**Object-oriented programming (OOP)** is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm" \o "Programming paradigm) based on the concept of "[objects](https://en.wikipedia.org/wiki/Object_(computer_science)" \o "Object (computer science))", which can contain **[data](https://en.wikipedia.org/wiki/Data" \o "Data)**and code: data in the form of [fields](https://en.wikipedia.org/wiki/Field_(computer_science)" \o "Field (computer science)) (often known as **[attributes](https://en.wikipedia.org/wiki/Attribute_(computing)" \o "Attribute (computing))**or **properties**), and **code**, in the form of **procedures** (often known as **[methods](https://en.wikipedia.org/wiki/Method_(computer_science)" \o "Method (computer science))**, **behaviour**).

A feature of objects is that an object's own procedures can access and often modify the data fields of itself (objects have a notion of [this](https://en.wikipedia.org/wiki/This_(computer_programming)" \o "This (computer programming)) or self).

In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are [class-based](https://en.wikipedia.org/wiki/Class-based_programming" \o "Class-based programming), meaning that objects are [instances](https://en.wikipedia.org/wiki/Instance_(computer_science)" \o "Instance (computer science)) of [classes](https://en.wikipedia.org/wiki/Class_(computer_science)" \o "Class (computer science)), which also determine their [types](https://en.wikipedia.org/wiki/Data_type" \o "Data type).

Eg Java, C++

**Functional programming** is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm" \o "Programming paradigm) where programs are constructed by [applying](https://en.wikipedia.org/wiki/Function_application" \o "Function application) and [composing](https://en.wikipedia.org/wiki/Function_composition_(computer_science)" \o "Function composition (computer science)) [functions](https://en.wikipedia.org/wiki/Function_(computer_science)" \o "Function (computer science)). It is a [declarative programming](https://en.wikipedia.org/wiki/Declarative_programming" \o "Declarative programming) paradigm in which function definitions are [trees](https://en.wikipedia.org/wiki/Tree_(data_structure)" \o "Tree (data structure)) of [expressions](https://en.wikipedia.org/wiki/Expression_(computer_science)" \o "Expression (computer science)) that map [values](https://en.wikipedia.org/wiki/Value_(computer_science)" \o "Value (computer science)) to other values, rather than a sequence of [imperative](https://en.wikipedia.org/wiki/Imperative_programming" \o "Imperative programming) [statements](https://en.wikipedia.org/wiki/Statement_(computer_science)" \o "Statement (computer science)) which update the [running state](https://en.wikipedia.org/wiki/State_(computer_science)" \o "State (computer science)) of the program.

In functional programming, functions are treated as [first-class citizens](https://en.wikipedia.org/wiki/First-class_citizen" \o "First-class citizen), meaning that they can be bound to names (including local [identifiers](https://en.wikipedia.org/wiki/Identifier_(computer_languages)" \o "Identifier (computer languages))), passed as [arguments](https://en.wikipedia.org/wiki/Parameter_(computer_programming)" \o "Parameter (computer programming)), and [returned](https://en.wikipedia.org/wiki/Return_value" \o "Return value) from other functions, just as any other [data type](https://en.wikipedia.org/wiki/Data_type" \o "Data type) can. This allows programs to be written in a [declarative](https://en.wikipedia.org/wiki/Declarative_programming" \o "Declarative programming) and [composable](https://en.wikipedia.org/wiki/Composability" \o "Composability) style, where small functions are combined in a [modular](https://en.wikipedia.org/wiki/Modular_programming" \o "Modular programming) manner

Eg haskell

| Imperative Programming | Declarative Programming |
| --- | --- |
| In this, programs specify how it is to be done. | In this, programs specify what is to be done. |
| It simply describes the control flow of computation. | It simply expresses the logic of computation. |
| Its main goal is to describe how to get it or accomplish it. | Its main goal is to describe the desired result without direct dictation on how to get it. |
| Its advantages include ease to learn and read, the notional model is simple to understand, etc. | Its advantages include effective code, which can be applied by using ways, easy extension, high level of abstraction, etc. |
| Its type includes procedural programming, object-oriented programming, parallel processing approach. | Its type includes logic programming and functional programming. |
| In this, the user is allowed to make decisions and commands to the compiler. | In this, a compiler is allowed to make decisions. |
| It has many side effects and includes mutable variables as compared to declarative programming. | It has no side effects and does not include any mutable variables as compared to imperative programming. |
| It gives full control to developers that are very important in low-level programming. | It may automate repetitive flow along with simplifying code structure. |
| Eg java, c | Eg haskell |
| x = [1, 2, 3, 4]  for i in range(len(x)):      x[i] = x[i]\*2  print(x) #[2, 4, 6, 8] | x = [1, 2, 3, 4]  y = list(map(lambda i : i\*2, x))  print(y) #[2, 4, 6, 8] |