CSSE1001 — Week 1 Notes

Table of contents

A Brief History of Computing **2**
- Outline . . . 2
- What is a computer? . . . 2
- What is software? . . . 3
- How do we describe something to a computer? . . . 3
- Algorithms, computer science, software engineering . . . 4
- Where this comes back . . . 4
- Next lecture . . . 5

Python as a Calculator **5**
- Outline . . . 5
- Learning objectives . . . 5
- The Python REPL . . . 5
- Syntax vs. semantics . . . 6
- Building the expression language . . . 6
- Order of operations . . . 8
- Associativity . . . 9
- Division, floats, exponents . . . 9
- Integer division . . . 10
- Worked exam questions . . . 11
- Variables . . . 11
- `min` and `max` . . . 12
- Exercise: second largest of three numbers . . . 12
- Next lecture . . . 12

Reference material **12**
- Algorithmic Complexity (Big-Oh) . . . 12
- History of Computing and Foundational Definitions . . . 18
- Programming Paradigms . . . 24
- Python Operator Precedence and Associativity . . . 27

Python Primitive Data Types 32
Python Recursion 43
Python While Loops 46
Sorting Algorithms 50

A Brief History of Computing

Lecture 1A, the first hour of week 1. Course logistics are in csse1001[1]. Lecture 1B, the same day, is 2025-07-29-python-as-a-calculator[2].

Six questions, grouped into four sections below. Five get a tentative definition, a "perhaps…", not a formal claim. The exception is *how do we describe something to a computer?*, which is answered with examples and code listings, not a definition. This hour is one of the two blocks csse1001[3] carves out of assessment (the other is all of week 13). It sets up the definitions that the end of the course returns to. See history-of-computing[4] for the full reference, including the hardware timeline, the Turing machine figure and the language progression.

Outline

- What is a computer?
- What is software?
- How do we describe something to a computer?
- What is an algorithm? What is computer science? What is software engineering?

What is a computer?

> A computer is a machine that can carry out sequences of arithmetic or logical operations automatically.

The word *machine* has not always applied: the Harvard "Computers" of the 1800s were people doing calculations by hand.

Formally, a **finite state machine** changes state when given input; a **Turing Machine** is a finite state machine with (infinite) read/write memory called tape, and is a *general purpose* computer — one machine that can be programmed to do different tasks. **Alan Turing** gave this definition, showed some problems are *not computable*, and helped break Enigma. **Charles Babbage** designed the first general purpose computer (the Analytical Engine; the schematics

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/2025-07-29-python-as-a-calculator.pdf
[3] courses/csse1001/csse1001.pdf
[4] courses/csse1001/history-of-computing.pdf

are dated 1840), never built in his lifetime — hardware following the design was assembled in 1910 as a four-operation calculator.

A concrete example of a finite state machine is a Turing machine that adds two binary numbers, with `carry` and `no carry` as its two working states. The formal theory belongs to Theory of Computing (COMP2048), so it is background here, not examinable material. The machine is drawn and read transition by transition in history-of-computing[5].

The hardware timeline runs Z3 (1941) → ENIAC (1945) → EDVAC (1949) → UNIVAC 1 (1951) → integrated circuit (1958) → UQ's GE-225 (1962) → Intel 4004 (1971) → 8086 (1978) → Pentium (1993) → today's Threadripper. UQ's own entry is the **GE-225, installed in the Hawken Building (now the Prentice Centre) in 1962**, weighing 900kg.

What is software?

> Software is the complement of hardware. They are instructions for the machine.

Ada Lovelace described an algorithm for the Analytical Engine to compute Bernoulli numbers, making her the first computer programmer. **Grace Hopper** pioneered machine-independent, English-word-based programming and invented COBOL.

The first computer bug is a moth taped to a 1947 logbook page. The entry, made at 15:45 on 9 September, reads:

> Relay #70 Panel F (moth) in relay.
>
> First actual case of bug being found.

How do we describe something to a computer?

First the Jacquard loom, punch cards and the 5MB punch-card program. Then a ladder of languages, every rung *the same* integer exponentiation program:

machine code → assembly (1947) → Fortran (1957) → COBOL (1959) → C (1972) → Python (1991)

The listings get shorter overall as they get higher-level, though not at every step: 19 lines of x86 machine code, 22 lines of assembly, 11 of Fortran, 19 of COBOL, 7 of C, 5 of Python. By the Python version the loop machinery has vanished entirely:

[5] courses/csse1001/history-of-computing.pdf

```python
def power(x: int, y: int) -> int:
    ans = 1
    for _ in range(y):
        ans = ans * x
    return ans
```

Python itself was written by **Guido van Rossum** and named after Monty Python — a general-purpose, high-level language meant to support several programming styles (see programming-paradigms[6]) and be fun to use.

Algorithms, computer science, software engineering

> An **algorithm** is an ordered sequence of unambiguous instructions that eventually terminates.
>
> The language we use to express ourselves is less important than what we are expressing: an instruction sequence that does something useful, like sorting a list.

The example for the second statement is a nine-row trace of selection sort on `[26, 54, 93, 17, 77, 31, 44, 55, 20]`, with no code beside it. The trace is drawn in sorting-algorithms[7], and selection sort comes back in week 13.

> **Computer science** is the practice/art of creating algorithms and measuring their relative effectiveness.
>
> **Software engineering** is the practice/art of creating and maintaining software (realized algorithms) in groups.

Where this comes back

"Creating algorithms and measuring their relative effectiveness" is the exact brief of the final lecture, 2025-10-28-complexity-and-sorting[8] — five ways to sort a list (sorting-algorithms[9]) and a formal way to compare them (algorithmic-complexity[10]). The course opens and closes on the same definition, and on the same selection-sort trace.

[6] courses/csse1001/programming-paradigms.pdf
[7] courses/csse1001/sorting-algorithms.pdf
[8] courses/csse1001/2025-10-28-complexity-and-sorting.pdf
[9] courses/csse1001/sorting-algorithms.pdf
[10] courses/csse1001/algorithmic-complexity.pdf

Next lecture

2025-07-29-python-as-a-calculator[11], Lecture 1B: arithmetic expressions and variables.

Python as a Calculator

See csse1001[12] for course logistics. This is Lecture 1B. The same day's first hour is 2025-07-29-a-brief-history-of-computing[13] (Lecture 1A).

Outline

- The Python REPL
- Syntax vs. semantics
- Arithmetic expressions, order of operations, and associativity
- Variables
- Worked exam questions

Learning objectives

1. Python expressions have syntax and semantics.
2. Python can evaluate arithmetic expressions, *but* this requires an order of operations and rules for associativity.

The Python REPL

A REPL (read-evaluate-print loop) is an interactive environment for a programmer to work in: it *reads* input, *evaluates* it, *prints* the result, then *loops*. A four-line transcript makes the acronym concrete, one letter per line:

The `>>>` prompt is only shown in the interactive REPL — it isn't included when writing Python instructions into files. Every transcript below is REPL output, so read `>>>` lines as what was typed and unprefixed lines as what Python printed back.

[11] courses/csse1001/2025-07-29-python-as-a-calculator.pdf
[12] courses/csse1001/csse1001.pdf
[13] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf

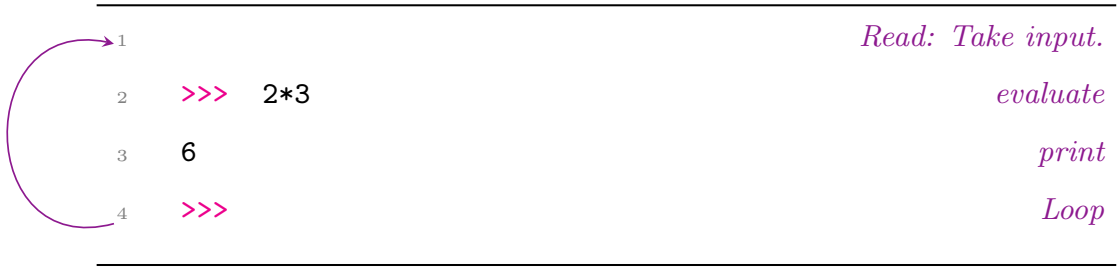


Figure 1: The four letters of read-evaluate-print-loop annotated onto the four lines of a transcript, with magenta `>>>` prompts and purple italic notes in the right margin. The return arrow is the loop. Only line 2 is typed by the user: line 3 is the printed result and line 4 is the prompt coming back.

Syntax vs. semantics

A programming language like Python has both:

- **Syntax** — defines what it means to be a *valid program*.
- **Semantics** — defines what a valid program *actually does*.

`2 + 3` is syntactically correct Python and evaluates to `5`. Prefix notation like `+ 2 3` (valid in Lisp) is *not* valid Python syntax:

```
>>> + 2 3
    + 2 3
      ^
SyntaxError: invalid syntax
```

Python's language rules must be followed *precisely* for a program to run. Each new *construct* below has the same two parts: its **syntax**, which says what counts as a well-formed expression, and its **semantics**, which says what it evaluates to. See python-operator-precedence-and-associativity[14] for all of them collected together, plus the order-of-operations table, associativity, integer division, and the `^` xor gotcha.

Building the expression language

Arithmetic is built up one construct at a time, each with its own transcript.

Constants and brackets. A constant is the most basic expression; if `x` is valid then so is `(x)`, and bracketed expressions get evaluated first.

[14] courses/csse1001/python-operator-precedence-and-associativity.pdf

```
>>> 2
2
>>> (2)
2
>>> (  2   )                # space is ignored
2
>>> ((2))
2
>>> 2)
    2)
     ^
SyntaxError: unmatched ')'
```

Negation and affirmation. If `x` is valid then `+x` and `-x` are valid; affirming is multiplying by positive one, negating is multiplying by negative one. Both chain freely, and neither can trail off the end of an expression.

```
>>> ++3
3
>>> +(+(3))
3
>>> 3+
    3+
      ^
SyntaxError: invalid syntax
>>> --3
3
>>> -+3
-3
>>> +-+-+-3
-3
```

Adding and subtracting. As a matter of style, bind the sign to the number it belongs to rather than leaving it floating next to the operator:

```
>>> 3 + - 3
0
>>> 3 + -3                  # best to group like this
0
>>> + 1 + + + 3
4
```

```
>>> +1 + ++3                 # best to group like this
4
>>> 2 - - 2
4
>>> 2 - -2                   # best to group like this
4
```

Multiplication, which is where evaluation order starts to matter:

```
>>> 2 + 3 * 2 - 1           # multiplication was done first
7
>>> (2 + 3) * 2 - 1         # brackets change outcome
9
>>> 1 + 2 * 3 + 4
11
```

That last line is the argument for needing a rule at all: since `1 + 2 * 3 + 4` is `11` and not `13`, `*` must be evaluated before `+`. Without an order of operations an expression like this is ambiguous.

Order of operations

For what has been introduced so far, the order is parentheses, unary `-x`/`+x`, `*`, then `+`/`-`. Once division, integer division, remainder and exponentiation are added, the full table becomes:

Operator	Description
`()`	Parenthesis
`**`	Exponents
`-x`, `+x`	Negation, Affirmation
`*`, `/`, `//`, `%`	Multiplication, Division, Integer Division, Remainder
`+`, `-`	Addition, Subtraction

Highest precedence at the top. Worked examples:

```
>>> 10 - 2 * 3              # multiplication before subtraction
4
>>> 2 * - 3                 # negation before multiplication
-6
```

```
>>> 2 * -3                  # best to group according to precedence
-6
>>> 1 - - 2 * - 3 - 4
-9
>>> 1 - -2 * -3 - 4         # best to group according to precedence
-9
```

Associativity

Precedence alone is not enough, because some operators *share* a level:

```
>>> 1 - 2 + 3
2
>>> 1 - (2 + 3)
-4
>>> (1 - 2) + 3
2
```

Since Python's answer matches the left bracketing, `+` and `-` are **left-associative**. The rule: operators of the same precedence are left associative, *excepting exponentiation which is right associative* — so if `#` and `@` have equal precedence, `a # b @ c = (a # b) @ c`.

```
>>> 3 * 1 // 2
1
>>> 3 * (1 // 2)
0
>>> 2 ** 1 ** 0
2
>>> (2 ** 1) ** 0
1
```

The bracketing Python uses is `(3 * 1) // 2` for the first pair and `2 ** (1 ** 0)` for the second.

Division, floats, exponents

```
>>> 4 / 2                   # notice the decimal
2.0
>>> type(2)
<class 'int'>
```

```
>>> type(2.0)
<class 'float'>
>>> 6 / 2 * (1 + 2)
9.0
>>> 1 / 3                    # notice this is an approximate solution
0.3333333333333333
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 1 / float('inf')
0.0
>>> 2 ** 3.0
8.0
>>> 2 ** -1
0.5
>>> 0 ** 0
1
```

The `int`/`float` distinction is picked up properly next week in python-primitive-data-types[15].

Warning. The caret `^` is bit-wise xor, *not* exponentiation, and using it wrongly raises no error at all — `2 ^ 3` is `1` and `4 ^ 1` is `5`.

Integer division

How can we divide two integers so that we get an integer?

Using "grade school" division: for `x = 17` and `y = 3` we say *17 ÷ 3 is 5 remainder 2*, or equivalently $17 = 5 \cdot 3 + 2$, or $\frac{17}{3} = 5 + \frac{2}{3}$.

```
>>> 17 // 3                  # quotient
5
>>> 17 % 3                   # remainder
2
>>> 3 * (17 // 3) + (17 % 3)
17
```

Division algorithm. For positive integers x and y there is a *unique* quotient q and a *unique* remainder r with $0 \leq r < y$ satisfying $x = q \cdot y + r$. The remainder can be computed two ways, both using machinery that comes later:

[15] courses/csse1001/python-primitive-data-types.pdf

```
>>> def remainder(x: int, y: int) -> int:
...     ans = x
...     while ans >= y:
...         ans = ans - y
...     return ans
>>> remainder(17, 7)
3
```

```
>>> def remainder(x: int, y: int) -> int:
...     return x if x < y else remainder(x - y, y)
>>> remainder(17, 7)
3
```

The first is python-while-loops[16] (week 4); the second is python-recursion[17] (week 12).

Worked exam questions

Q: What does `2 ** 3 % 5 - 1` evaluate to? (Options: 0, 2, 4, 7.)

`((2 ** 3) % 5) - 1 = (8 % 5) - 1 = 3 - 1 = 2`

Q: What value gets assigned to `x` in `x = 35 * 2 % 5 ** 2`? (Options: 0, 20, 70, 140, none of the above.)

`(35 * 2) % (5 ** 2) = 70 % 25 = 20`. python-operator-precedence-and-associativity[18] draws out the two steps that produce that bracketing.

Variables

Values can be named — these names are called **variables**. Assigning a value to a variable doesn't print anything in the REPL:

```
>>> width = 2
>>> height = 3
>>> area = width * height
>>> 4 * area
24
```

[16] courses/csse1001/python-while-loops.pdf
[17] courses/csse1001/python-recursion.pdf
[18] courses/csse1001/python-operator-precedence-and-associativity.pdf

`min` and `max`

```
>>> max(1, 2)
2
>>> min(1, 2)
1
>>> max(5, 1, -2, 3, 2*7)
14
```

In the last one the arguments are themselves expressions, evaluated before `max` ever sees them.

Exercise: second largest of three numbers

Given three distinct numbers `a`, `b`, `c`, write an expression for the *second largest*. Three answers, set up with a multiple assignment:

```
>>> a, b, c = -9, 19, 3      # assignment like this saves vertical space
>>> a + b + c - max(a, b, c) - min(a, b, c)
3
>>> min( max(a, b), max(a, c), max(b, c) )
3
>>> max( min(a, b), min(a, c), min(b, c) )
3
```

Next lecture

1. 2025-08-04-python-memory-model[19] — Python memory model.
2. 2025-08-04-primitive-data[20] — Primitive data types.

Reference material

Algorithmic Complexity (Big-Oh)

An **algorithm** is an ordered sequence of unambiguous instructions that eventually terminates (see history-of-computing[21]). Once we have more than one algorithm for the same problem, we need a way to say which is better — that measure is **complexity**.

[19] courses/csse1001/2025-08-04-python-memory-model.pdf
[20] courses/csse1001/2025-08-04-primitive-data.pdf
[21] courses/csse1001/history-of-computing.pdf

This is not a side topic bolted on at the end of semester. The entire subject is defined in one sentence:

> **Computer science** is the practice/art of creating *algorithms* and *measuring their relative effectiveness*.

First met in lecture 1A[22]. sorting-algorithms[23] is the first half of that sentence; complexity is the second half. The course opens on the definition in week 1 and closes on it in week 13.

Why not just time it?

Timing depends on the machine, the language, what else is running, and the particular input. Instead we *count operations*. For a sorting algorithm the operations worth counting are **comparisons** and **swaps**; more generally we count whatever the algorithm does repeatedly.

The machine matters a great deal. The hardware timeline runs from Konrad Zuse's Z3 to a current AMD part, and eight of its machines have a quoted speed. Laid on a single logarithmic axis, those speeds are the argument against the stopwatch.

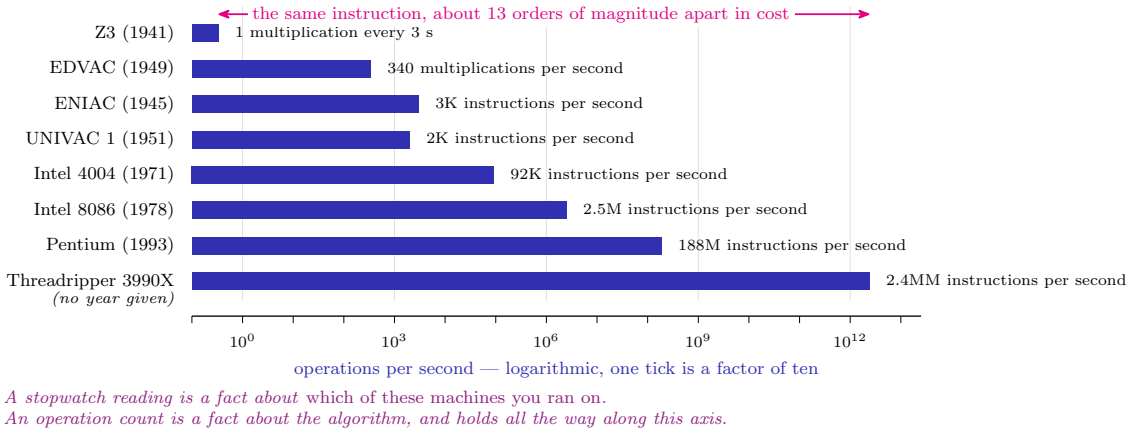


Figure 2: The quoted speeds of the machines in the hardware timeline, on one logarithmic axis. Each bar is one machine. The **units are not comparable** - Z3 and EDVAC are quoted in *multiplications* per second and the rest in *instructions* per second. The rows are not in year order: EDVAC (1949) sits above ENIAC (1945), and the Threadripper has no year. The last figure, `2.4MM`, is placed at 2.4×10^{12}, reading MM as million-million. The magenta span runs from the slowest machine to the fastest. The full table with prices and manufacturers is in history-of-computing[24].

[22] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf
[23] courses/csse1001/sorting-algorithms.pdf
[24] courses/csse1001/history-of-computing.pdf

Nothing about selection sort changed between 1941 and today. The number of comparisons it makes on a nine-element list is the same on the Z3 as on the Threadripper; only the wall-clock time moves, and it moves by a factor of about 10^{13}. **That invariant count is what complexity measures.**

Definition

> We measure the complexity of an algorithm by counting the number of operations performed. But the number of operations is proportional to the length of the input (usually denoted n) — and it is really this *proportion* we are interested in, rather than a precise accounting of operations.

That last sentence is the whole idea: `2n + 2` and `5n + 100` are both "linear", and the difference between them matters far less than the difference between linear and quadratic.

Big-Oh notation

Definition. We say "$f(n)$ is big-oh of $g(n)$", written $f(n) = O(g(n))$, when

$$\exists\, c, k \in \mathbb{R}_{>0} : \forall n \geq k,\ 0 \leq f(n) \leq c\, g(n).$$

That is, when there is some positive constant c such that $c \cdot g$ is *eventually* larger than f. The direction matters: "f is eventually larger than cg" is backwards and contradicts the inequality.

Worked examples:

1. $2x^2 + x + 1 = O(n^2)$, because for $x > 2$ we have $2x^2 + x + 1 < 3x^2$ (take $c = 3$, $k = 2$).
2. $x^3 - \sin(x) = O(n^3)$, because for $x > 1$ we have $x^3 - \sin(x) \leq x^3 + 1 < 2x^3$ (take $c = 2$, $k = 1$).

In practice this means: **drop the constants and keep only the fastest-growing term.**

The four complexity classes in this course

Linear — $O(n)$

```
def linear(xs: list[int]):
    n = len(xs)         # 1  assignment
    index = 0           # 1  assignment
    while index < n:    # n  comparisons
        index += 1      # n  assignments / additions
                        # 2n + 2 = O(n) operations
```

Quadratic — $O(n^2)$

A loop nested inside a loop, each running n times:

```
def foo(xs: list[int]):
    n = len(xs)         # 1
    i = 0               # 1
    while i < n:        # n
        j = 0           # n
        while j < n:    # n * n
            j += 1      # n * n
        i += 1          # n
                        # 2n**2 + 3n + 1 = O(n**2)
```

Constant — $O(1)$

The trap here is that a big number is still a *constant*: the loop below runs ten billion times, but that count does not depend on `n` at all.

```
def foo(xs: list[int]):
    n = len(xs)           # 1
    index = 0             # 1
    while i < 10**10:     # 10**10
        index += 1        # 10**10
                          # 2*10**10 + 2 = O(1)
```

Logarithmic — $O(\log n)$

The loop variable is in an *exponent*, so the loop runs about $\log_2 n$ times:

```
def log(xs: list[int]):
    n = len(xs)                 # 1
    index = 0                   # 1
    while 2 ** i < len(xs):     # log[2](n)
        i += 1                  # log[2](n)
                                # 2log[2](n) + 2 = O(log(n))
```

Both this listing and the constant one above mix their counter names: each initialises `index` and then increments `i`. Neither would run; the operation counts in the comments are still the point.

The base of the logarithm is a constant factor ($\log_2 n = \log n / \log 2$), so it disappears inside the big-oh — $O(\log_2 n)$ and $O(\log n)$ mean the same thing.

How to read a loop's complexity

- A loop that advances by a *fixed step* toward `n` $\rightarrow O(n)$.
- A loop that *multiplies or divides* its counter each pass $\rightarrow O(\log n)$.
- Nested loops $\rightarrow$ multiply their counts ($O(n) \times O(n) = O(n^2)$).
- A bound that doesn't mention `n` $\rightarrow O(1)$, however large it is.
- A bound that is an input **value** rather than an input **length** $\rightarrow$ see the next section, and be careful.

When the input is a value, not a length

Every example above takes `n = len(xs)`: the input is a collection and n counts its elements. One of the course's earliest algorithms, integer division from Lecture 1B[25], does not fit that mould.

Dividing two integers and still getting an integer is grade-school division: for $x = 17$ and $y = 3$, *17 ÷ 3 is 5 remainder 2*, i.e.

$$17 = 5 \cdot 3 + 2 \qquad \text{or} \qquad \frac{17}{3} = 5 + \frac{2}{3}.$$

Python hands you both halves directly, and the two together reconstruct the input:

```
>>> 17 // 3
5
>>> 17 % 3
2
>>> 3 * (17 // 3) + (17 % 3)
17
```

`17 // 3` is the *quotient* and `17 % 3` is the *remainder*. The theorem the two operators rest on:

[25] courses/csse1001/2025-07-29-python-as-a-calculator.pdf

Division algorithm. Let x and y be positive integers. There is a **unique** quotient q and **unique** remainder r with $0 \leq r < y$ satisfying $x = q \cdot y + r$.

The remainder can also be computed by hand, subtracting `y` away until nothing more can be taken (`while` is not taught until week 4):

```
>>> def remainder(x: int, y: int) -> int:
...     ans = x
...     while ans >= y:
...         ans = ans - y
...     return ans
>>> remainder(17, 7)
3
```

The recursive version (recursion is week 12, python-recursion[26]):

```
>>> def remainder(x: int, y: int) -> int:
...     return x if x < y else remainder(x-y, y)
>>> remainder(17, 7)
3
```

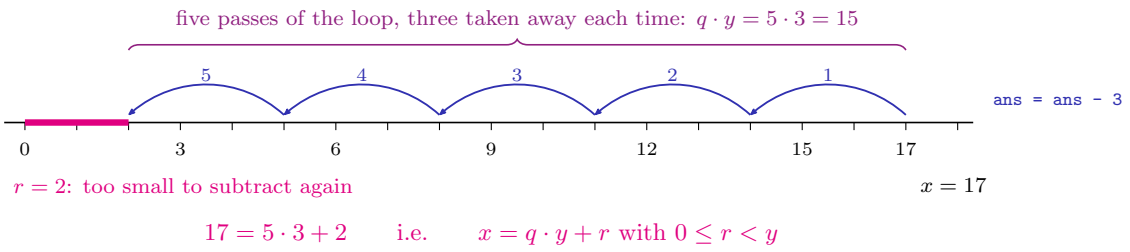


Figure 3: Repeated subtraction for $17 \div 3$. The number line, the arcs and the count of passes show what the `remainder` loop does; the equation beneath it is $17 = 5 \cdot 3 + 2$, with the general form $x = q \cdot y + r$ and $0 \leq r < y$. The code above runs on $(17, 7)$ and the arithmetic on $(17, 3)$; the figure follows the arithmetic.

Now count operations. The loop body runs once per subtraction, and the number of subtractions is exactly $\lfloor x/y \rfloor$ — the quotient. So `remainder(x, y)` costs on the order of x/y operations, and `remainder(10**9, 1)` would run a billion times while `remainder(10, 1)` runs ten. **The cost is set by the magnitude of the input, not by its length.**

[26] courses/csse1001/python-recursion.pdf

That is why complexity is always stated against the *size* of the input rather than its value. It is also why `//` and `%` are worth having: they reach the same two answers without the loop, so the number of subtractions never appears in the running time at all. Whenever you meet a loop whose bound is a number rather than a `len`, check which of the two kinds you are looking at before naming a complexity class.

Complexity of the sorting algorithms

Algorithm	Complexity
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Bubble sort	$O(n^2)$
Merge sort	$O(n \ln n)$

See sorting-algorithms[27] for how each of these works. Merge sort is written here as $O(n \ln n)$ — the same class as $O(n \log n)$, since the base is a constant factor.

Why this matters elsewhere

Complexity is what makes dynamic programming[28] worth doing: a branching recursion like the naive Fibonacci recomputes the same subproblems exponentially many times, and caching collapses that back down to linear. It is also the justification for the dictionary's "magic" hash function in python-dictionaries[29] — a hash table finds a key in roughly constant time where scanning a list would take linear time.

History of Computing and Foundational Definitions

The course's opening definitions — what a computer, software, an algorithm, computer science, and software engineering actually *are* — come with their history. Each is a tentative "perhaps…", not a formal claim. First met in 2025-07-29-a-brief-history-of-computing[30].

[27] courses/csse1001/sorting-algorithms.pdf
[28] courses/csse1001/python-recursion.pdf
[29] courses/csse1001/python-dictionaries.pdf
[30] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf

What is a computer?

> A computer is a machine that can carry out sequences of arithmetic or logical operations automatically.

Before it was a machine it was a job title: the **Harvard "Computers"** of the 1800s were people employed to perform calculations by hand. A photograph of the group shows only women.

The formal definition

A **finite state machine** is a machine that can change state when given input. A **Turing Machine** is a finite state machine that additionally has (infinite) read/write memory called *tape*.

A Turing Machine is a **general purpose computer** — a single machine that can be programmed to do different tasks, rather than a machine built to do one thing. (Pursued properly in COMP2048, Theory of Computing.)

Alan Turing (1912–1954) gave this formal definition of a computer and of what it means for something to be *computable* — and in particular demonstrated that there are problems which are **not** computable. He also played a critical role in breaking the Enigma machine.

A Turing machine that adds two binary numbers makes "changes state when given input" concrete.

The labels read as arithmetic. From $\bar{c}$, no carry is pending, so $1 + 1$ writes 0 and moves to c. From c a carry is pending, so $1 + 1$ writes 1 and stays in c, while $0 + 0$ writes 1 and drops back to $\bar{c}$. Run off the end of both numbers with a carry still pending and the machine writes the final 1; run off the end without one and it halts leaving the tape alone. It is school-book column addition with the carry held *in the state* instead of in memory. Two states suffice because a carry is one bit. That is what makes it a usable illustration of "changes state when given input".

Charles Babbage (1791–1871), an English mathematician, drew the schematics (1840) for the first general purpose computer — the **Analytical Engine**. It was never built in his lifetime; hardware following the design was assembled in 1910 as a calculator for addition, subtraction, multiplication and division.

The hardware timeline

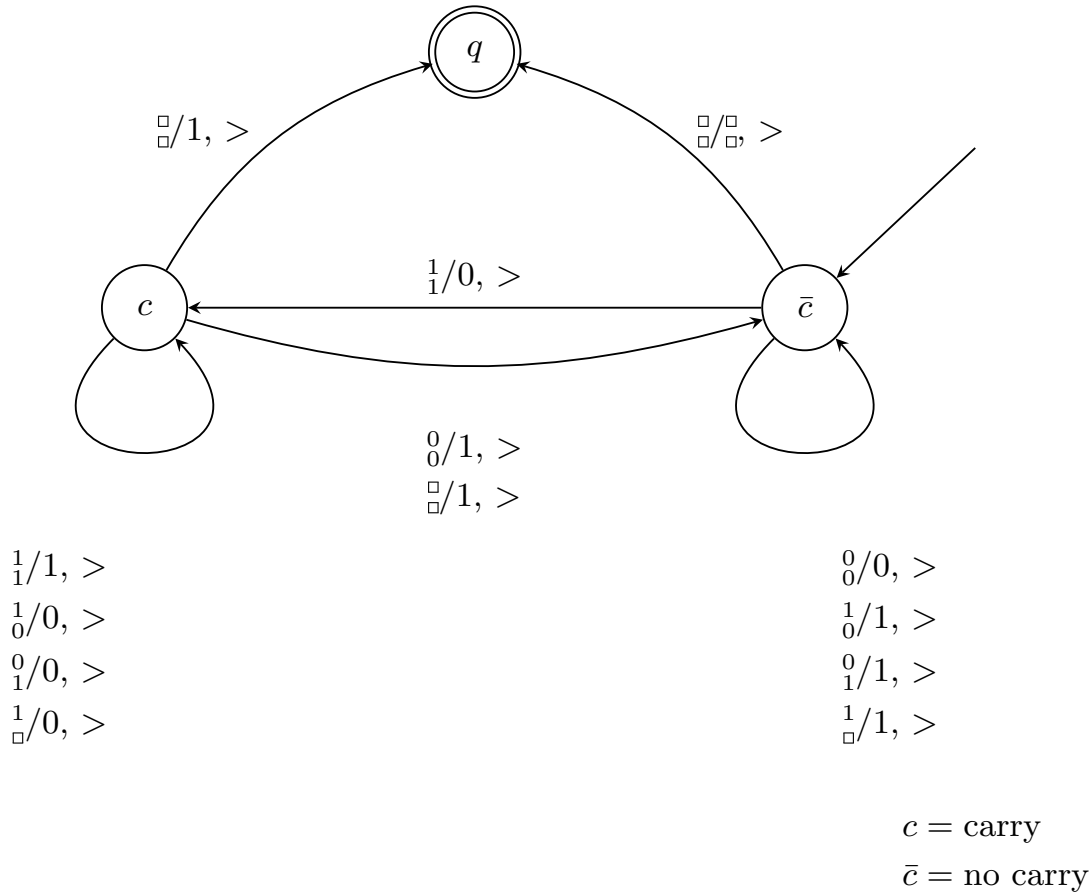


Figure 4: A Turing machine for binary addition. Three states: q, c = carry and $\bar{c}$ = no carry. Every transition is read as *what is on the tape / what gets written*, *which way the head moves*: the stacked pair is one bit from each of the two numbers being added, the empty square is a blank cell, and `>` is a step to the right. The label format, the incoming arrow as the start state, and the double circle on q as a halting state follow the standard finite-automaton conventions. The transition on a pair of blanks writing 1 appears twice, on both $c \to q$ and $c \to \bar{c}$. The topic belongs to COMP2048.

Year	Machine	Notes
1941	**Z3**	First general purpose computer. Germany, Konrad Zuse. 1 multiplication every 3 seconds.
1945	**ENIAC**	Electronic Numerical Integrator and Computer. U. Pennsylvania. $7.3M. 3K instructions/sec.
1949	**EDVAC**	Electronic Discrete Variable Automatic Computer. Ballistic Research Laboratory. $1.4M. 340 multiplications/sec.
1951	**UNIVAC 1**	Universal Automatic Computer. Designed for business and admin. $1.8–14M. 2K instructions/sec.
1958	**First integrated circuit**	Jack Kilby (Nobel Prize in Physics) — a collection of electronic components fabricated onto silicon.
1962	**UQ's GE-225**	Installed in the Hawken Building (now the Prentice Centre). Weighed 900kg.
1971	**Intel 4004**	Large scale integration. First commercially produced microprocessor. 92K instructions/sec.
1978	**Intel 8086**	Very large scale integration. 2.5M instructions/sec.
1993	**Pentium**	Intel. $1.7K. 188M instructions/sec.
—	**AMD Ryzen Threadripper 3990x**	Ultra large scale integration. $4K. 2.4MM instructions/sec.

The through-line is **integration**: a collection of components fabricated onto silicon (1958) → large scale (1971) → very large scale (1978) → ultra large scale (Threadripper).

The speed figures are not in comparable units. Z3 and EDVAC are quoted in *multiplications* per second, everything else in *instructions* per second. This is a trend, not a benchmark.

What is software?

> Software is the complement of hardware. They are instructions for the machine.

Ada Lovelace (1815–1852), English mathematician and only child of the poet Lord Byron, described an algorithm for the Analytical Engine to compute Bernoulli numbers — she is considered the first computer programmer.

Grace Hopper (1906–1992), a United States Navy rear admiral, pioneered *machine independent* programming based on English words and invented COBOL. She has both a guided-missile destroyer and a supercomputer named after her.

The "first computer bug" is a moth taped into a 1947 logbook page. The entry, at 15:45 on 9 September, reads:

> Relay #70 Panel F (moth) in relay.
>
> First actual case of bug being found.

The page's other entries — *Started Cosine Tape (Sine check)*, *Started Mult+Adder Test* — are the ordinary shift log around it. Debugging is routine work, and this is the one entry where the bug was literally an insect.

How do we describe something to a computer?

The earliest answer predates computers entirely: the **Jacquard loom** used punched cards to control weaving patterns, and **punch cards** carried programs into machines well into the era of electronic computing. The largest punch card program was 5MB: a shoulder-high bank of loose card stacks.

Above the cards, the same program — integer exponentiation, `power(x, y)` — walks up through the generations of language:

- **Machine code** (x86 32-bit): raw bytes, `55` / `89 E5` / `B8 01 00 00 00`, each one an instruction the CPU decodes directly. The two jumps are a *relative jump 9 bytes forward if zero* and a *relative jump back 11 bytes*.
- **Assembly (1947)**: the same instructions given mnemonic names and labels — `push ebp`, `imul eax, ebx`, `jmp .loop`. One line per machine instruction, but readable.
- **Fortran (1957)**: `DO 10 I = 1, Y` … `10 CONTINUE`. Numbered statements, but arithmetic looks like arithmetic.
- **COBOL (1959)**: `IDENTIFICATION DIVISION.` … `PERFORM UNTIL COUNTER = 0`. Deliberately English-like, designed for business and administration.
- **C (1972)**: `for (int i = 0; i < y; i++) { ans = ans * x; }` — structured control flow, close to the machine but portable.

- **Python (1991)**: `for _ in range(y): ans = ans * x` — the loop mechanics disappear entirely.

Each step trades control over the machine for expressiveness to the human. The line count shows it: 19 lines of machine code and 22 of assembly become 5 lines of Python for the same function.

Python

Python was developed by **Guido van Rossum** and named after the British comedy troupe **Monty Python** (not the snake). It is a general-purpose, high-level programming language, created to support a variety of programming styles (see programming-paradigms[31]) and to be fun to use.

What is an algorithm?

> An algorithm is an ordered sequence of unambiguous instructions that eventually terminates.

The three requirements are each load-bearing: *ordered* (see the sequencing examples in python-variables-and-memory-model[32]), *unambiguous*, and *eventually terminates* — which is exactly what an infinite `while True` loop fails to do.

The language we use to express an algorithm matters far less than what we are expressing: an instruction sequence that does something useful, like sorting a list. That is why the same exponentiation program above is recognisably the same algorithm in six different languages.

Selection sort on `[26, 54, 93, 17, 77, 31, 44, 55, 20]` is the example of "something useful". A nine-row trace shows it with no code, because the code is the part that does not matter yet. The trace is drawn in sorting-algorithms[33], and selection sort returns in week 13.

What is computer science? What is software engineering?

> **Computer science** is the practice/art of creating algorithms and measuring their relative effectiveness.
>
> **Software engineering** is the practice/art of creating and maintaining software (realized algorithms) in groups.

[31] courses/csse1001/programming-paradigms.pdf
[32] courses/csse1001/python-variables-and-memory-model.pdf
[33] courses/csse1001/sorting-algorithms.pdf

The first half of that computer science definition is sorting-algorithms[34]; the second half is algorithmic-complexity[35]. The end of the course does both at once, as a transition from software engineering to computer science.

Programming Paradigms

A **paradigm** is the style in which you describe a computation to a machine. The top-level split is between saying *how* to do something and saying *what* you want.

	Question answered
Imperative	How to do it
Declarative	What to do

Imperative programming

The programmer spells out *how* to do it, step by step. (Imperative, as an adjective, means giving an authoritative command.) It divides further by how those commands are grouped:

1. **Procedural** — groups instructions into *functions*. This is everything up to week 7: python-functions[36], python-if-statements[37], python-while-loops[38], python-for-loops[39].
2. **Object-oriented** — groups instructions into *objects* that combine data (state, attributes) and behaviour (methods). This is weeks 7–11: python-classes-and-objects[40] onward.

Both are imperative — OOP does not stop you writing step-by-step instructions, it just changes what those instructions are attached to.

Declarative programming

The programmer declares what they want, through:

1. **Functional** — a series of function applications.
2. **Logic** — a question asked about a system of facts and rules.
3. **Mathematical optimization** — an objective and a set of constraints.

[34] courses/csse1001/sorting-algorithms.pdf
[35] courses/csse1001/algorithmic-complexity.pdf
[36] courses/csse1001/python-functions.pdf
[37] courses/csse1001/python-if-statements.pdf
[38] courses/csse1001/python-while-loops.pdf
[39] courses/csse1001/python-for-loops.pdf
[40] courses/csse1001/python-classes-and-objects.pdf

Python is not a declarative language, but it has declarative *moments*. A list comprehension is the clearest one:

```
# imperative: how to build the list
acc = []
for k in range(4):
    acc.append(k**2)

# declarative: what the list is
[k**2 for k in range(4)]
```

The comprehension reads like the set-builder notation $\{k^2 : k \in \{0, 1, 2, 3\}\}$ — you state the set, not the procedure for accumulating it. See python-comprehensions[41].

Paradigm is not the same as level of abstraction

Take **one program written six times over** — integer exponentiation, `power(x, y)` — in six languages, from raw x86 bytes up to Python (the listings themselves are walked through in history-of-computing[42]). It is tempting to read that ladder as a tour of paradigms. It is not. Every rung is **imperative**: set an accumulator to 1, run a loop `y` times, multiply.

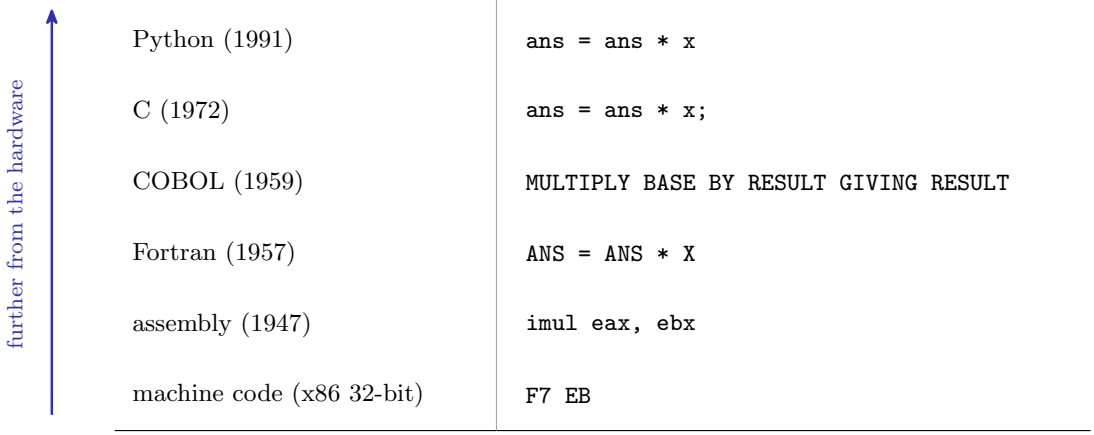
The loop is the clearest tell. All six spell out *how* to repeat:

Language	How it repeats
machine code	`74 09` and `EB F5` — a conditional jump 9 bytes forward, an unconditional jump 11 bytes back
assembly	`test ecx, ecx` / `jz .done` … `jmp .loop`
Fortran	`DO 10 I = 1, Y` … `10 CONTINUE`
COBOL	`PERFORM UNTIL COUNTER = 0` … `END-PERFORM.`
C	`for (int i = 0; i < y; i++)`
Python	`for _ in range(y)`

COBOL looks least like the rest. `MULTIPLY BASE BY RESULT GIVING RESULT` is English, and that was the point: Grace Hopper's contribution was *machine independent programming based on english words*. But English-like syntax is not a declarative paradigm — the COBOL program still names every step and every repetition, in order. It is imperative prose.

So the ladder measures a different axis from the imperative/declarative split:

[41] courses/csse1001/python-comprehensions.pdf
[42] courses/csse1001/history-of-computing.pdf



language (year)	the multiplication step, as that language writes it
Python (1991)	`ans = ans * x`
C (1972)	`ans = ans * x;`
COBOL (1959)	`MULTIPLY BASE BY RESULT GIVING RESULT`
Fortran (1957)	`ANS = ANS * X`
assembly (1947)	`imul eax, ebx`
machine code (x86 32-bit)	`F7 EB`

Every rung says how*: set an accumulator to 1, run the body y times, multiply.*
Only the distance from the hardware changes — the paradigm does not.

Figure 5: The one step that survives all six languages. Each row is that language's way of writing *multiply the accumulator by x*. The rows run from the oldest language at the bottom to the newest at the top. The bottom two rows are the same instruction twice: `F7 EB` is what the machine stores, `imul eax, ebx` is the assembly for it.

- **Level of abstraction** — how much of the machine you have to say out loud. This is what changes from `F7 EB` to `ans = ans * x`.
- **Paradigm** — whether you say *how* or *what*. This does not change anywhere on the ladder.

Python's `**` operator (see 2025-07-29-python-as-a-calculator[43]) needs no loop written at all, and is one more step up the *first* axis: `x ** y` hands the repetition to the language. That still is not a change of paradigm — a single operator call is an abstraction, not a style of description. The genuine paradigm shift is the one shown further up, where a comprehension replaces the accumulation with a statement of what the result *is*. For `**` alongside the rest of the arithmetic, see python-operator-precedence-and-associativity[44].

Why Python has several

Python was created to *support a variety of programming styles* rather than enforce one (see history-of-computing[45]). CSSE1001 uses this deliberately: the course is structured as procedural first, then object-oriented, then (in the final, non-assessed week) a look at the functional style.

Where OOP fits

> The fundamental building block of object-oriented programming is the class or object. In this paradigm, the design principle is to solve a problem by creating objects that interact with one another.

An **object** is a collection of fields and attributes (the data comprising the object's state) along with methods (class-scoped functions) that can act on the object itself. An object can reference and change its own state, and has a notion of `self`.

Continue in python-classes-and-objects[46] for the mechanics, and 2025-09-09-object-oriented-programming[47] for the lecture that introduces them.

Python Operator Precedence and Associativity

Syntax vs. semantics

Each *construct* below has two parts, and it is worth keeping them apart:

[43] courses/csse1001/2025-07-29-python-as-a-calculator.pdf
[44] courses/csse1001/python-operator-precedence-and-associativity.pdf
[45] courses/csse1001/history-of-computing.pdf
[46] courses/csse1001/python-classes-and-objects.pdf
[47] courses/csse1001/2025-09-09-object-oriented-programming.pdf

- **Syntax** — what makes the expression a *valid* Python program.
- **Semantics** — what the expression *evaluates to*.

A syntax rule is always of the form "if `x` and `y` are valid Python expressions, then *this* is a valid Python expression" — which is what makes arbitrarily long arithmetic legal without a separate rule for every length. Precedence and associativity do not take this shape; they are plain rules.

Constants and brackets

- A constant (e.g. `2`) is the most basic expression.
- If `x` is a valid expression then `(x)` is also a valid expression — bracketed expressions get evaluated first. Whitespace inside brackets is ignored, and brackets can nest arbitrarily (`((2))`).
- An unmatched closing bracket (`2)`) is a `SyntaxError`.

Negation and affirmation

- If `x` is a valid expression then `+x` and `-x` are valid expressions.
- `+x` is equivalent to multiplying `x` by `1`; `-x` is equivalent to multiplying `x` by `-1`.
- These can be chained (`--3` is `3`, `+-+-+-3` is `-3`), but a trailing operator with nothing after it (`3+`) is a `SyntaxError`.
- Style: write `3 + -3` rather than `3 + - 3`, and `2 - -2` rather than `2 - - 2` — group the sign with the number it belongs to.

Arithmetic operators

If `x` and `y` are valid expressions, then all of the following are valid expressions, evaluated according to the rules of math:

Operator	Meaning
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Integer (floor) division
`x % y`	Remainder
`x ** y`	Exponentiation

Order of operations

`1 + 2 * 3 + 4` evaluates to `11`, not `13`, so `*` must happen before `+`. **Without an order of operations that expression is ambiguous.**

From highest to lowest precedence:

Operator	Description
`()`	Parenthesis
`**`	Exponents
`-x`, `+x`	Negation, Affirmation
`*`, `/`, `//`, `%`	Multiplication, Division, Integer Division, Remainder
`+`, `-`	Addition, Subtraction

Two consequences of the middle row that catch people out: exponentiation outranks unary minus, and unary minus outranks multiplication. So `2 * -3` is `2 * (-3)` = `-6`, and `1 - -2 * -3 - 4` is `1 - ((-2) * (-3)) - 4` = `-9`.

`*`, `/`, `//`, and `%` share a precedence level, as do `+` and `-` — so the table by itself still leaves ties. See Associativity below.

Associativity

An order of operations alone doesn't remove all ambiguity — e.g. `1 - 2 + 3` needs a rule for whether it means `(1 - 2) + 3` or `1 - (2 + 3)`. Python evaluates `1 - 2 + 3` as `2`, matching `(1 - 2) + 3` rather than `1 - (2 + 3)` = `-4`, so `+`/`-` are **left-associative**.

Operators of equal precedence are left-associative in general, **except exponentiation (`**`), which is right-associative**:

- `3 * 1 // 2` = `(3 * 1) // 2` = `1`, whereas `3 * (1 // 2)` = `0` (left-associative `*`/`//`)
- `2 ** 1 ** 0` = `2 ** (1 ** 0)` = `2`, whereas `(2 ** 1) ** 0` = `1` (right-associative `**`)

In general, if `#` and `@` are operators of equal precedence, `a # b @ c` = `(a # b) @ c` (left-associative case).

```
x = 35 * 2 % 5 ** 2
```
*** has the highest precedence, so it is bracketed first*

```
x = 35 * 2 % (5 ** 2)
```
** and % are equal precedence and left-associative, so the left-hand pair is bracketed first*

```
x = (35 * 2) % (5 ** 2)

x = 70 % 25

x = 20
```

Figure 6: Fully bracketing `x = 35 * 2 % 5 ** 2`, one rule at a time, to reach `(35 * 2) % (5 ** 2) = 70 % 25 = 20`. Each step boxes the tokens it brackets and names the rule in purple on the right; magenta marks the answer. Step one is precedence: `**` outranks everything else present. Step two is associativity: `*` and `%` sit on the same row of the table, so the tie is broken left-first. After that no choices remain.

Putting both rules on one expression

The exam question `x = 35 * 2 % 5 ** 2` needs precedence *and* associativity to settle, which is why it is the one worth working slowly.

The other exam question resolves by precedence alone: `2 ** 3 % 5 - 1` is `((2 ** 3) % 5) - 1` = `(8 % 5) - 1` = `3 - 1` = 2.

The tempting wrong answers to `35 * 2 % 5 ** 2` (`0`, `140` and `70`) are a good check on the table. Evaluating strictly left to right, ignoring precedence altogether, gives `((35 * 2) % 5) ** 2` = `0 ** 2` = `0`. Letting `%` bind first, then `**`, and leaving `*` until last gives `35 * ((2 % 5) ** 2)` = `35 * 4` = `140`. And `70` is just `35 * 2` with the rest of the expression dropped.

Division and floats

- `/` always returns a `float`, even when the division is exact (`4 / 2` is `2.0`, not `2`) — `type(2)` is `int`, `type(2.0)` is `float`.
- `1 / 3` gives an *approximate* result (`0.3333333333333333`) since floats have finite precision.
- `1 / 0` raises `ZeroDivisionError`; `1 / float('inf')` is `0.0`.

- `6 / 2 * (1 + 2)` is `9.0` — brackets first, then `/` and `*` left to right, and the `/` makes the whole result a float.

Exponentiation edge cases

- `2 ** 3` is `8` (int); `2 ** 3.0` is `8.0` (float) — a float exponent/base produces a float result.
- `2 ** -1` is `0.5`.
- `2 ** 0` and `0 ** 0` are both `1`.

Warning: `^` is not exponentiation

The caret `^` is Python's **bit-wise xor** operator, not exponentiation — e.g. `2 ^ 3` is `1` and `4 ^ 1` is `5`. Using `^` where you meant `**` does *not* raise an error, so this mistake can silently produce wrong results.

Integer division (the division algorithm)

For positive integers `x` and `y`, there is a unique quotient `q` and remainder `r` (with `0 <= r < y`) satisfying `x = q * y + r` — "grade school" division. E.g. for `x = 17`, `y = 3`: `17 = 5 * 3 + 2`, so `17 // 3` is `5` (the quotient) and `17 % 3` is `2` (the remainder), and `3 * (17 // 3) + (17 % 3) == 17`.

The division algorithm can be computed directly (repeated subtraction) or recursively:

```
def remainder(x: int, y: int) -> int:
    ans = x
    while ans >= y:
        ans = ans - y
    return ans
```

```
def remainder(x: int, y: int) -> int:
    return x if x < y else remainder(x - y, y)
```

Both give `remainder(17, 7) == 3`. They are previews of python-while-loops[48] and python-recursion[49].

[48] courses/csse1001/python-while-loops.pdf
[49] courses/csse1001/python-recursion.pdf

Python Primitive Data Types

The types "built-in" to Python by default are called **primitive data**: booleans, integers, floats, strings, and tuples. All of them are **immutable** — once created, an object of one of these types cannot be changed.

Booleans

The boolean type provides, comprehensively, the values `True` and `False`:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Comparison operators

Having a boolean type is what makes **comparison operators** useful — they return a true/false answer:

```
>>> 3 == 9
False
>>> 3 > 7
False
>>> 3 >= 7
False
```

All comparison operators have **equivalent precedence** and are **left associative**; when mixed with arithmetic, comparison is evaluated *last* (see python-operator-precedence-and-associativity[50]).

Operator	Description
`==`	Equal
`!=`	Not equal
`<`, `<=`	Less than, less than or equal
`>`, `>=`	Greater than, greater than or equal

[50] courses/csse1001/python-operator-precedence-and-associativity.pdf

```
>>> 3 + 2 > 1 + 3   # arithmetic first
True
>>> 3+2 > 1+3       # better to group like this
True
>>> 3 + (2>1) + 3   # True has int value 1
7
```

Logical connectors

Functions that return booleans are called **predicates**, and are (very) important to programming. More sophisticated predicates are built with the logical connectors `and`, `or`, and `not` — covered in detail in 2025-08-11-sequence-selection-and-iteration[51] and python-boolean-logic[52].

`bool` — truthy / falsy

Any object can be converted to a boolean with `bool`. Data that converts to `True` is **truthy**; data that converts to `False` is **falsy**. Usually the falsy element is whatever acts as zero in the type, and everything else is truthy.

```
>>> bool(0)
False
>>> bool(1)
True
```

Integers

> **Definition (Integer).** An **integer** is a number *without* a fractional part. It can be zero, positive, or negative.

Integers are **unbounded** in Python: we can work with integers of arbitrary size without worrying about **overflow**, which is not typical of programming languages.

```
>>> 2 ** 256 - 1
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

Any object can be converted to an integer with `int`. `True` behaves as `1` and `False` as `0`, so booleans can be used in arithmetic:

[51] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[52] courses/csse1001/python-boolean-logic.pdf

```
>>> int(True)
1
>>> int(False)
0
>>> True + False
1
>>> True * False
0
```

Strings

> **Definition (String).** Anything (with some exceptions) enclosed by single-quotes `' '` or double-quotes `" "` is considered a **string** by Python. A string is an **ordered collection** of the characters (e.g. unicode and ascii) allowed by the computer.

The quotes are what make it a string; without them the same text is either a syntax error or a name:

```
>>> "hello world"
'hello world'
>>> type("hello world")
<class 'str'>
>>> hello world   # note the lack of quotes
SyntaxError: invalid syntax
>>> hello         # note the lack of quotes
NameError: name 'hello' is not defined
```

`str` conversion and formatting

Any object can be converted to a string with `str`:

```
>>> str(1)
'1'
>>> str(True)
'True'
>>> str("Hello")
'Hello'
```

Substitution converts values into strings inside a sentence, using the string formatter — an `f` before the quotes, and `{}` around each expression to substitute:

```
>>> x = 1
>>> y = "two"
>>> z = f"x is {x} y is {y}"     # formatted string
>>> z
'x is 1 y is two'
>>> print(f"x is {x} y is {y}")
x is 1 y is two
>>> f"{x}"                       # alternative to str
'1'
```

Concatenation and scalar multiplication

Adding strings concatenates them — and because a string can be added to itself repeatedly, it also makes sense to multiply a string by a positive integer:

```
>>> "hello" + "world"
'helloworld'
>>> type(" ")
<class 'str'>
>>> space = " "
>>> "hello" + space + "world"
'hello world'
>>> "Hello World!" + "Hello World!" + "Hello World!"
'Hello World!Hello World!Hello World!'
>>> 3 * "Hello World!"
'Hello World!Hello World!Hello World!'
```

When strings are added a **new string is created** — the originals are untouched, which is what immutability forces.

Comparing strings

Strings compare by character order; `ord` gives a character's order and `chr` recovers the character from the order:

```
>>> "a" < "b"
True
>>> ord("a"), ord("b")   # the "order" of the characters
(97, 98)
>>> chr(97)              # get character from order
'a'
```

```
>>> "A" < "a"
True
>>> "Z" < "a"
True
```

Comparison then proceeds character-by-character, with a shorter string counting as less than any extension of itself:

```
>>> "a" < "aa"
True
>>> "b" < "aa"
False
>>> "aba" < "ab"
False
>>> "aZ" < "aa"
True
```

> **Exercise.** Write `string_less_than(cs: str, ds: str) -> bool`, returning `True` only when `cs < ds`, restricting yourself to comparing integers only. `string_less_than("aba", "ab")` is false and `string_less_than("aZ", "aa")` is true.

Escape characters

A **new line** is an escape character that prints whatever follows it on a new line; the escape character is `\n`. A **tab** (`\t`) is a fixed amount of horizontal space, though how it is displayed depends on the program doing the displaying — which is why tabs are the worst.

```
>>> "hello\nworld"
'hello\nworld'
>>> print("hello\nworld")
hello
world
>>> "hello\tworld"
'hello\tworld'
>>> print("hello\tworld")
hello   world
```

A string can be **stored** differently than it is **printed**.

Numbers versus strings

`+` never silently converts between `int` and `str` — mixing them is a `TypeError`, and you have to say which conversion you meant:

```
>>> 3 + 7
10
>>> "3" + "7"
'37'
>>> 3 + "7"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(3) + "7"
'37'
>>> 3 + int("7")   # convert string to integer
10
```

Converting a string to a number only works for numbers expressed with digits, and `int` will not take a fractional literal:

```
>>> int("3.14")
ValueError: invalid literal for int() with base 10: '3.14'
>>> float("123.456")
123.456
>>> int("seven")
ValueError: invalid literal for int() with base 10: 'seven'
```

Length and inclusion

A string's **length** is the number of characters that comprise it:

```
>>> len("h")
1
>>> len("hello")
5
>>> cs = "world"
>>> len(cs)
5
>>> len(cs+"world") == len(cs) + len("world")
True
```

We can ask whether a string is a **substring** of another with `in`:

```
>>> "h" in "hello world"
True
>>> "hello" in "hello world"
True
>>> cs = "world"
>>> cs in "hello world"
True
>>> "ow" in "hello world"
False
```

Indexing

Because a string is *ordered*, we can number its characters **starting from zero** and access them with square brackets. We can also index from the end with negative numbers:

```
>>> cs = "hello world"
>>> cs[0]
'h'
>>> cs[1]
'e'
>>> cs[2]
'l'
>>> cs[len(cs)]
IndexError: string index out of range
>>> cs[-1]
'd'
>>> cs[-2]
'l'
>>> cs[-11]
'h'
```

Slicing

A slice `cs[start:stop:step]` takes *start* inclusive through *stop* exclusive, moving by *step*. The examples use `cs = "0123456789"` throughout, so that each index matches the character sitting at it.

Omitting an endpoint defaults it to the far end in the direction of travel:

```
>>> cs = "0123456789"
>>> cs[1:4]      # 1st inclusive through 4th exclusive
'123'
```

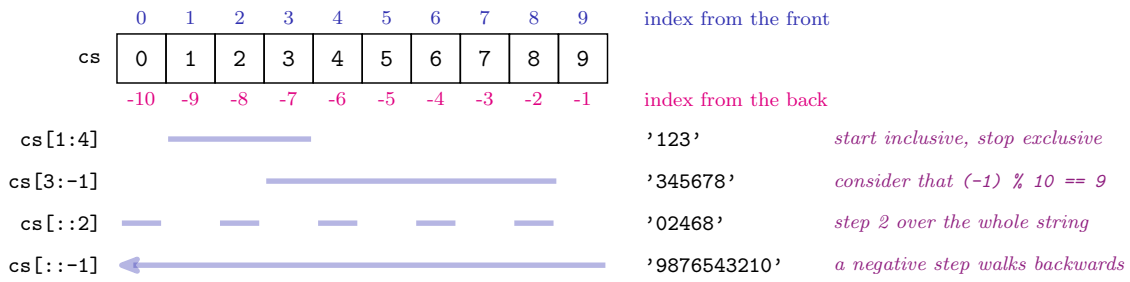


Figure 7: The string `cs = '0123456789'` with both index systems and four of its slices. Positive indices count from the front, negative from the back, and the two agree because a negative index i names position $i \bmod 10$ — which is why `cs[3:-1]` stops at 8, one before position 9.

```
>>> cs[0:9]
'012345678'
>>> cs[0:10]
'0123456789'
>>> cs[0:]       # omitting the right endpoint defaults to the line above
'0123456789'
>>> cs[:-1]      # omitting the left endpoint defaults to zero
'012345678'
>>> cs[:]        # omitting both gives everything
'0123456789'
```

Negative endpoints are the same positions counted from the back — read a negative index modulo the length:

```
>>> cs[-1] == cs[len(cs)-1]
True
>>> cs[3:-1]     # consider that (-1) % 10 == 9
'345678'
>>> cs[-7:]      # consider that (-7) % 10 == 3
'3456789'
```

A step may be given third, and a negative step travels backwards:

```
>>> cs[::2]      # increment by two, zero inclusive to ten exclusive
'02468'
>>> cs[0:10:2]   # the above defaulted to this
'02468'
```

```
>>> cs[::-1]     # decrement by one, ten inclusive to zero exclusive
'9876543210'
>>> cs[10::-1]   # the above defaulted to this
'9876543210'
>>> cs[:10:-1]   # not to this, because we are moving backwards
''
```

Mixing all three, forwards and backwards:

```
>>> cs[1:-3:3]    # increment by three, one inclusive to seven exclusive
'14'
>>> cs[-8:-2:2]   # increment by two, two inclusive to eight exclusive
'246'
>>> cs[-8:-2:-4]  # decrement by four, two inclusive to eight exclusive
''
>>> cs[::-4]      # decrement by four, ten inclusive to zero exclusive
'951'
```

Immutability

Something is **immutable** when it cannot be changed. **Strings are immutable** — so a word cannot be capitalised by writing over its first character:

```
>>> cs = "hello"
>>> cs[0] = "H"
TypeError: 'str' object does not support item assignment
```

Strings as booleans

The empty string is the falsy string; every other string is truthy. It is also "smaller" than every other character:

```
>>> bool("a")
True
>>> bool("Hello")
True
>>> bool("")
False
>>> "" < "A"
True
>>> "" < chr(0)
True
```

A space is not the same thing as empty:

```
>>> space = " "      # a space
>>> len(space)
1
>>> bool(space)
True
>>> empty = ""       # empty string
>>> len(empty)
0
>>> bool(empty)
False
```

Tuples

> **Definition (tuple).** A **tuple** is an *immutable* ordered collection of elements. These elements are *not necessarily* the same type and are *not* necessarily distinct. Round brackets `()` are used to create tuples in Python.

Being ordered, tuples index and slice exactly like strings; being immutable, they refuse item assignment exactly like strings:

```
>>> xs = (0, 1, 2, 3, 4, 5)
>>> type(xs)
<class 'tuple'>
>>> xs[-1]        # indexing
5
>>> xs[0:4]       # slicing
(0, 1, 2, 3)
>>> xs[0] = -1    # immutable
TypeError: 'tuple' object does not support item assignment
```

Tuples support the same addition and scalar multiplication as strings, and their elements can be of mixed type:

```
>>> xs = (1, 'a', 2, 'b')    # mixed types are fine
>>> ys = (3, 'c', 4, 'd')
>>> xs + ys                  # tuple addition
(1, 'a', 2, 'b', 3, 'c', 4, 'd')
>>> 2*xs                     # tuple scalar multiplication
(1, 'a', 2, 'b', 1, 'a', 2, 'b')
```

Nomenclature

Tuples are named by size: couple (2), triple (3), quadruple (4), quintuple (5), sextuple (6), septuple (7), octuple (8), and an n-tuple in general. The general form `(0, 1, 2, ..., n-1)` is *not* valid Python syntax — it is informal notation for the pattern.

Singleton and empty tuples

A single value in round brackets is **not** a tuple — brackets there are just grouping. The trailing comma is what makes a singleton:

```
>>> (1) + (2)
3
>>> type((1))
<class 'int'>
>>> (1)
1
>>> type((1,))
<class 'tuple'>
>>> (1,) + (2,)
(1, 2)
>>> (1) + (2,3)   # common error
TypeError: unsupported operand type(s)
```

The empty tuple is `()` — a genuine tuple, the identity for tuple addition, and falsy:

```
>>> ()
()
>>> type(())
<class 'tuple'>
>>> () + (1,)
(1,)
>>> (1,) + ()
(1,)
>>> bool(())
False
```

Comparing tuples

Tuples compare element-by-element, and — like strings — a shorter tuple is less than a longer tuple that extends it:

```
>>> (1, 2, 3) == (1, 2, 3)
True
>>> (1, 2, 3) == (1, 2)
False
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3) < (1, 2)
False
>>> (1, 2) < (1, 2, 3)
True
```

Packing and unpacking

Several names can be assigned on one line, and several values printed on one line, through an (implicit) tuple:

```
>>> x, y, z = 2, 3, 4    # saves vertical space
>>> x
2
>>> y
3
>>> z
4
>>> x, y, z              # repl-printing several values
2, 3, 4
```

A real REPL echoes the last line as the tuple it built, brackets included — `(2, 3, 4)` — which is the more useful way to read it: the right-hand side of a packing assignment is a tuple.

Python Recursion

A **recursive function** is a function that calls itself. Every recursion needs at least one of each:

- **Base case**: a case defined outright, with no further recursive call (`if base_case: return constant`).
- **Recursive step**: a line where the function calls itself on "smaller" input, making progress toward a base case.

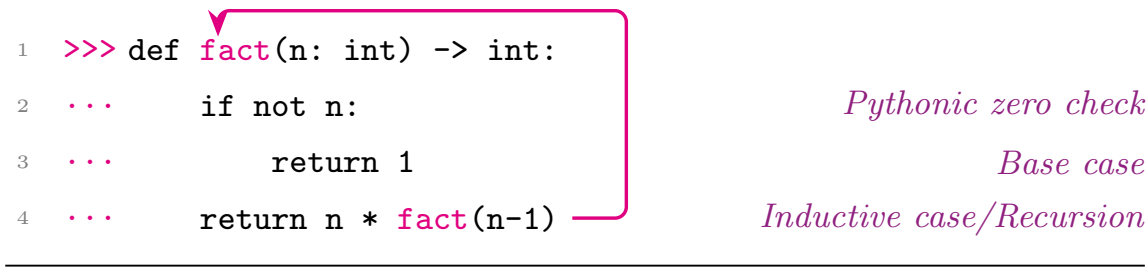


Figure 8: The anatomy of a recursive function. The purple notes on the right name each part. The two occurrences of fact are coloured magenta and joined by an arrow, to show the one thing the definition of recursion turns on: line 4 calls the function being defined on line 1.

Canonical example: factorial

```
def fact(n: int) -> int:
    if not n:              # base case
        return 1
    return n * fact(n-1)   # recursive step
```

Evaluation has two phases:

- **Winding**: recursive calls build up a stack of pending computations (`4 × (3 × (2 × (1 × fact(0))))`). Each call pushes an *activation record*; they complete last in, first out.
- **Unwinding**: once the base case is hit, each pending computation resolves from the inside out.

See 2025-10-20-recursion[53] for the full winding/unwinding trace of `fact(4)` drawn out line by line.

If a base case is never reached, the recursion **bottoms out** with a `RecursionError: maximum recursion depth exceeded` once Python's call stack limit (1000 by default) is hit — `fact(-1)` does this, since `n` only gets more negative. The limit can be raised with `sys.setrecursionlimit(n)`:

```
>>> import sys
>>> sys.setrecursionlimit(1500)
```

[53] courses/csse1001/2025-10-20-recursion.pdf

Overflowing the stack and bottoming out are **different bugs with the same error message**. Raising the limit fixes a recursion that legitimately needs more depth than Python allows (`fact(999)` works once the limit is 1001); it never fixes a missing or unreachable base case (`fact(-1)` fails at any limit).

Choosing base cases

Type	Typical base case
`int` (counting down)	`0`
`list`	`[]`
`str`	`''` (empty string)

For less obvious base cases, take an example that makes a *single* recursive call and ask what the base case **must be** for that example to come out right. For example, $2^k = 2 \cdot 2^{k-1}$ leaves open what "2 multiplied 0 many times" is. Since $2^1 = 2$ and $2^1 = 2 \cdot 2^0$, the base case must be $2^0 = 1$.

The same advice stated the other way round: set the base cases to whatever makes the **singleton** case (the second-smallest one) work.

Linear vs. tree recursion

Most recursive functions make a single recursive call per invocation (linear recursion, e.g. `fact`, `is_palindrome`, `increasing`, `sum_of_digits`) and so unwind along one chain. Some make more than one (tree recursion), e.g. checking whether any sublist sums to a target:

```
def sublist_sum(xs: list[int], target: int) -> bool:
    if not xs:
        return not target
    return sublist_sum(xs[1:], target-xs[0]) or sublist_sum(xs[1:], target)
```

Two calls per invocation means 2^n leaf calls for a list of n elements. See 2025-10-20-recursion[54] for the call tree drawn out, and for the other branching examples (`num_paths`, `lcs`, `hanoi`, naive `fib`).

Tree recursion can recompute the same subproblem many times. **Dynamic programming** — caching previous results — avoids this when subproblems overlap heavily (the classic example being naive recursive Fibonacci, which is exponential without a cache but linear with one). Here the cache is a module-level dict seeded with the base cases:

[54] courses/csse1001/2025-10-20-recursion.pdf

```
fib_cache = {0: 1, 1: 1}
def fib(n: int) -> int:
    global fib_cache
    if n in fib_cache:
        return fib_cache[n]
    fib_cache[n] = fib(n-1) + fib(n-2)
    return fib_cache[n]
```

Relationship to induction

Recursion is essentially the Principle of Mathematical Induction realised as code: a base case (the $P(0)$ case) plus a step that reduces a general case to a smaller one already known to work (the $P(n) \implies P(\text{succ}(n))$ step). The recursive call *is* the appeal to the induction hypothesis — which is easiest to see in the Towers of Hanoi proof, where the hypothesis is used twice and the code makes exactly two recursive calls.

See python-scope[55] for how local variables/recursive calls interact with scope, and python-composition[56]/python-inheritance[57] for other structuring tools. See towers-of-hanoi[58] for a worked induction/recursion example, and algorithmic-complexity[59] for the vocabulary that makes "exponential without a cache, linear with one" precise — merge sort and quick sort in sorting-algorithms[60] are both recursive.

Python While Loops

A **loop** is a control structure that repeats code that belongs to it. A **while-loop** is a loop that repeats code *while some condition is satisfied*:

```
while <condition>:
    <code>
```

The condition is checked before every repetition (including the first) — if it's false to begin with, the loop body never runs at all:

[55] courses/csse1001/python-scope.pdf
[56] courses/csse1001/python-composition.pdf
[57] courses/csse1001/python-inheritance.pdf
[58] courses/csse1001/towers-of-hanoi.pdf
[59] courses/csse1001/algorithmic-complexity.pdf
[60] courses/csse1001/sorting-algorithms.pdf

```
>>> x = 0
>>> while False:
...     print(x)
...     x += 1
>>> # nothing prints
```

See the while-loop control-flow figure in 2025-08-18-while-loops[61] for a picture of this.

Augmented assignment operators

Operator	Equivalent to
`x += y`	`x = x + y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

These are the four used in the course. They are what makes an accumulator loop readable:

```
>>> x = 0
>>> while x < 10:
...     x += 1
>>> x
10
```

`break`

The `break` keyword terminates the (innermost) loop immediately and continues with the rest of the program — so code after it *inside the loop body* is unreachable:

```
>>> while True:
...     print("hello")
...     break
...     print("world")
hello
```

[61] courses/csse1001/2025-08-18-while-loops.pdf

Simulating a do-while

A **do-while** (or **repeat-until**) is a while-loop variant available in other languages but not natively supported in Python. It runs its code *at least once*, because the condition is only tested after the body:

```
do:
    <code>
while <condition>   # not valid Python syntax
```

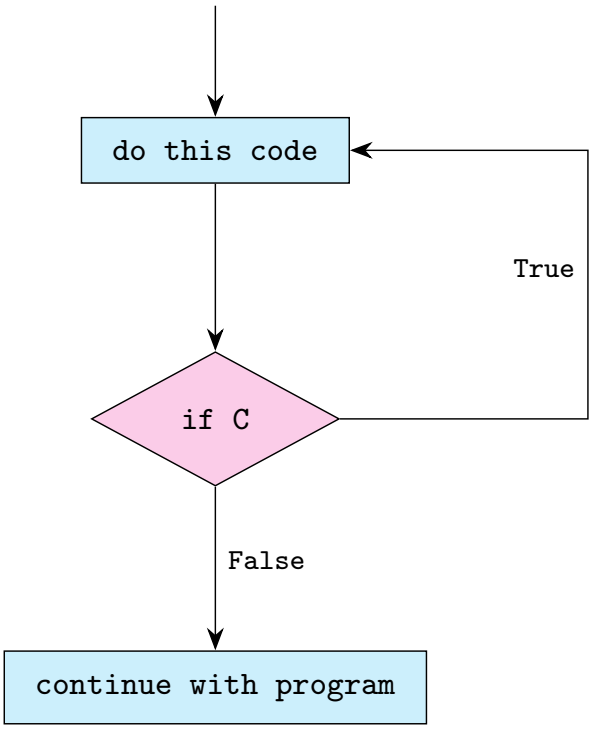


Figure 9: The do-while as a flowchart. Compare the while-loop figure in 2025-08-18-while-loops[62]: the body is above the test rather than below it, so the body always runs once before `C` is ever evaluated, and the `True` edge is the one that loops back.

Although not natively supported in Python, we can simulate one with `while True` and a `break` (some programmers prefer `1` to `True`):

```
>>> x = 0
```

[62] courses/csse1001/2025-08-18-while-loops.pdf

```
>>> while 1:
...     x += 1
...     if x > 0:
...         break
>>> x            # x retains its value outside the while
1
```

`x` ends at `1`, not `0`: the body ran once before the exit test was reached, which is the whole point of the shape.

User input

`x = input()` waits for input from the keyboard and assigns it to `x`, always with `type(x) is str`. Numeric input therefore has to be converted — `int(input("Guess: "))`.

Random number generation

Random number generation is handled by an external library. It is **not** built-in and therefore must be imported. Importing the single command:

```
>>> from random import randint
>>> randint(1, 6)   # chosen uniformly over the interval
2
```

or the entire library:

```
>>> import random
>>> random.randint(1, 6)
2
```

Both endpoints are included: `randint(1, 6)` is the course's six-sided die, and its prompt reads `Guess in [1..6]`.

Sorting Algorithms

Five ways to solve one problem — sorting a list. Each is derived from a different restriction: *suppose the only thing you could do was X, could you still sort?* Taken together they are the course's worked example for comparing algorithms, which is what algorithmic-complexity[63] formalises.

Sorting is also the course's very first example of an algorithm at all (2025-07-29-a-brief-history-of-computing[64]). The selection-sort trace below needs no code beside it: *what* is being expressed matters more than the language it is expressed in.

Selection sort

> Suppose we could only *select* (find the index of) the largest element from a list.

Find the largest member of `xs` and make it the last element. Then find the next largest and make it the second last, and so on. After k passes the last k elements are in their final positions.

Two things the trace shows that the one-line description does not. The list of nine takes only eight passes — once the last eight are placed the first element has nowhere else to go, which is the `17 ok` row. And the *work* is invisible in the picture: each row costs a full scan of the unsorted prefix to find its grey cell, and it is those scans, not the eight moves, that make selection sort quadratic.

Insertion sort

> Suppose we could only *place an element correctly into an already-sorted list.*

Sort the list by inserting elements repeatedly into the front portion of the list. After k steps the first k elements of the list are sorted (though not necessarily in their final positions relative to the rest).

Bubble sort

> Suppose the only operation we could perform was *swapping adjacent cells.*

Pass through the list swapping neighbouring elements so that larger elements move forward — larger elements "bubble" to the top. Make repeated passes until the list is sorted.

[63] courses/csse1001/algorithmic-complexity.pdf
[64] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf

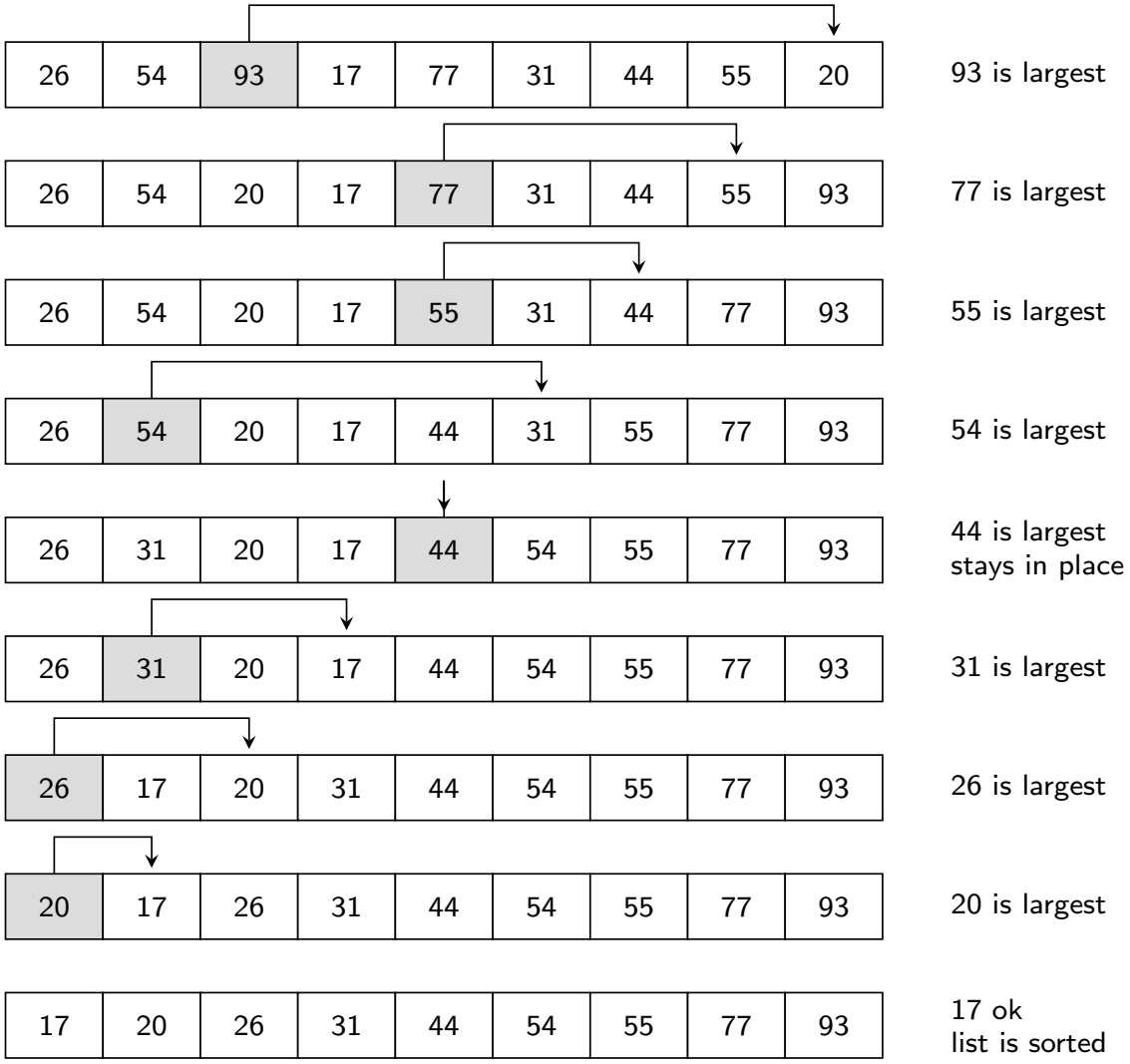


Figure 10: Selection sort on a list of nine. Each row is one pass; time runs downward. The grey cell is the largest element of the still-unsorted prefix and the arrow above the row takes it to the position it will keep, so after k rows the last k cells are final. In row five 44 is already where it belongs, so the arrow points straight down and nothing moves.

Merge sort

> Suppose the only operation we could perform was *merging two sorted lists into one sorted list*.

Break the list in half, sort each half (using merge sort — this is recursion[65]), then merge the two sorted halves.

The evaluation has the same two phases as any recursion: a **winding** phase where the list is repeatedly halved down to trivial pieces, and an **unwinding** phase where the sorted pieces are merged back up.

Quick sort

> Could we sort a list if all we could do was *split* a list into two pieces based on some predicate?

Divide the list into two lists — those greater than or equal to the head, and those less than or equal to the head. Sort those lists (recursively) and then just concatenate them.

Which is best?

Rather than timing them, count how many times the characteristic operations are performed — for sorting, that means **comparisons** and **swaps**.

Algorithm	Complexity
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Bubble sort	$O(n^2)$
Merge sort	$O(n \ln n)$

Merge sort's divide-and-conquer structure is what buys it the better class: the halving contributes a $\log n$ factor and each level of the recursion does $O(n)$ merging work.

Exercise: implement selection, insertion, bubble and merge sort. The `hanoi` function in 2025-10-20-recursion[66] is the closest worked example of the same divide-and-conquer shape that merge sort and quick sort use.

[65] courses/csse1001/python-recursion.pdf
[66] courses/csse1001/2025-10-20-recursion.pdf