











Types of Programming Languages

Python has been gaining popularity since its release in 1991 and is currently the most in-demand language in the world ([TIOBE Index](#)).

Apr 2022	Apr 2021	Change	Programming Language		Ratings	Change
1	3	▲		Python	13.92%	+2.88%
2	1	▼		C	12.71%	-1.61%
3	2	▼		Java	10.82%	-0.41%
4	4			C++	8.28%	+1.14%
5	5			C#	6.82%	+1.91%
6	6			Visual Basic	5.40%	+0.85%
7	7			JavaScript	2.41%	-0.03%
8	8			Assembly language	2.35%	+0.03%
9	10	▲		SQL	2.28%	+0.45%
10	9	▼		PHP	1.64%	-0.19%

But why is Python so popular? How is it different from other languages, and does it have any limitations? To answer these questions, take a look at the most common characteristics of programming languages.

Each language can be described using the following characteristics:

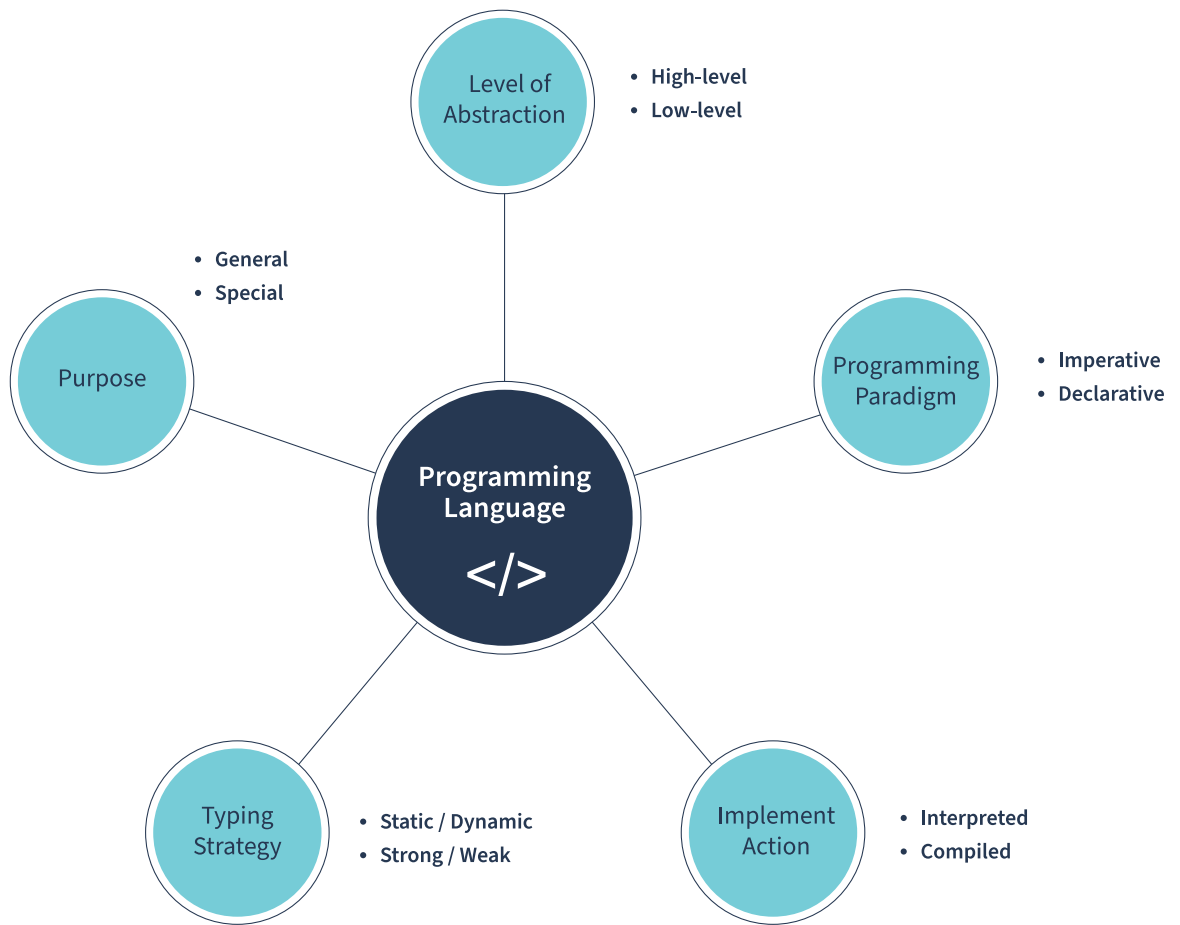
Now, explore each one in detail.

Purpose

Languages can differ in terms of the scope of tasks they perform.

But why is Python so popular? How is it different from other languages, and does it have any limitations? To answer these questions, take a look at the most common characteristics of programming languages.

Each language can be described using the following characteristics:



Now, explore each one in detail.

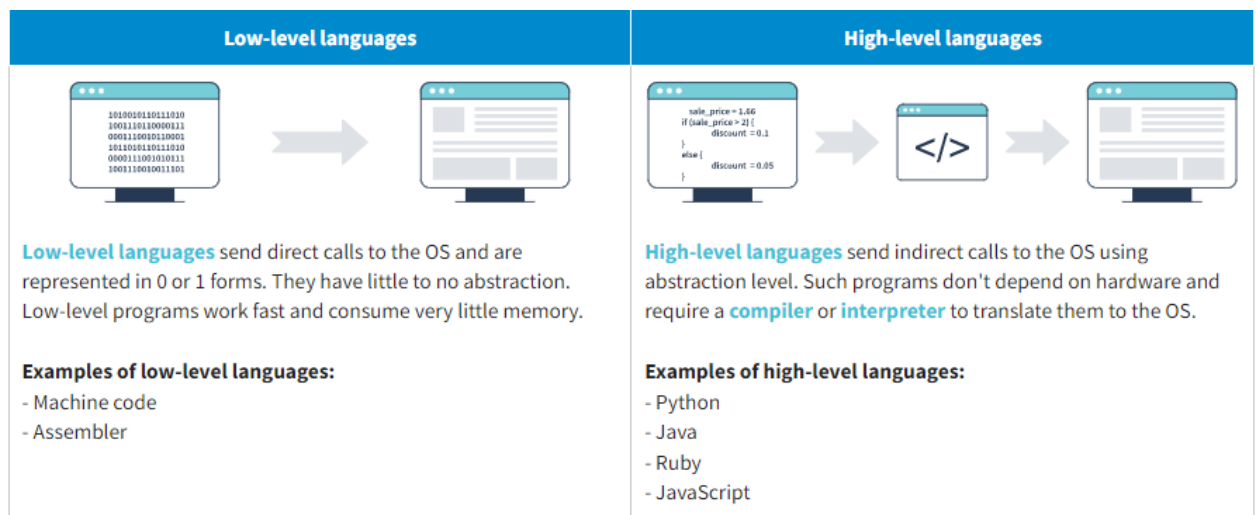
Purpose

Languages can differ in terms of the scope of tasks they perform.

General programming languages	Special programming languages
<p>These languages can solve various tasks without specific language constructions.</p> <p>Examples:</p> <ul style="list-style-type: none"> - Python - Rust - Java - Lisp 	<p>These languages were created to solve specific kinds of problems, which imposes certain restrictions on how they are used.</p> <p>Examples:</p> <ul style="list-style-type: none"> - Maple for mathematical computations - SQL for database queries

Level of Abstraction

Another characteristic is the level of abstraction of the programming language when interacting with hardware.



Programming Paradigm

Some languages are better suited for a specific way of programming than others.

There are two main programming paradigms: **declarative** and **imperative**. These terms don't refer to specific languages but to the programming styles they employ.

Declarative paradigm

Declarative languages describe the intended **result** of a program without specifying how to achieve it. You can use a language's features, extensions, or libraries to do this.

Suppose you want a pizza. According to the declarative paradigm, you describe what kind of pizza you want — pepperoni—but not how it is prepared. It can be ordered from a restaurant or cooked in a kitchen. The process doesn't matter to you.

Examples of declarative languages:

- HTML—You don't need to know how a browser renders HTML. You describe the structure of the webpage.
- SQL—You don't need to know how queries work. You just describe the result.

Imperative paradigm

The main goal of **imperative programming** is to describe the **process** of achieving a result. An imperative language describes how to execute a program command by command.

Returning to the pizza analogy, this time, you need to describe the process:

Make dough → Add pepperoni → Add tomatoes → Bake pizza

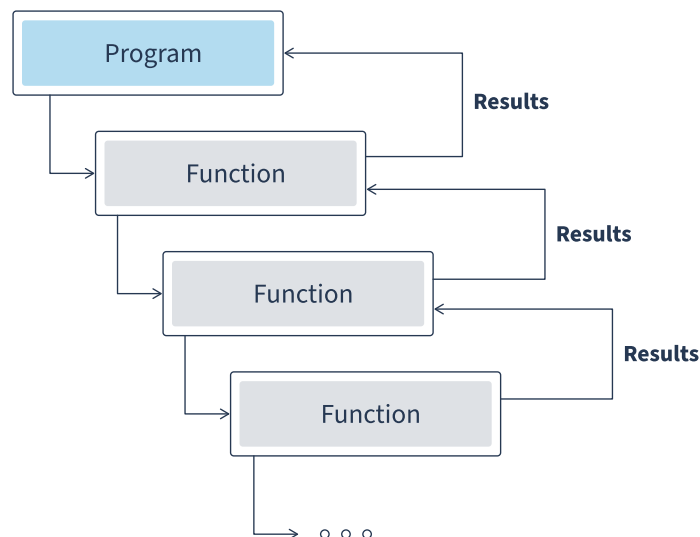
Examples of imperative languages:

- Python
- Java
- Ruby
- JavaScript

Both declarative and imperative paradigms can be subdivided into even more specific ways of programming:

Functional paradigm

- **The functional paradigm** is an example of declarative programming in which programs are executed as a chain of function calls. Such a chain forms a recursion: Functions accept inputs and return outputs that can be used as input by the consecutive functions.
- You can apply standard functions or define new ones. What's more, functional programs don't have a state. Therefore, the data doesn't change; it just gets copied.



Advantages:

- It is easier to understand, test, debug, and support code because the program doesn't have a state. Functions depend only on input data.
- Functions can take other functions as arguments and return functions as a result.
- This enables parallel execution of the same functions because they don't have a state.

Disadvantages:

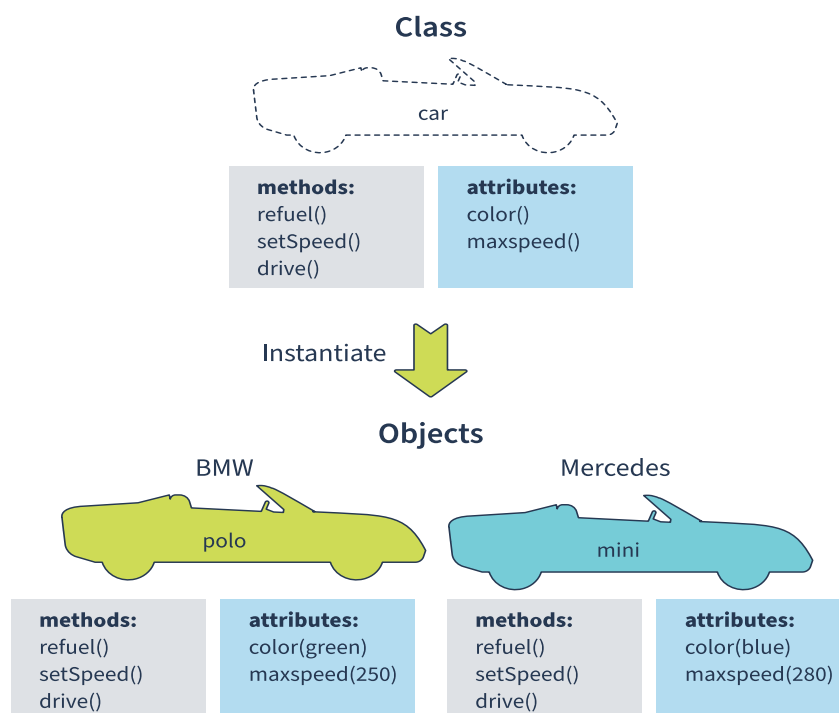
- Recursion depth is limited in some languages.
- Code readability can be affected.
- Immutability of the values can cause problems with performance since you have to copy the values every time.

Examples of functional languages:

- Erlang
- Haskell
- F#
- Wolfram Language

Object-oriented paradigm

- **An object-oriented paradigm** is a form of imperative programming. According to this paradigm, a program interacts with a set of *objects* instantiated from *classes*. Classes have *attributes* (data stored in classes) and *methods* (code to manage the data).
- For example, a program has a car class that serves as a template for other car objects within the program. The program should first instantiate it from the class to interact with the object. Objects inherit methods and attributes: New cars can *drive* and *change speed* and have the *same color* as their parental class. At the same time, the car objects have a unique state (*color* and *maxspeed*).



Advantages:

- Parallel development. Each team member can work independently with their own module/classes.
- Scalable. Very often, classes can be reused.

- Maintainable. The coding base is centralized, so it is easier to create maintainable code. That makes it easier to keep your data accessible when it becomes necessary to perform an upgrade.

Disadvantages:

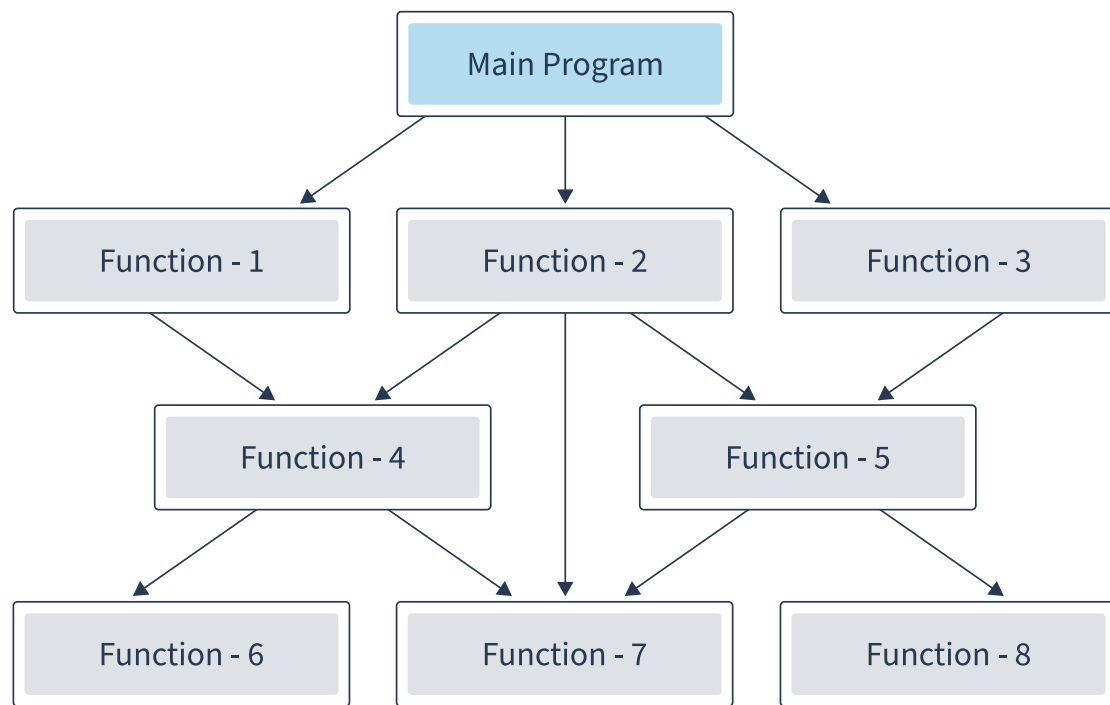
- Can be inefficient. Using an OOP can increase CPU usage.
- Unnecessary classes. An insufficiently well-designed and thought-out inheritance structure can lead to quite a large number of unnecessary classes.
- Duplication. OOP projects are quite easy to develop but sometimes quite difficult to implement. You can get up new projects and run them at a greater speed. But sometimes projects look like they've been cloned.

Examples of object-oriented languages:

- Python
- Ruby
- C++

Procedural paradigm

The procedural paradigm is yet another form of imperative programming. It requires grouping sequences of instructions into procedures. A procedure can store data that is accessed only from within the procedure. You can call any procedure at any part of the code.



Structure of procedural-oriented programs

Advantages:

- Easy to reuse procedures
- Consumes less memory than other paradigms

Disadvantages:

- No data protection, unlike in other paradigms
- Harder to write a procedural program
- Difficult to handle errors

Examples of procedural languages:

- C
- Pascal
- Ada

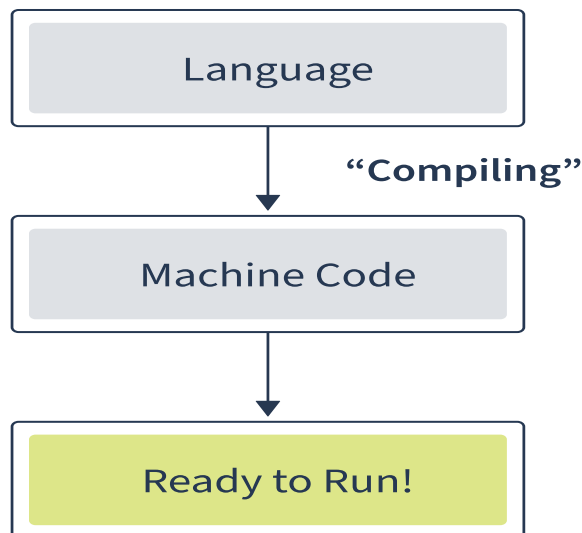
Implementation

High-level languages make developing a program easier and faster than low-level languages. However, the OS doesn't directly understand source code written using some high-level language. The special programs, which are called compiler and interpreter, allow transforming source code to code understandable by the OS. Based on this, there are two large groups of languages:

Compiled languages

Compilation is the process of translating source code from high-level programming language to lower-level language (e.g., assembly code, object code, or machine code) to create an executable program (["Compiler" \(2022\) Wikipedia](#)).

Compilation



Advantages:

- Faster execution than interpreted language

Disadvantages:

- It takes time to compile the program before its execution
- Compiled code depends on the compilation platform

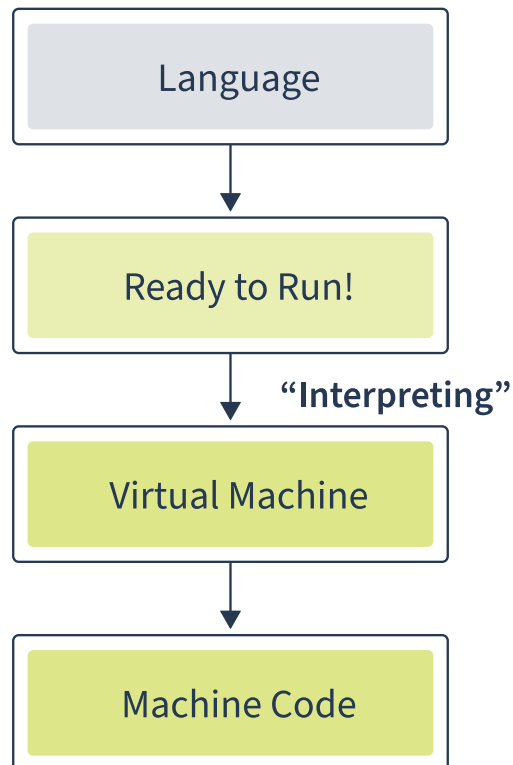
Examples of compiled languages:

- C
- Go
- Pascal

Interpreted languages

Interpretation transforms source code to bytecode (an intermediate representation language), which is then executed by the [interpreter](#) step by step. Interpreted languages are cross-platform but take longer to execute than compiled languages.

Interpretation



Advantages:

- Platform-independent code
- Smaller programs than compiled languages

Disadvantages:

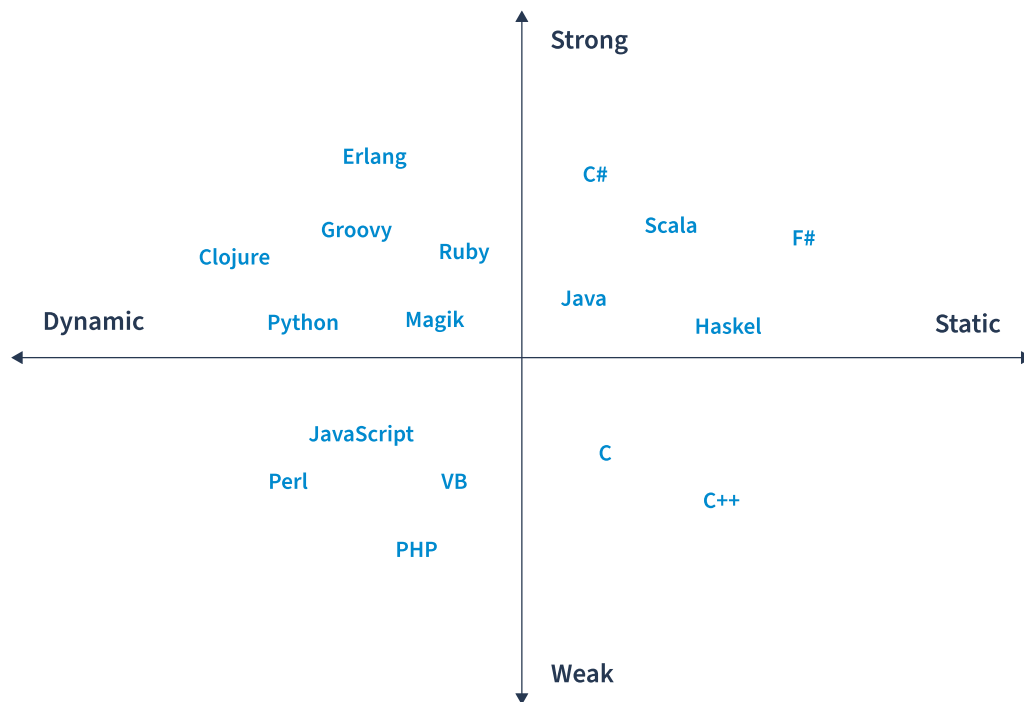
- Slower execution

Examples of interpreted languages:

- Python
- Lisp
- PHP

Typing Strategy

Type-checking is the process of verifying the type of construct and its usage context. This helps to minimize the possibility of type errors in the program.



Strong

Strong typing languages permit mixing types in expressions and don't apply different implicit casts of types.

Dynamic

In **dynamic typing**, code is checked at runtime, and there is no need to specify the data type of each variable. In dynamic languages, variables can store any data type, and you can change this at any time during program execution. This speeds up development because you can reuse existing variables to store new values.

Static

Static typing checks are performed without running the program, so every detail of the variables and data types must be known before compiling. This means that you can handle most bugs in the code during compilation.

Weak

Weak typing languages apply implicit casts of types. As a result, the output of some expressions may be surprising.

Take a look at some examples of code:

Dynamic vs. Static

Dynamic / Python

```
def mean( array ):  
    result = 0  
    for item in array:  
        result += item  
    return result / len(array)
```

Static / C

```
double mean( double array[], int size ) {  
    double result = 0.0;  
    for ( int i = 0; i < size; ++i )  
    {  
        result += array[i];  
    }  
    return result / size;  
}
```

Strong vs. Weak

Strong / Python

```
number = 42 + "666"  
TypeError: Unsupported operand type(s)  
for +: "int" and "str"  
  
number = str(42) + "666"  
"42666"  
  
number = 42 + int("666")  
708  
  
number = "42" + "666"  
"42666"
```

Weak / JavaScript

```
var number = 42 + "666"  
"42666"
```