UNIVERSITÉ PARIS SCIENCES & LETTRES

Introduction to Computing

The Art of Computer Programming

Pierre Senellart

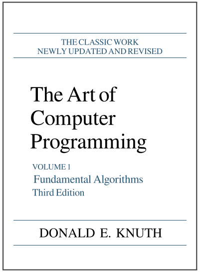
14 September 2026

Outline

- Introduction
- Computing
- Python, C, C++
- Data representation
- Variables
- Expressions
- Basic I/O
- Coda

Introduction

The Art of Computer Programming



Name of this course: Reference to a very famous book series (continuously written since the 1960s!) about algorithms by 1974 Turing award laureate Donald Knuth (b. 1938)

What you are going to learn: Computer programming, but not only; also introduction to:

- **computational thinking**
- **algorithms**, **data structures**
- evaluation of the **quality** of an algorithm
- basics of **software engineering**

Distinguishing factor: Focus on **principles**, **good practices**; instantiation to 2–3 programming languages: **Python** and **C** (along with a subset of C++)

Why does this matter?

- Understanding and designing AI systems require as much skill in **maths** as in **programming**, **software engineering**, and **algorithms**
- Critical to become **fluent in programming** to acquire the habit of **computational thinking**: "How can I automate this?", "How can I make this faster?", "How to solve this problem step by step?"
- Some of you will specialize in fields where programming is a **daily activity**
- Some will not, but still need to be able to resort to **programming as a tool**

Why learn to program when LLMs can code?

- **Training:** LLMs write **plausible** code; deciding whether it is **right** requires the very skills you acquire by programming yourself; you cannot supervise what you could not do – calculators did not remove the need to learn arithmetic
- **Responsibility:** you are **accountable** for the code you deliver (bugs, security, legal and ethical consequences); “the AI wrote it” is not a defense
- **Control:** “vibe coding” means **losing control** of the system you are designing; when it breaks, only someone who can read, debug, and modify the code can fix it
- **Specification:** the hard part of programming is saying **precisely** what you want, to a compiler or to an LLM; this requires knowing the concepts and the vocabulary
- **Limits:** LLMs are weakest exactly where you will need them most: **novel** problems, performance-critical or low-level code, code that must be **verified**
- **Durability:** models and tools change every few months; the **fundamentals** you learn here do not

This semester

Monday mornings (09:00–12:15) **Lectures** & 10-min **flash quizzes** (with Pierre Senellart)

Wednesday afternoons (13:45–17:00) **Tutorials** (hands-on exercise sessions) on your own computer, in two groups (with Changhong Wang for 6 weeks then Angelo Saadeh for 8 weeks for one group; with Panagiotis Patsilinakos for the other group)

Friday November 6 (afternoon) 1.5-hour **mid-semester exam**

Monday January 25 2-hour **final exam**

Course evaluation

45% of the grade Final exam
35% of the grade Mid-semester exam
20% of the grade Weekly flash quizzes

Tutorials are not graded – but critical for you to practice.

Where to find help?

By order of priority:

- Refer to **lecture materials** (see the Moodle site or `https://pierre.senellart.com/enseignement/2026-2027/programming-1/`)
- Check **reference documentation** (see links on Moodle)
- Ask **us** (your tutors and Pierre)
- Discuss with your **classmates**
- Read/consult **textbooks** (see references further)
- Search on the Web, especially on Stack Overflow (but keep a critical mind!)

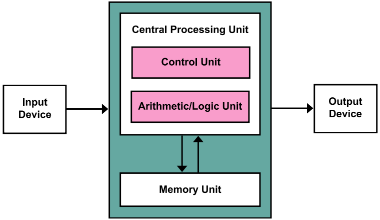
A note about LLMs

- LLMs are good at generating content that is **similar to what they were trained on**
- $\Rightarrow$ they are usually **quite good at simple programming exercises**
- However:
  - you're **not going to learn anything** if you're asking LLMs to solve problems for you
  - their solution may be **hard to understand**
  - they may use programming language or algorithmic constructs **outside of the scope** of this class
  - they may be **subtly wrong**, and you will not notice if you cannot read the code
- **Not recommended**, except in situations you are confident enough in your programming abilities to verify and **fully** understand their solution

UNIVERSITÉ PARIS SCIENCES & LETTRES

Computing

What is a computer?



Simplified architecture of a computer, as envisioned by John von Neumann (1903–1957), one of the founding fathers of computer science.

- Device able to perform **computation** (*computer*), to process **information** (*informatics*), to control a computation process (*ordinateur*)
- Has to be **programmable**: its user can specify which (arbitrarily complex) computation to perform
- Must include:
  - a **CPU**, performing computations and controlling the flow of the program
  - **memory**, storing (short-term or long-term) the program itself and the data on which it operates
  - an **input device** to load data and program
  - some form of **output** for the result

Digital electronic computer

- By far the most common form of computer nowadays
- Based on **electronic components** (transistors, capacitors, resistors, diodes, etc.) arranged in **integrated circuits** (aka **chips**)
- Storage and processing of information based on **digital binary signals** (discretization of electric levels into 0s and 1s)
- Components:
  - a **CPU**, along with other integrated circuits (**chipset**, **GPU**, etc.) for computation, control flow, and management of other devices
  - random-access memory (**RAM**) for short-term memory storage (CPU cache, main RAM, video RAM) and **solid-state** or **magnetic** drives for long-term storage
  - **keyboard**, mouse, webcam, mike, and other peripherals for input
  - **screen**, printer, speakers, headphones and other peripherals for output
- Other technologies exist to design a computer: vacuum tubes, quantum mechanics, molecular biology of DNA or RNA, etc.

Computer science, informatics

- The **science of computation and information processing**, i.e., the underlying aspects of what computers do
- Large variety of subfields, some close to **mathematics**, other closer to **computer engineering**:

Computability theory How can we define what is a computation, and which problems are computable?

Algorithmics Given a problem, what is the “best” way to solve it in a systematic way?

Machine learning How to train a system to generalize from examples and recognize patterns in data?

Software engineering How to organize, maintain, test programs within large software collections?

Human–Computer interaction How can we design efficient ways for humans to interact with a computer?

Database management How to efficiently query and manage large data collections?

...

Algorithms

- Comes from the name of محمد بن موسى الخوارزمی

Algorithms

- Comes from the name of محمد بن موسى الخوارزمی (Muhammad ibn Musa al-Khwarizmi), a Persian scientist from the 9th century

Algorithms

- Comes from the name of محمد بن موسى الخوارزمی (Muhammad ibn Musa al-Khwarizmi), a Persian scientist from the 9th century
- ... who is also at the origin of the word **algebra** (الْجَبْر, *setting* (of fractured bones) in Arabic), from the title of one of his books on equation solving

Algorithms

- Comes from the name of محمد بن موسى الخوارزمی (Muhammad ibn Musa al-Khwarizmi), a Persian scientist from the 9th century
- ... who is also at the origin of the word **algebra** (الْجَبْر, *setting* (of fractured bones) in Arabic), from the title of one of his books on equation solving
- An algorithm is a **formal specification** of how to solve a given problem: starting from an **input**, how to produce the **output** corresponding to the solution of the problem, by combining elementary operations

Algorithms

- Comes from the name of محمد بن موسى الخوارزمی (Muhammad ibn Musa al-Khwarizmi), a Persian scientist from the 9th century
- ... who is also at the origin of the word **algebra** (الْجَبْر, *setting* (of fractured bones) in Arabic), from the title of one of his books on equation solving
- An algorithm is a **formal specification** of how to solve a given problem: starting from an **input**, how to produce the **output** corresponding to the solution of the problem, by combining elementary operations
- **Algorithmics** or **Algorithm design** is the **study of algorithms**: design of the algorithm, analysis of its performance, proof of its correctness, etc.

Algorithms

- Comes from the name of محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), a Persian scientist from the 9th century
- ... who is also at the origin of the word **algebra** (الْجَبْر, *setting* (of fractured bones) in Arabic), from the title of one of his books on equation solving
- An algorithm is a **formal specification** of how to solve a given problem: starting from an **input**, how to produce the **output** corresponding to the solution of the problem, by combining elementary operations
- **Algorithmics** or **Algorithm design** is the **study of algorithms**: design of the algorithm, analysis of its performance, proof of its correctness, etc.
- **Programming** is the way to transform (we say **implement**) an algorithm into code in a programming language, to execute the algorithm on concrete data

Algorithmics and Programming

- Every algorithm is **implementable**: it must be described in terms precise enough so that there is no ambiguity about how to transform it into a program
- ... but that does **not** mean that the program implementing the algorithm is **easy to write**, because the programmer must take into account machine limitations, language specifics, low-level objects rather than high-level concepts, etc.
- **Algorithm:** abstraction of what is **implementable**
- The programming language has no influence on what is implementable; all programming languages have the same **expressive power** (they are said to be **Turing-complete** as they have the same power as the abstract model of a **Turing machine**)
- ... but a language does have an impact on the **ease** (cf. `https://pierre.senellart.com/other/languages/languages.xml`) or on the **efficiency** of the implementation

Data structure

- A **basic element** used in more complex algorithms, reused in various algorithms to solve different problems
- Formal specification of an abstract mathematical **object** (list, set, graph, matrix, etc.), the possible **operations** on this object (insertion, enumeration, inversion, etc.), and the **algorithms** realizing them
- An **implementable** element, often in the form of a **class** in object-oriented programming
- It is often possible to design different data structures for the same mathematical object, more or less **efficient** for a given task

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):
  - the **compiler** transforms the program into language-specific **bytecode**

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):
  - the **compiler** transforms the program into language-specific **bytecode**
  - the **interpreter** executes the bytecode instruction by instruction

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):
  - the **compiler** transforms the program into language-specific **bytecode**
  - the **interpreter** executes the bytecode instruction by instruction
- In **compiled** languages (like C):

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):
  - the **compiler** transforms the program into language-specific **bytecode**
  - the **interpreter** executes the bytecode instruction by instruction
- In **compiled** languages (like C):
  - the **compiler** transforms the program into machine-specific assembly code

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):
  - the **compiler** transforms the program into language-specific **bytecode**
  - the **interpreter** executes the bytecode instruction by instruction
- In **compiled** languages (like C):
  - the **compiler** transforms the program into machine-specific assembly code
  - the **assembler** transforms the assembly code into machine code directly executable by the processor

Who does what?

- The **algorithm designer** designs, describes, and analyzes an algorithm; the algorithm is typically written in **pseudo-code**, a fake programming language in which we mix text and semi-formal statements, ignoring basic details
- The **programmer** implements this algorithm in a given programming language, for a given environment
- In **interpreted** languages (like Python):
  - the **compiler** transforms the program into language-specific **bytecode**
  - the **interpreter** executes the bytecode instruction by instruction
- In **compiled** languages (like C):
  - the **compiler** transforms the program into machine-specific assembly code
  - the **assembler** transforms the assembly code into machine code directly executable by the processor
- The **software engineer** integrates the program within a larger piece of software, tests it, validates it, stores it in a code repository

Example algorithm

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
for i ← 0 to n − 1 do
    if T[i] = x then
        return i
    end if
end for
return not found
```

Python Program

Python

```python
def find(x):
    for i in range(0, n):
        if T[i] == x:
            return i
    return -1
```

Python Bytecode

(CPython 3.7; more recent versions produce a slightly more complex bytecode)

```
          0 SETUP_LOOP              43 (to 46)
          3 LOAD_GLOBAL              0 (range)
          6 LOAD_CONST               1 (0)
          9 LOAD_GLOBAL              1 (n)
         12 CALL_FUNCTION            2
         15 GET_ITER
    >>   16 FOR_ITER                26 (to 45)
         19 STORE_FAST               1 (i)
         22 LOAD_GLOBAL              2 (T)
         25 LOAD_FAST                1 (i)
         28 BINARY_SUBSCR
         29 LOAD_FAST                0 (x)
         32 COMPARE_OP               2 (==)
         35 POP_JUMP_IF_FALSE       16
         38 LOAD_FAST                1 (i)
         41 RETURN_VALUE
         42 JUMP_ABSOLUTE           16
    >>   45 POP_BLOCK
    >>   46 LOAD_CONST               2 (-1)
         49 RETURN_VALUE
```

C Program

C

```c
long find(int x) {
  for(long i=0; i<1000; ++i)
    if(T[i]==x)
      return i;
  return -1;
}
```

x86-64 Assembly Code (compiled from C)

```
      push    rbp
      mov     rbp, rsp
      mov     DWORD PTR [rbp-20], edi
      mov     QWORD PTR [rbp-8], 0
      jmp     .L2
.L5:  mov     rax, QWORD PTR [rbp-8]
      mov     eax, DWORD PTR [rax*4+0x404040]
      cmp     DWORD PTR [rbp-20], eax
      jne     .L3
      mov     rax, QWORD PTR [rbp-8]
      jmp     .L4
.L3:  add     QWORD PTR [rbp-8], 1
.L2:  cmp     QWORD PTR [rbp-8], 999
      jle     .L5
      mov     rax, -1
.L4:  pop     rbp
      ret
```

x86-64 Machine Code

```
        push    rbp                             #55
        mov     rbp, rsp                        #48 89 e5
        mov     DWORD PTR [rbp-20], edi         #89 7d ec
        mov     QWORD PTR [rbp-8], 0            #48 c7 45 f8 00 00 00 00
        jmp     .L2                             #eb 1b
.L5:    mov     rax, QWORD PTR [rbp-8]          #48 8b 45 f8
        mov     eax, DWORD PTR [rax*4+0x404040] #8b 04 85 40 40 40 00
        cmp     DWORD PTR [rbp-20], eax         #39 45 ec
        jne     .L3                             #75 06
        mov     rax, QWORD PTR [rbp-8]          #48 8b 45 f8
        jmp     .L4                             #eb 16
.L3:    add     QWORD PTR [rbp-8], 1            #48 83 45 f8 01
.L2:    cmp     QWORD PTR [rbp-8], 999          #48 81 7d f8 e7 03 00 00
        jle     .L5                             #7e db
        mov     rax, -1                         #48 c7 c0 ff ff ff ff
.L4:    pop     rbp                             #5d
        ret                                     #c3
```

Python, C, C++

Programming languages

- Rich history of **programming languages**, from the 1950s onwards
- Many programming languages widely used and relevant today
- We focus on languages:
  - among the **most important** in history and nowadays
  - most **useful for AI** students
  - that illustrate well important **computer science topics**
- Once you are fluent in a programming language, learning new ones is much easier

Python

- Programming language initially designed by **Guido van Rossum** (Dutch programmer) in the early 1990s, now managed by the **Python Software Foundation**
- Major change in 2008 with the advent of Python 3.0, **not backwards-compatible** with previous Python versions; regular evolutions of the language since (most recent stable version in September 2026 is 3.14)
- **High-level** language, with a focus on code readability, over the ability of being close to low-level machine code and raw efficiency
- Several existing implementations of Python but the reference one and most commonly used by far, CPython (free and open source software), uses **compilation to bytecode**, then **interpretation of the bytecode**
- 2025 Stack Overflow developer survey: 2nd most popular programming language among developers (after JavaScript/TypeScript)
- By far the programming language of reference for **data science applications**

C

- Programming language initially designed by **Dennis Ritchie** (American computer scientist and 1983 Turing award laureate) in 1972, now standardized by a committee of the International Organization for Standardization (ISO)
- Major new standards in 1999 (C99), 2011 (C11), and 2023 (C23) – we will use C23, though mostly features already present in C11
- **Low-level** language, with a focus on efficiency and precise memory management
- Several existing implementations (all through direct **compilation to machine code**), including GCC, Clang (both free and open source software), and Microsoft Visual C++ (proprietary software, free to use in Visual Studio under Windows)
- 2025 Stack Overflow developer survey: 6th most popular programming language among developers (after JavaScript/TypeScript, Python, Java, C#, C++)
- Language of reference for **embedded applications** and other low-level development (e.g., in **operating systems**)

C++

- Programming language initially designed by **Bjarne Stroustrup** (Danish computer scientist) in 1979 as "C with Classes", now standardized by a committee of the International Organization for Standardization (ISO)
- Major new standards in 1998 (C++98), 2011 (C++11), 2017 (C++17), 2020 (C++20), and 2023 (C++23) – we will mostly use features of C++17
- Designed to keep C's **low-level** features, focus on efficiency and precise memory management, along with high-level programming features (object-oriented programming, generic programming, etc.); C is *almost* a subset of C++
- Very **rich and complex** programming language, we will only see a small fraction of its features, mostly to complement C
- Same implementations as for C (GCC, Clang, Microsoft Visual C++), all through direct **compilation to machine code**
- Very popular language for applications where **both performance and high-level features** are important (e.g., video game engines, deep learning models)

A Python, C, C++ program

- Programs are written as **plain-text files**, with any text editor (Notepad, Emacs, Vim, etc.) or integrated development environment (IDE, such as Visual Studio Code, Eclipse, Xcode) – we will use Visual Studio Code (free, for Windows, macOS, Linux) in tutorials
- One program: **one or more** such text files, possibly along with other files
- Python, C, C++ programs can use functionalities provided by their **standard library** (a library of functions accompanying the core language) or from **third-party libraries** (programs developed by others and meant to be integrated within a user's code)
- Programs can include a **main** function, which defines the code that will be executed when the program is launched; optional in Python (by default, everything in the file is run), required in C and C++
- Programs are formed as a succession of **statements** or **instructions**: in Python, a newline completes a statement; in C or C++, a final semicolon (`;`) does, and line breaks are irrelevant

Importing library functionalities

To use a feature from a library *mylibrary* from the standard library, one needs to add (usually at the beginning of a file) declarations such as:

Python

```python
import mylibrary
```

C

```c
#include <mylibrary.h>
```

C++

```cpp
#include <mylibrary>
```

Comments

A program may include **comments**, which are just text meant for a human reader, and do not impact the compiler or interpreter in any way.

- In Python, everything that starts with `#` till the end of the line is a comment
- In C and C++, everything that starts with `//` till the end of the line is a comment
- In C and C++, everything in between `/*` and `*/` (possibly spanning multiple lines) is a comment

"Hello world" in different languages

Python

```python
print("Hello world")  # This is a comment.
```

C

```c
#include <stdio.h>

int main() {
  printf("Hello world\n"); /* This is a comment. */
  return 0;
}
```

C++

```cpp
#include <iostream>

int main() {
  std::cout << "Hello world" << std::endl; // This is a comment.
  return 0;
}
```

Data representation

Refresher: Bases

- We usually work in base 10 (**decimal**), but other bases, notably 2 (**binary**) and 16 (**hexadecimal**), are often more convenient to deal with data in digital electronic computers; we occasionally write the base as $\underline{\text{number}}_{\text{base}}$

Decimal base	Digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Binary base	Digits are 0, 1
Hexadecimal base	Digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- By definition, $\underline{42}_{10} = 2 \times 10^0 + 4 \times 10^1$
- Similarly:
  - $\underline{101010}_2 = 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 = 2^1 + 2^3 + 2^5 = 2 + 8 + 32 = \underline{42}_{10}$
  - $\underline{42}_{16} = 2 \times 16^0 + 4 \times 16^1 = 2 + 64 = \underline{66}_{10}$
  - $\underline{\text{BF}}_{16} = 15 \times 16^0 + 11 \times 16^1 = \underline{15}_{10} + \underline{176}_{10} = \underline{191}_{10}$
- A hexadecimal number with n hexadecimal digits transforms into a binary number with $\leqslant 4n$ binary digits

Binary data

- Digital electronic computers manipulate **bits** (or **binary digits**), i.e., every piece of information is formed of 0s and 1s
- Since a single bit is not much, bits are grouped together in larger structures:

  Byte sequence of 8 bits (or two hexadecimal digits!)

  Multibyte word a sequence of multiple bytes; e.g., in 64-bit architectures (most common nowadays), a native word is 64 bits, or 8 bytes
- Within a byte, bit 0 is the least significant; bit 7 is the most significant (e.g., 11000010 is a binary representation of the decimal number 194 with bits 1, 6 and 7 set to 1 and the rest to 0)
- Within a word, the most significant byte depends on the **endianness** of the architecture:

  Little-Endian (LE) architecture the first byte is the least significant (most common)

  Big-Endian (BE) architecture the first byte is the most significant one (rare)

Endianness example

Example (32-bit word)

bit 2 (within Byte 0 (LE))

bit 7 (within Byte 3 (LE))

0	1	0	1	1	0	1	1	1	0	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0

Byte 0 (LE) Byte 3 (BE)	Byte 1 (LE) Byte 2 (BE)	Byte 2 (LE) Byte 1 (BE)	Byte 3 (LE) Byte 0 (BE)

Binary data representations

- In programs, one wants to manipulate data that is more complex than just bits or binary words, starting with:

  integers (nonnegative or arbitrary), for counters, indices, etc.

  approximations of real numbers for numerical computations

  text in the form of characters or finite sequences of characters (called **character strings**)
- How are these data represented as binary data?
- Also, more complex data types; we will cover them in a later lecture

Booleans

- **George Boole:** 19th century mathematician who came up with an **algebra of truth values**
- **Boolean**: a representation of a truth value, either **True** or **False**
- Often natural outcome of data processing; e.g., checking whether two integers are equal results in a Boolean value
- Binary representation is **trivial**: just use a bit!
  - 1 means True
  - 0 means False

Unsigned integers

- Nonnegative integers are called **unsigned** integers in computing
- Two strategies:
  - Fix an upper limit on the value of the integer (aka **fixed-size unsigned integers**), say $2^k - 1$
    - All integers between 0 and $2^k - 1$ can be represented as a sequence of k bits; k is usually chosen to be a multiple of 8, to have whole bytes

Bits	Bytes	Range of integer
8	1	0 to 255
16	2	0 to 65 535
32	4	0 to 4 294 967 295
64	8	0 to 18 446 744 073 709 551 615

    - **Advantage:** the **CPU already knows** how to add, multiply, divide, etc., unsigned integers of 8, 16, 32, or 64 bits
  - Allow integers of **arbitrary value** (aka **arbitrary-size unsigned integers**) – but then:
    - the number of bits required to represent them is **not fixed any more**
    - **Arithmetic is more expensive**, does not map to a single CPU operation

Signed integers

- Arbitrary integers are called **signed** integers in computing
- Several ways of representing the sign of the integer n:
  - **Sign–Magnitude:** represent the absolute value $|n|$ using unsigned integer binary representation, and add one bit to indicate whether $n = +|n|$ (bit = 0) or $n = -|n|$ (bit = 1)
    - Quite **simple**!
    - **Problem:** 0 has two representations +0 and −0
    - With k bits, can represent integers **between** $-2^{k-1} + 1$ **and** $2^{k-1} - 1$
  - **Two's complement:** zero and positive numbers are represented as unsigned numbers; to represent a negative number $-n$, first **compute** n, then **invert all bits**, then **add one to the result**, with regular addition on binary numbers. With k bits, can represent integers **between** -2^{k-1} **and** $2^{k-1} - 1$ (and 0 has a single representation). For example, using 8 bits, 42 is 00101010 and −42 is 11010110.
- Modern **CPUs already know** how to add, multiply, etc., signed integers with **two's complement representation** of common sizes

Integer overflow

- What happens when a computation leaves the range of a fixed-size integer?
  - **Unsigned integers** **wrap around**: the result is taken modulo 2^k; on 8 bits, $255 + 1 = 0$ and $0 - 1 = 255$
  - **Signed integers** in C/C++: **undefined behavior**, the standard makes no promise; in practice the value usually wraps around, e.g., on 32 bits $2\,147\,483\,647 + 1$ gives $-2\,147\,483\,648$
  - The compiler does **not** warn you, and neither does the program: the wrong value is silently used
- Consequence: choose an integer type whose range covers all values of the computation, including intermediate ones
- Python integers have arbitrary size: **no overflow**, at the cost of slower arithmetic

Floating-point numbers

- **Impossible** to represent an arbitrary real number (with infinite decimal expansions) using a finite representation (**rational numbers** can be represented as **pairs of integers**, numerator and denominator)
- For numerical computation, possible to store a **floating-point number**: an approximation of a real number inspired by scientific notation (e.g., $6.02214076 \times 10^{23}$: a fixed number of significant digits, and an exponent)
- **IEEE-754**: standard explaining how to do this using 32-bit, 64-bit, 128-bit floating-point numbers, everything in **binary**: a number is written as $\pm(1.b_1b_2\dots b_p)_2 \times 2^e$, with a fixed number p of significant binary digits and an integer exponent e
- For example, for an IEEE-754 32-bit representation:

Original decimal number	$6.02214076 \times 10^{23}$
IEEE-754 32-bit representation	0 11001101 11111110000110000101110
Meaning of the representation	$(+1) \times 2^{205-127} \times (1 + 0.992559194564819 3)$
Actual value of the representation	$\approx 6.02214064 \times 10^{23}$

Character sets

Unicode: **character repertoire**, assigning to each character, whatever its script or language, an integer number.

Examples

A	$\rightarrow$	65	ε	$\rightarrow$	949
é	$\rightarrow$	233	ۃ	$\rightarrow$	1731

Character encoding: concrete method for representing a Unicode character.

Examples (é)

iso-8859-1	`11101001`	only for some characters
utf-8	`11000011 10101001`	
utf-16	`11101001 00000000`	

utf-8 can represent all Unicode characters, in a way compatible with the legacy **ASCII** encoding (that specifies how to represent the first 128 characters of Unicode); however, depending on the character, its utf-8 representation takes anywhere between 1 and 4 bytes.

Characters and character strings

- We **fix a character encoding** (usually utf-8, but not always)
- A **character** is represented by the sequence of bytes for this character in this character encoding
- A **character string** is represented by:
  - The sequence of bytes representing each character, one after the other
  - Some way to determine where this sequence terminates, either:
    - an unsigned integer giving the **number** of characters or of bytes
    - a **special byte** put after all other bytes, indicating the character string is finished (usually the **null byte** 00000000)
- The length in bytes of a character string **is not** its length in characters!
- **ASCII characters** (first 128 characters of Unicode: Latin letters, Arabic numerals, most common punctuation signs and special characters) are **encoded as one byte** in most (but not all!) character encodings, including utf-8; so if a string only contains ASCII characters, its byte and character lengths are equal

Variables

Variables in programming languages

- Not the same as a variable in mathematics!
- A **variable** is a **named memory location containing a value of some type**
- A variable has:

a name used to refer to that particular variable

an address i.e., an integer indicating the location of the variable in the memory of the computer (through a mapping maintained by the operating system or interpreter of addresses to actual physical locations)

a type specifying which data representation to use to interpret the value of the variable

a value the actual interpretation (for the variable type) of the data stored at the variable address

a scope which indicates from which part of the program the variable can be referred to

an extent which indicates the lifespan of the variable

Naming variables

- Rules (same in Python, C, C++): a name is made of **letters**, **digits**, and **underscores**, does **not start with a digit**, and is **case-sensitive** (`count` and `Count` are different variables)
- **Keywords** of the language (`if`, `for`, `int`, `return`, ...) cannot be used as names
- Conventions: **meaningful** names (`total_price`, not `tp`), lowercase words separated by underscores for variables and functions (**snake_case**), `UPPERCASE` for constants; short names such as `i`, `j`, `n`, `x` are fine for loop counters and mathematical quantities
- Stick to ASCII letters: accented and non-Latin letters are allowed in Python but not portable in C

Typing in Python, C/C++

- **C and C++ are statically typed languages**: the type of a variable is fixed when the variable is declared (explicitly, or – in C++ and very recent versions of C – by letting the compiler infer it from context) and never changes, nor does its address
- **Python is a dynamically typed language**: the type of a variable is just the type of whatever value is currently assigned to it, and it may change over time (as may the address of the variable)
- **Python is (informally) said to be strongly typed**: every value carries its type at runtime and is only implicitly converted to a wider numeric type (`int` to `float`); anything else is an error: `"1" + 1` fails, one must write `int("1") + 1`
- **C is (informally) said to be weakly typed**: nothing is checked at runtime, and implicit conversions are silent even when lossy (`int i = 3.99;` stores 3, `'A' + 1` is 66, `-1 < 1u` is false); a **cast** can even reinterpret a value as another type; C++ is somewhat stricter
- **Note:** Python can be turned into a statically typed language using type annotations and tools such as `mypy` – we will not cover this

Converting between types

Python: explicit conversion by calling the target type

Python
```python
int("42")       # 42
int(3.99)       # 3 (truncation)
float(3)        # 3.0
float("2.5")    # 2.5
str(3.5)        # '3.5'
7 / 2, 7 // 2 # 3.5, 3
```

Between strings and numbers, C needs library functions (`atoi`, `snprintf`, …): see a later lecture.

C/C++: implicit conversion on assignment or in mixed expressions, explicit conversion with a **cast** `(type) value`

C
```c
double d = 3;           // 3.0
int i = 3.99;           // 3 (truncation)
int m = 7, n = 2;
double q = m / n;    // 3.0: 7 / 2 is 3
double r = (double) m / n; // 3.5
```

C++ prefers the more explicit form:

C++
```cpp
// same as (double) m / n
double r = static_cast<double>(m) / n;
```

Defining and assigning variables

- Python, C, C++ all use the *name* = *value* syntax to assign a value to a variable
- In C/C++, the first time a variable is used, it needs to be defined by prefixing the name with a *type*
- The value of a variable can be used by simply using the variable name on the right-hand side of an assignment

Python

```python
i = 42
i = 'Hello'
j = 4.5
j = i
```

C

```c
int i = 42;
i = 54;
const char* s = "Hello";
double d = 4.5;
```

Primitive types in Python

Type	Example literal values	Note on representation
`bool`	`True False`	
`int`	`123 +123 123_456 -42 0b1111 0xABCD`	arbitrary-precision signed integer
`float`	`3.14159 +6.02e23`	64-bit IEEE-754
`str`	`'Hello' "Hello" 'Ça va?' """Long "text"."""`	Unicode character strings with length
`complex`	`42j`	pair of `float`s

- No difference between `'Hello'` and `"Hello"`
- Special characters are written with a backslash (**escape sequences**): `"\n"` newline, `"\t"` tabulation, `"\\"` backslash, `"\""` and `'\''` quotes
- Strings enclosed with `"""..."""` can include line breaks

Primitive types in C, C++ (1/2)

Type	Example literal values	Note on representation
`bool`	`true false`	
`int`	`123 +123 123'456 -42 0b1111 0xABCD`	signed integer of default size
`unsigned`	`123u 123'456u 0b1111u 0xABCDu`	unsigned integer of default size
`long`	`123l +123l 123'456l -42l 0b1111l 0xABCDl`	signed integer of longer size
`unsigned long`	`123ul 123'456ul 0b1111ul 0xABCDul`	unsigned integer of longer size
`long long`	`123ll +123ll 123'456ll -42ll 0b1111ll 0xABCDll`	signed integer of even longer size
`unsigned long long`	`123ull 123'456ull 0b1111ull 0xABCDull`	unsigned integer of even longer size
`short`		signed integer of shorter size
`unsigned short`		unsigned integer of shorter size

- What default, longer, even longer, shorter means is not defined by the standard, depends on the CPU, operating system, compilation environment; may be for example 32, 64, 64, 16 bits
- The `'` digit separator and the `0b` prefix in literals, as well as the `bool`, `true`, `false` keywords, require C23 (or C++14, resp. any C++)
- For this reason, C/C++ also provide fixed-size types `int8_t int16_t int32_t int64_t` (signed) and `uint8_t uint16_t uint32_t uint64_t` (unsigned) of respectively 8, 16, 32, 64 bits

Primitive types in C, C++ (2/2)

Type	Example literal values	Note on representation
`float`	`3.14159f +6.02e23f`	32-bit IEEE-754
`double`	`3.14159 +6.02e23`	64-bit IEEE-754
`long double`	`3.14159l +6.02e23l`	"longer" floating-point number
`char`	`'A' '9'`	Byte
`const char*`	`"Hello"`	Byte sequence

- Same **escape sequences** as in Python: `"\n"` newline (needed at the end of every line printed with `printf`), `"\t"`, `"\\"`, `"\""`, `'\''`
- `char` and `const char*` are about bytes, not characters, despite their names – they are suitable for ASCII character (strings); to use `const char*` for arbitrary (utf-8) strings, the character encoding needs to be manually managed

Type and address of a variable

- In Python, `type(variable)` returns a description of the type of the variable `variable`
- In C/C++, this is not required as the type is known statically
- In Python, `id(variable)` returns the address of the variable `variable` as an integer
- In C/C++, `&variable` returns the address of the variable `variable` of type `vartype` as a value of type `vartype*` (called a **pointer to a `vartype`**); if one has an address `addr` of type `vartype*`, `*addr` returns the value at the address `addr`
- One can also use `*addr` as the left-hand-side of an assignment: `*addr = val` changes the value pointed to by `addr` to `val`

Example use of pointer

C

```c
int i = 42;  // i is assigned the integer value 42
int* p = &i; // p is assigned the address of i
int j = *p;  // j is assigned the value at the address p, i.e., 42
*p = 121;    // We modify the value at the address p to 121
int k = i;   // k is set to the value of i, now 121
```

UNIVERSITÉ PARIS SCIENCES & LETTRES

Expressions

Expressions

- Everywhere a value is expected (e.g., on the right-hand side of an assignment), one can use an **expression** instead, made from variables and values
- This expression may include **operators** such as +, *, /, or call to **functions** such as `len()`
- As the expressions of Python are largely inspired from C, many operators are **common** to all three languages
- Arguments to function calls or expressions can themselves be **arbitrary expressions**
- As in maths, there are **precedence rules** to determine in which order operators are evaluated (e.g., multiplication before addition) but **parentheses** can be used to remove any ambiguity
- The **type** of an expression is the type of the value returned by this expression

Comparison operators

Operator	Example	Meaning
`==`	`i == 42`	equal to
`!=`	`i != 42`	not equal to
`<`	`i < 42`	less than
`<=`	`i <= 42`	less than or equal to
`>`	`i > 42`	greater than
`>=`	`i >= 42`	greater than or equal to

- These operators take expressions of the same types
- Cannot be used for comparing `const char*` byte sequences in C/C++
- These operators return `bool` values (or in the case of C, 0 or 1 integer values that can be transparently used as `bool`)
- Python also has operators `is` and `is not` which are synonymous for address (in)equality: `a is b` iff `id(a) == id(b)`

Boolean operators

Python Operator	Example	Meaning
`not`	`not (i == 42)`	not
`and`	`(i != 42) and (i != 21)`	and
`or`	`(i == 42) or (i==21)`	or

C/C++ Operator	Example	Meaning
`!`	`!(i == 42)`	not
`&&`	`(i != 42) && (i != 21)`	and
`||`	`(i == 42) || (i==21)`	or

- These operators take `bool` and return `bool` values (or in the case of C, 0 or 1 integer values that can be transparently used as `bool`)

Arithmetic operators

Operator	Example	Meaning
`+`	`i + 42`	addition
`*`	`i * 2.0`	multiplication
`-`	`-i`	subtraction (or unary minus)
`/`	`i / 42`	division
`%`	`i % 42`	remainder of integer division
`//`	`i // 42`	integer division (Python only)
`**`	`2 ** 8`	exponentiation (Python only)

- These operators take expressions of the same numeric (integer or floating-point) types (or types that can be converted to the same type) and return a value of the same type
- In C/C++, `/` is the integer division if both arguments are integers, the floating-point division otherwise; in Python, it is always the floating-point division

Mathematical functions in Python

- Requires `import math`
- Powers and roots:
  - `math.sqrt(x)`, `math.pow(x, y)`, `math.exp(x)`
- Logarithms:
  - `math.log(x)` (natural), `math.log10(x)`, `math.log2(x)`
- Trigonometry:
  - `math.sin(x)`, `math.cos(x)`, `math.tan(x)`
  - Inverses: `math.asin(x)`, `math.acos(x)`, `math.atan(x)`
- Other:
  - `math.fabs(x)`, `math.floor(x)`, `math.ceil(x)`

Mathematical functions in C/C++

- Requires `#include <math.h>` (C) or `#include <cmath>` (C++)
- Powers and roots:
  - `sqrt(x)`, `pow(x, y)`, `exp(x)`
- Logarithms:
  - `log(x)` (natural), `log10(x)`, `log2(x)`
- Trigonometry:
  - `sin(x)`, `cos(x)`, `tan(x)`
  - Inverses: `asin(x)`, `acos(x)`, `atan(x)`
- Other:
  - `fabs(x)`, `floor(x)`, `ceil(x)`

String functions in Python

- `len(s)` returns the number of characters in `s`
- Strings can be compared with standard Python comparison operators
- `s1 + s2` is the concatenation of the two strings (all characters of `s1` followed by all characters of `s2`)
- `s * n` is the string where `s` is repeated `n` times

String functions in C

- Requires `#include <string.h>`
- `strlen(s)` returns the number of bytes in `s`
- `strcmp(s1,s2)` compares `s1` and `s2`: returns 0 if identical, a negative or positive value otherwise depending on whether `s1` comes before or after `s2` in lexicographic order
- C++ has better functionalities for character strings (or for byte sequences), will be discussed in a further lecture

Basic I/O

Basic input and output

- A program often needs to **interact with a user**
- Can be **very complex** graphical, audio, video interfaces
- We are going to consider very simple **text-based interfaces** used within a program: a user can enter simple data in a text console, and the program can output simple character strings

Basic input in Python

Python

```python
s = input()
# s will contain the string typed by the user
i = int(s) # convert the string to an integer
```

Basic input in C

C

```c
int i = 0;
scanf("%d", &i);
// i will contain the integer typed by the user
```

Basic output in Python

Python

```python
name = 'Pierre'
age = 45
print(f"Your name is {name} and you are {age} years old.")
```

These are called **f-strings**. Every expression within curly braces is evaluated and replaced in the string with its value. (Use double curly braces if you want to write an actual curly brace character.)

Basic output in C

C

```c
const char* name = "Pierre";
int age = 45;
printf("Your name is %s and you are %d years old.\n", name, age);
```

- Requires `#include <stdio.h>`
- See the documentation of `printf` to output other types of data

Coda

Reference material

- Purely **optional**! If you want to deepen your understanding of some concepts. Goes into much more depth than this course.
- For programming languages, textbooks and tutorials are helpful to learn the language; reference documentation is useful for beginners and experienced programmers alike, to look some concept or function up

Programming	Wassberg [2020]
Algorithms	Cormen et al. [2022]
Python	Lutz [2025] and `https://docs.python.org/3/`
C	King [2008] and `https://en.cppreference.com/c`
C++	`https://www.learncpp.com` and `https://en.cppreference.com/`

More about digital electronic computers

- How everyday computers are actually implemented down to the electronic and physical level is a **fascinating subject**
- No time to cover this in class
- But if you are interested, check the following **games**:
  - NandGame
    `https://nandgame.com/`
  - Turing Complete `https://store.steampowered.com/app/1444480/Turing_Complete/`
  - Hard Chip
    `https://store.steampowered.com/app/2844290/Hard_Chip/`

In preparation for Wednesday's tutorial

- Wednesday's tutorial is **hands-on**: bring your **own laptop** (Windows, macOS, or Linux), charged, with administrator rights to install software
- Goal of the session: get a working programming environment (Visual Studio Code, Python, a C/C++ compiler), write and run "Hello world" in Python, C, and C++, then practice the material of this lecture
- A step-by-step guide will be available on Moodle; if possible, to gain time on Wednesday, start with:
  - Install Visual Studio Code, see `https://code.visualstudio.com/download`
  - Install Python, see `https://www.python.org/downloads/` (Windows and macOS; on Windows, check "Add python.exe to PATH" in the installer) or use your distribution's package manager (Linux)
  - Install a C/C++ compiler: MSYS2 on Windows, see `https://code.visualstudio.com/docs/cpp/config-mingw`; `xcode-select --install` in a terminal on macOS; `build-essential` or equivalent package on Linux

Licensing

Media used:

- Knuth's picture slide 4 is CC-BY-SA 2.0 by Alex Handy
- Von Neumann architecture slide 12 is CC-BY-SA 3.0 by Kapooht
- John Von Neumann's picture slide 12 is from Los Alamos National Laboratory, and made free to use

Bibliography I

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022. ISBN 978-0-262-04630-5. URL https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

K. N. King. *C Programming: A Modern Approach*. W.W. Norton & Company, 2 edition, 2008. ISBN 0393979504. URL http://knking.com/books/c2/index.html.

Mark Lutz. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, 6 edition, 2025. ISBN 1098171306. URL https://www.oreilly.com/library/view/learning-python-6th/9781098171292/.

Joakim Wassberg. *Computer Programming for Absolute Beginners: Learn Essential Programming Concepts, Terms, and Coding Techniques*. Packt Publishing, Limited, 1 edition, 2020. ISBN 978-1-83921-686-2. URL https://www.packtpub.com/en-dk/product/computer-programming-for-absolute-beginners-9781839216862.