Python IV: Decorators and Generators

Raul P. Pelaez

April 28, 2026

Contents

1 Introduction

Today we will learn about two advanced Python features:

- Decorators
- Generators.

Decorators are a tool for modifying the behavior of functions, while generators provide a way to create iterators with a function that can yield multiple values one at a time. For instance, a decorator could be used to log function calls, or to time their execution. Generators, on the other hand, can be used to generate sequences of values lazily, which can be more memory-efficient than storing them in a list.

2 Decorators

Info

Decorators
A decorator is a callable that takes another function as an argument (the decorated function) and returns a new function. The new function usually extends or modifies the behavior of the original function. When a decorator is applied to a function, using the `@decorator_name` syntax, the original function is replaced by the new function returned by the decorator.
Example

```
from typing import Callable

def mydecorator(f: Callable) -> Callable:
    def inner():
        print("running inner")
        f()  # Optionally call the original function
    return inner

@mydecorator
def foo():
    print("This will print via the decorator")

foo()
```

Output

```
running inner
This will print via the decorator
```

A decorator typically wraps or replaces a function to modify its behavior.

Info

Decorate functions that have arguments
If the original function has arguments, the decorator function must accept them and pass them to the original function. This can be done using the `*args` and `**kwargs` syntax to capture all positional and keyword arguments.
Example

```python
from typing import Callable

def mydecorator(f: Callable) -> Callable:
    def inner(*args, **kwargs):
        print("running inner")
        f(*args, **kwargs)  # Pass the arguments to the original function
    return inner

@mydecorator
def foo(a, b):
    print(f"{a} and {b}")

foo(1, 2)
```

Output

```
running inner
1 and 2
```

- `*args` means: "accept any number of positional arguments and store them in a tuple", i.e. it captures all the positional arguments.
- `**kwargs` means: "accept any number of keyword arguments and store them in a dictionary", i.e. it captures all the named arguments.

In `some_function(1,2, c=3)`, `args=(1,2)` and `kwargs={'c':3}`.

Info

Variadic functions
A variadic function is a function that can accept a variable number of arguments. In Python, this is achieved using the `*args` and `**kwargs` syntax. The `*args` syntax is used to pass a variable number of positional arguments, while the `**kwargs` syntax is used to pass a variable number of keyword arguments.
Example

```python
def variadic_function(*args, **kwargs):
    print(args)
    print(kwargs)

variadic_function(1, 2, 3, a=4, b=5, some_keyword="value")
```

Output

```
(1, 2, 3)
{'a': 4, 'b': 5, 'some_keyword': 'value'}
```

Info

The `__name__` and `__doc__` dunder attributes
All Python objects (including functions) have a `__name__` attribute that stores their name. Similarly, functions can have a `__doc__` attribute that stores their docstring.
Example

```
def myfunc():
  """This is the docstring of myfunc"""
  pass

print(myfunc.__name__)
print(myfunc.__doc__)
```

Output

```
myfunc
This is the docstring of myfunc
```

Advice

Decorators are useful for adding functionality to functions without modifying their code. They are commonly used for tasks like logging, timing, and access control. Decorators can also be used to modify the behavior of functions based on certain conditions or to add functionality to multiple functions at once.
Another common use I have seen for decorators in the wild is for ”compilation” of functions. For instance, Numba uses decorators to compile Python functions to machine code, which can greatly speed up your code effortless. The `torch.compile` decorator in PyTorch is another example of this.

Advanced

Decorators run at import time

The code inside decorators is executed when the module is imported. In other words, the decorator functions are invoked as soon as the decorated functions are defined.

Example

Observe the order of print statements in this code:

```python
print("Script starts")

def register(f):
    print(f"Registering {f.__name__}")
    return f

@register
def f1():
    print("f1 runs")

def f2():
    print("f2 runs")

@register
def f3():
    print("f3 runs")

print("Calling functions for the first time:")
f1()
f2()
f3()
```

Output

```
Script starts
Registering f1
Registering f3
Calling functions for the first time:
f1 runs
f2 runs
f3 runs
```

Warning

Decorators overwrite the original function's metadata (name, docstring)
When a decorator is applied to a function, the original function's metadata (like its name, docstring, and signature) is replaced by the metadata of the decorator function.
Example

```
def mydecorator(f):
  def inner():
      print("running inner")
      f()
  return inner

@mydecorator
def foo():
  """This is the docstring of foo"""
  print("This will print via the decorator")

print(foo.__name__)  # Output: inner
print(foo.__doc__)   # Output: None
```

Output

```
inner
None
```

The `functools.wraps` decorator can be used to preserve the original function's metadata.

```
from functools import wraps

def mydecorator(f):
    @wraps(f) # This is the only change needed
    def inner():
        print("running inner")
        f()
    return inner

@mydecorator
def foo():
    """This is the docstring of foo"""
    print("This will print via the decorator")

print(foo.__name__)  # Output: foo
print(foo.__doc__)   # Output: This is the docstring of foo
```

Output

```
foo
This is the docstring of foo
```

2.1 The `functools` module

The `functools` module in Python provides higher-order functions that operate on or return other functions. It includes utilities for working with functions and callable objects. While this is a truly useful and vast module, we will for now just keep it in our radar. We have already seen one of its functions, `wraps`. Let me show you just another one: `functools.partial`.

The `functools.partial` function

This function is used to create a new function with some of the arguments of an existing function prefilled. Some objects/functions might require as input a function with fewer arguments than the original one. In this case, `functools.partial` can be used to create a new function with the desired number of arguments.

Example

```python
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)
print(square(5))  # Output: 25
print(cube(5))    # Output: 125
```

Output

```
25
125
```

Advanced

A real life example
Scipy has a method called `scipy.sparse.linalg.gmres`, which is used to solve linear systems. In particular, this function solves the linear system `Ax = b` using the Generalized Minimal Residual Method. This function requires a callable (another function) that computes the matrix-vector product `Ax`. Normally, we would have a function already defined that, given some information about the matrix `A`, computes the product `Ax`. However, this function might have more arguments than the ones required by `gmres`. In this case, we can use `functools.partial` to create a new function with the desired number of arguments.
Example

```
from scipy.sparse.linalg import gmres, LinearOperator
import numpy as np
from functools import partial
def matrix_vector_product(A, x):
   return A @ x

n = 10
A = np.random.rand(n,n)
b = np.random.rand(n)
# This is a function that only requires the vector x as argument
mdot = partial(matrix_vector_product, A)
op = LinearOperator((n, n), matvec=mdot)
x = gmres(op, b)
print(x)
```

Output

```
(array([-1.25812674,  0.62105063,  1.16655017,  1.01591448, -0.45657109,
        -0.75483407,  0.50359589, -0.2548077 ,  0.28192289,  0.15750996]), 0)
```

3 Generators

Generators provide a way to create iterators with a function that can yield multiple values one at a time. Unlike regular functions that return a single value and exit, a generator yields values and maintains its state between iterations. In other languages, generators are also known as *coroutines*.

Info

The `yield` keyword
The `yield` keyword is a "replacement" for the `return` keyword. If present in the body of a function, it turns the function into a generator. When a function `yield` s a value, the value is returned to the caller, but the function's state is saved. The next time the function is called, it resumes execution from where it left off, until another `yield` is encountered or the function ends.

Info

Generators

Generators are functions that `yield` values instead of `return`-ing them. When a generator is called, it returns an iterator object that can be used to iterate over the yielded values. Generators are useful for creating sequences of values lazily or "on the fly".

Example

```
import random
def even_elements(l):
    """ Only yield the even elements of a list """
    for elem in l:
        if elem % 2 == 0:
            yield elem

# A list of 10 random integers between 0 and 100
l = list(random.randint(0, 100) for _ in range(10))
for value in even_elements(l):
    print(value)
# Equivalent
print(list(even_elements(l)))
```

Output

```
40
26
26
86
[40, 26, 26, 86]
```

In the example above, the function `even_elements` is a generator that yields only the even elements of a list. The generator is used in a `for` loop to iterate over the values it yields. The sequence of events when the loop starts is as follows:

- The generator is called, and an iterator object is returned without executing the function body.
- The `for` loop calls `next()` on the iterator object, which executes the function body.
- When the function `even_elements` is called, it starts executing the function body until it reaches the `yield` statement. The value is yielded, and the function is paused.
- The body of the loop is then executed with the yielded value.
- This process is repeated until the function body is exhausted.

The second version of the code works in a similar way, but uses the `list` function, which consumes the iterator and returns a list with all the values.

Advanced

Iterators and Iterables

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. An iterable is an object that can return an iterator. All sequences (lists, tuples, strings, etc.) are iterable, but not all iterables are sequences.

In practical terms, an iterator is an object that implements the `__iter__` and `__next__` methods. The `__iter__` method returns the iterator object itself, and the `__next__` method returns the next value in the sequence. When there are no more values to return, the `__next__` method raises a `StopIteration` exception. This is the mechanism used by Python to iterate over objects, for instance in a `for` loop.

If `__iter__` is not defined, Python will try to use the `__getitem__` method, which is used to access elements by index.

The conceptual difference between iterators and generators is blurred in Python, as generators can be understood as a special type of iterator.

Advanced

The `yield from` syntax

Generators can also delegate to other generators using the `yield from` syntax, making it easier to compose complex iterators.

```
def generator1():
    yield 1
    yield 2

def generator2():
    yield from generator1()
    yield 3

for val in generator2():
    print(val)
```

Output

```
1
2
3
```

3.1 The `itertools` module

The `itertools` module provides a collection of tools for working with iterators. It includes functions for creating and manipulating iterators, such as `count`, `cycle`, `repeat`, and `chain`. This module is very useful when working with iterators and generators. Similar to `functools`, we will keep it in our radar for now. Let me show you just one of its functions: `itertools.chain`.

Example

Chain receives a series of iterables and returns an iterator that yields the elements of each iterable in sequence. First it yields all the elements of the first iterable, then all the elements of the second iterable, and so on.

```python
from itertools import chain
# Create an iterator that yields the first 3 natural numbers
natural_numbers = list(range(1, 4))
# Create an iterator that yields the first 3 natural numbers in reverse
reversed_natural_numbers = list(reversed(range(1, 4)))
print(list(chain(natural_numbers, reversed_natural_numbers)))
```

Output

```
[1, 2, 3, 3, 2, 1]
```

Advice

As you might have noticed, I am showing you a lot of examples of the interaction between iterators, generators and other Python features. The code becomes ever more succinct and expressive as you get used to these tools, i.e. we can say more with less code. I encourage you to practice with them, as they will make your code more readable and efficient. It is true, however, that the meaning of the code becomes harder to grasp if the syntax is not familiar to you. Try to play around with these examples; modify them, break them and fix them. This is the best way to learn.

4 Exercises

4.1 Registering function calls

Goal

Create a decorator that logs the arguments with which a function is called. For example:

```python
@register_call
def foo(a):
    return 2 * a

a = foo(2)
b = foo(4)
c = foo(12)
```

This should print:

```
foo was called with argument 2
foo was called with argument 4
foo was called with argument 12
```

hint

- You can use the `*args` and `**kwargs` syntax to capture the arguments of the function.
- Your decorator should be a function that receives another function as an argument and returns a new function.
- You can use the `__name__` attribute of the function to get its name.

Advanced milestone

Make your decorator include the time at which the function was called. For example:

```
@register_call
def foo(a):
    return 2 * a
foo(2)
```

Should print:

```
[2025-03-05 12:00:00] foo was called with argument 2
```

hint

- The function `datetime.now()` from the `datetime` module can be used to get the current time.
- You can use the `strftime` method to format the time as a string, e.g. `datetime.now().strftime("%Y-%m-%d %H:%M:%S")`.

Advanced milestone

Make your decorator also print the return value of the function. For example:

```
@register_call
def foo(a):
    return 2 * a
foo(2)
```

Should print:

```
[2025-03-05 12:00:00] foo was called with argument 2 and returned 4
```

4.2 Fibonacci generator

Goal

Create a generator that yields the Fibonacci sequence. The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, starting from 0 and 1. For example, the first 10 numbers in the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. The rule is:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$

The generator should take a single argument, `length`, and yield the Fibonacci sequence up to that length. For example, `fibonacci(10)` should yield the first 10 numbers in the Fibonacci sequence.
You can use the following code to test your generator:

```python
length = 10
print(list(fibonacci(length)))
# Equivalent
for value in fibonacci(length):
    print(value)
# Also equivalent
l = [i for i in fibonacci(length)]
print(l)
```

hint

- You will need to store the last two values of the sequence to calculate the next one as the "state" of the generator. You can simply use two variables and update them in each iteration.
- Your generator should have a `for` loop that `yield` s the values of the sequence and updates the state.

Milestone

Write a `test_fibonacci.py` file with tests for your Fibonacci generator using `pytest`. You can use the following code as a starting point:

```python
from fibonacci import fibonacci

def test_fibonacci():
    assert list(fibonacci(10)) == [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

hint

- The test code itself is pretty simple, what I want you to practice here is how you must structure the rest of the files in your project. You should have a `fibonacci.py` file with the generator, and a `test_fibonacci.py` file with the tests. You can use the `pytest` library to run the tests.
- You can create a whole project structure with a `src` folder for the code and a `tests` folder for the tests. But for something like this, simply having the two files in the same directory will suffice, as long as the file containing the generator is called `fibonacci.py`.