Roadmap For Scalable Generators

Hidden Barriers To Scalability

A generator function is only as scalable as the LEAST scalable line of code in its body.

You must learn to identify potential memory bottlenecks lurking even in simple lines of code...

And become skilled in the strategies to mitigate them.

Fine-Grained Advancing

Usually, the consumer of a generator object will be a `for` loop.

More advanced use cases require a finer-grained technique for stepping through the sequence.

This also proves useful for working with more generic iterators provided by outside libraries.

Fanning In And Out

The simple generator functions you have seen so far produce one output value for each input value.

But what if that mapping is not one-to-one?

Many useful generator functions diverge from that one-to-one pattern. Learning to code them is a valuable advanced skill.

Scalable Composability

At the highest level, you have an architectural design concept:

Scalable Composability.

This is about building a collection of generator functions, each robustly scalable on their own…

Yet are designed to flexibly connect together. Creating richer, yet equally scalable data-processing components.

Think of it as constructing a toolkit for expressively creating your program’s internal data pipelines.

Scalable Comprehensions

Generators share a surprising connection to Python comprehensions.

Learn how this works, and you can "shortcut" the process of creating simple but useful generator objects.

This lets you infuse your code with a greater degree of scalability. Once you learn this "trick", you will use it *all the time*.

Passing Data Into Generators

So far, everything you have seen is about generator objects *producing* data.

But it turns out, you can transmit data INTO a generator object.

This enables an entire new universe of idioms and patterns, that few ever learn about.