group.items():
    if youngest_date is None or date_str > youngest_date:
        youngest_name = name
        youngest_date = date_str

print(f"The youngest person is {youngest_name}, born on {youngest_date}.")
```

Make some changes

1. **Add two new members to the group**:

```
- Frank, born on September 1st, 2011 ("2011-09-01")
- Grace, born on March 20th, 1975 ("1975-03-20")
```

2. **Run** the program. **Check** the result.
3. Now, we would like to find the **oldest person** instead of the youngest:
    i. **Copy** the program to a new file called `find_oldest.py`
    ii. **Modify** the code so it **finds** and **displays the oldest person** in the group. **Hints**:
        a. **Rename** the variables from `youngest_`... to `oldest_`....
        b. **Change** the `>` operator to `<` in the `if` statement (an older birth date comes *before* in time, so it is *smaller*).
        c. **Update** the sentence in the `print()` statement in order to make sense. **Run** and **check** the result.

> Making mistakes and obtaining errors is OK! It is the best and only way to learn. Be sure to read carefully errors thrown by the interpreter

Discussion

See the *Find the oldest person* python program

On the usage of Gen. AI (LLMs)

- Use it to :
  - **Learn**
  - **Explain** something (a concept, a copy/paste piece of code)
  - **Debug** some code
- **Don't use it to code/work for you !** Writing code has never been the problem
- You **must** be able to **Understand**, **Explain** and **Modify** your code **without AI**. You are **responsible** of your code and what it does.
- **Make mistakes**. In programming, **you can test your understanding** by **executing your program**! You *will* make mistakes and that's ok. A (good) program is written iteratively.

> Don't worry, you will have PLENTY OF TIME **to be productive in your (future) job!** We are here to **LEARN**, not to be productive.

Learning Space vs Production Space

- **Learning space**: **Learn** by making mistakes, **Receive feedback**, **Do things manually**, Use simple or specialized tools, Work on **small-scale problems** and systems, **Follow provided specifications** (labs, exams, projects)
- **Production space**: **Gather requirements**, **specify a solution**, Respect constraints (deadline, budget) and tradeoffs (quality/cost), **Communicate**, Use **more complex environments and tools**, Take **responsibility**, Deliver, Deploy to production, Monitor, Maintain, **Work on large and complex systems**

Next Module

Go to next module

Going Further/Extra information

About Von Neumann, the *man from the future*

Von Neumann has made major contributions to quantum mechanics, functional analysis, mathematical logic, quantum logic, computer science, economics, etc! He also participated in U.S. military programs (Manhattan Project).

> See The Man from the Future: The Visionary Life of John von Neumann, by Ananyo Bhattacharya, 2021, Allen Lane



Dynamic vs static languages

The difference comes down to **when variable types are checked**: at compile time (before running) or at runtime (while running).

- **Statically Typed languages**:
  - **Types** are associated with variables and **determined at compile time**. Once a variable is declared as an integer, it cannot hold a string. Examples: C, C++, Java, Rust, Go, Swift.
  - Compilers can detect errors **before** running the program (at compile time)
- **Dynamically Typed Languages** :
  - **Types** are associated with values, not variables, and **checked at runtime**. A single variable can hold an integer, then a string, then a list, etc. Examples: **Python**, JavaScript, Ruby, PHP.
  - Faster to write, with less boilerplate code. Ideal for rapid prototyping.
  - Type errors cause crashes **while the program is running**!

> Python IS a (strong) dynamically typed language

Demonstration: from high level code to binary code in C

Compilation of a C program with gcc compiler.

1. **High level C code** (`main.c`)
2. After compilation (`cc -S main.c`), **assembly code** (`main.s`)
3. After assembling (`cc main.c`), **binary code** (`a.out`). See binary code in hexadecimal format with `hexdump -C a.out`

> See full demo and documentation