Programming Paradigms

Table of contents

Imperative programming 1
Declarative programming 2
Paradigm is not the same as level of abstraction 2
Why Python has several 3
Where OOP fits 3

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[1], python-if-statements[2], python-while-loops[3], python-for-loops[4].
2. **Object-oriented** — groups instructions into *objects* that combine data (state, attributes) and behaviour (methods). This is weeks 7–11: python-classes-and-objects[5] onward.

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

[1] courses/csse1001/python-functions.pdf
[2] courses/csse1001/python-if-statements.pdf
[3] courses/csse1001/python-while-loops.pdf
[4] courses/csse1001/python-for-loops.pdf
[5] courses/csse1001/python-classes-and-objects.pdf

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.

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[6].

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[7]). 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++)`

[6] courses/csse1001/python-comprehensions.pdf
[7] courses/csse1001/history-of-computing.pdf

Language	How it repeats
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:

- **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[8]) 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[9].

Why Python has several

Python was created to *support a variety of programming styles* rather than enforce one (see history-of-computing[10]). 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`.

[8] courses/csse1001/2025-07-29-python-as-a-calculator.pdf
[9] courses/csse1001/python-operator-precedence-and-associativity.pdf
[10] courses/csse1001/history-of-computing.pdf

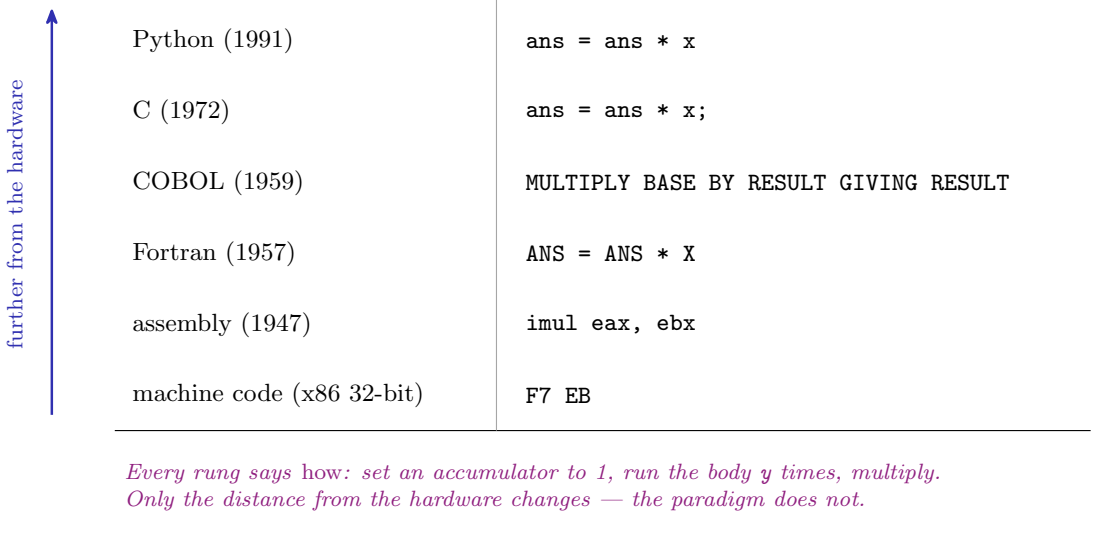


Figure 1: The one step that survives all six languages. Each row is that language's way of writing *multiply the accumulator by x*. The rows run from the oldest language at the bottom to the newest at the top. The bottom two rows are the same instruction twice: `F7 EB` is what the machine stores, `imul eax, ebx` is the assembly for it.

Continue in python-classes-and-objects[11] for the mechanics, and 2025-09-09-object-oriented-programming[12] for the lecture that introduces them.

[11] courses/csse1001/python-classes-and-objects.pdf
[12] courses/csse1001/2025-09-09-object-oriented-programming.pdf