**Understanding Compile-Time and Run-Time Polymorphism in Python**

In Python, polymorphism means that the **same function or operator** can behave differently based on context.

Polymorphism is mainly of two types:

**1. Compile-Time Polymorphism (also called Static Polymorphism)**

This kind of polymorphism is **resolved before the program runs**—typically during compilation. However, Python is a dynamically typed language and doesn't have traditional compile-time polymorphism like C++ or Java.

One common example that mimics compile-time polymorphism in Python is:

* **Operator Overloading**: Python allows you to define how operators like +, -, or \* behave for your custom classes. You do this by overriding special methods like \_\_add\_\_, \_\_mul\_\_, etc. Although Python resolves these methods at runtime, the behavior is conceptually similar to compile-time overloading in statically typed languages.

**2. Run-Time Polymorphism (also called Dynamic Polymorphism)**

This type of polymorphism is **resolved when the program is running**. Python heavily supports runtime polymorphism in several ways:

* **Method Overriding via Inheritance**: When a child class provides its own version of a method defined in the parent class, the correct method is chosen at runtime depending on the object type. This is classic runtime polymorphism.
* **Function Overloading using \*args or \*\*kwargs**: Python does not support real function overloading (i.e., multiple functions with the same name but different parameters). Instead, you can use default arguments, variable arguments (\*args), or keyword arguments (\*\*kwargs) to simulate overloading. The logic to differentiate between the types and number of arguments is handled at runtime.
* **Duck Typing**: Python relies on the principle of “if it walks like a duck and quacks like a duck, it's a duck.” In other words, Python doesn’t care about the actual type of the object but whether it has the required behavior (methods or properties). This is a clear example of dynamic or runtime polymorphism.

**What About @dispatch?**

The @dispatch decorator from the multipledispatch library allows function overloading based on argument types, which looks similar to compile-time overloading in languages like Java or C++. Here's a quick example:

from multipledispatch import dispatch

@dispatch(int, int)

def add(a, b):

return a + b

@dispatch(str, str)

def add(a, b):

return a + " " + b

While this **appears** to be compile-time polymorphism, in Python, it is **actually resolved at runtime**. The dispatch library inspects the argument types at the moment the function is called, and decides which version of the function to execute. So, it's still a form of **runtime polymorphism**, even though the syntax feels like static overloading.

**To sum up:**

* **Operator overloading** in Python is similar to compile-time polymorphism but technically resolved at runtime.
* **All other forms of polymorphism in Python, including @dispatch overloading, method overriding, use of \*args and \*\*kwargs, and duck typing, are examples of runtime polymorphism.**
* Python's flexibility and dynamic nature make it naturally suited to runtime polymorphism rather than compile-time.