**Wait! What are Pipelines in Python?**

[[Ayush Thakur](https://medium.com/@ayush-thakur02?source=post_page-----628f4b5021fd--------------------------------)](https://medium.com/@ayush-thakur02?source=post_page-----628f4b5021fd--------------------------------)

[Ayush Thakur](https://medium.com/@ayush-thakur02?source=post_page-----628f4b5021fd--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fc61c106e0290&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40ayush-thakur02%2Fwait-what-are-pipelines-in-python-628f4b5021fd&user=Ayush+Thakur&userId=c61c106e0290&source=post_page-c61c106e0290----628f4b5021fd---------------------post_header-----------)

5 min read

·

Nov 6

1.2K

21

If you are a Python developer, you might have heard of the term **pipeline**. But what exactly is a pipeline and why is it useful? In this blog post, we will explore the concept of pipelines in Python, how they work, and how they can help you write cleaner and more efficient code.

**What is a Pipeline?**

A pipeline is a way of organizing a series of operations or functions that process some data. The output of one operation becomes the input of the next one, and so on, until the final result is obtained. A pipeline can be visualized as a chain of pipes, where the data flows from one pipe to another, undergoing some transformation or manipulation along the way.

For example, suppose you have a list of numbers and you want to perform the following operations on them:

* Filter out the odd numbers
* Multiply each number by 10
* Add 5 to each number
* Calculate the average of the resulting numbers

One way to do this is to write a loop that iterates over the list and applies each operation one by one, storing the intermediate results in a new list. For example:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
filtered = []  
for n in numbers:  
 if n % 2 == 0:  
 filtered.append(n)  
multiplied = []  
for n in filtered:  
 multiplied.append(n \* 10)  
added = []  
for n in multiplied:  
 added.append(n + 5)  
total = 0  
count = 0  
for n in added:  
 total += n  
 count += 1  
average = total / count  
print(average)

This code works, but it is not very elegant or efficient. It creates three new lists, which take up memory and make the code less readable. It also requires writing four loops, which can be tedious and error-prone.

A better way to do this is to use a pipeline. A pipeline allows you to chain the operations together, without creating intermediate lists or loops. You can use the built-in map and filter functions, which take a function and an iterable as arguments, and return a new iterable that applies the function to each element of the original iterable. You can also use the sum and len functions, which calculate the sum and the length of an iterable, respectively. For example:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
average = sum(map(lambda n: n + 5, map(lambda n: n \* 10, filter(lambda n: n % 2 == 0, numbers)))) / len(numbers)  
print(average)

This code is much shorter and simpler than the previous one. It does not create any new lists or loops, and it is easy to see the flow of data from one operation to the next. However, it is still not very readable, as it uses nested map and filter calls, which can be confusing and hard to follow.

A more readable way to write a pipeline is to use the pipe function from the toolz library. The pipe function takes an initial value and a series of functions, and applies each function to the output of the previous one, returning the final result. For example:

from toolz import pipe  
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
average = pipe(numbers,   
 filter(lambda n: n % 2 == 0),   
 map(lambda n: n \* 10),   
 map(lambda n: n + 5),   
 lambda x: sum(x) / len(x))  
print(average)

This code is much more readable and elegant than the previous ones. It clearly shows the sequence of operations that are applied to the data, and it does not use any nested calls or intermediate lists. The pipe function makes the code look like a pipeline, where the data flows from one function to the next.

**Why Use Pipelines?**

Pipelines have several advantages over other ways of organizing code. Some of them are:

* **Readability**: Pipelines make the code more readable and understandable, as they show the flow of data from one operation to the next, without cluttering the code with intermediate variables or loops. Pipelines also make the code more modular and reusable, as each operation can be defined as a separate function that can be easily tested and reused in different contexts.
* **Efficiency**: Pipelines can improve the efficiency of the code, as they avoid creating intermediate lists or iterables that take up memory and slow down the execution. Pipelines also allow for lazy evaluation, which means that the operations are only executed when the data is needed, and not before. This can save time and resources, especially when dealing with large or infinite data sources.
* **Flexibility**: Pipelines can be easily modified or extended, as they allow for adding, removing, or changing the order of the operations without affecting the rest of the code. Pipelines also support different types of operations, such as filtering, mapping, reducing, aggregating, grouping, sorting, etc., which can be combined in various ways to achieve different results.

**How to Use Pipelines in Python?**

There are different ways to create and use pipelines in Python. Some of them are:

* **Using built-in functions**: Python provides several built-in functions that can be used to create pipelines, such as map, filter, reduce, zip, enumerate, sorted, reversed, etc. These functions take a function and an iterable as arguments, and return a new iterable that applies the function to each element of the original iterable.
* **Using list comprehensions**: List comprehensions are a concise and expressive way to create lists in Python, by using a syntax that resembles mathematical notation. List comprehensions can also be used to create pipelines, by applying a series of operations to each element of an iterable and collecting the results in a new list.
* **Using generator expressions**: Generator expressions are similar to list comprehensions, but they return a generator object instead of a list. A generator object is an iterable that produces the elements on demand, without storing them in memory. Generator expressions can also be used to create pipelines, by applying a series of operations to each element of an iterable and yielding the results as a generator.

I hope this blog post has helped you understand what pipelines are and how to use them in Python. If you have any questions or feedback, please leave a comment below. Thank you for reading!

Tags — #python #pipeline #programming #blog #tutorial