Published in TechToFreedom

Yang Zhou Follow
Jan 1 · 7 min read · Member-only · Listen

PYTHON

9 Python Built-In Decorators That Optimize Your Code Significantly

Do more by less: leverage the power of decorators

Image from Wallhaven

"Simple is better than complex."

The best Python feature that applies this philosophy from the "zen of Python" is the decorator.

Decorators can help you write less and simpler code to implement complex logic and reuse it everywhere.

More importantly, there are many awesome built-in Python decorators that make our lives much easier, since we can just use one line of code to add complicated functionalities to the existing functions or classes.

Talk is cheap. Let's see my hand-picked 9 decorators that will show you how elegant Python is.

1. @lru_cache: Speed Up Your Programs by Caching

The simplest way to speed up your Python functions with caching tricks is to use the `@lru_cache` decorator.

This decorator can be used to cache the results of a function, so that subsequent calls to the function with the same arguments will not be executed again.

It is especially helpful for functions that are computationally expensive or that are called frequently with the same arguments.

Let's see an intuitive example:

```
import time


def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


start_time = time.perf_counter()
print(fibonacci(30))
end_time = time.perf_counter()
print(f"The execution time: {end_time - start_time:.8f} seconds")
# The execution time: 0.18129450 seconds
```

The above program calculates the Nth Fibonacci number with a Python function. It's time-consuming cause when you calculate the `fibonacci(30)`, many previous Fibonacci numbers will be calculated many times during the recursion process.

Now, let's speed it up with the `@lru_cache` decorator:

```
from functools import lru_cache
import time


@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


start_time = time.perf_counter()
print(fibonacci(30))
end_time = time.perf_counter()
print(f"The execution time: {end_time - start_time:.8f} seconds")
# The execution time: 0.00002990 seconds
```

As the above code shows, after using the `@lru_cache` decorator, we can get the same result in `0.00002990` seconds, which is super faster than the previous `0.18129450` seconds.

The `@lru_cache` decorator has a `maxsize` parameter that specifies the maximum number of results to store in the cache. When the cache is full and a new result needs to be stored, the least recently used result is evicted from the cache to make room for the new one. This is called the **least recently used (LRU) strategy.**

By default, the `maxsize` is set to `128`. If it is set to `None`, as our example, the LRU features are disabled and the cache can grow without bound.

2. @total_ordering: A Class Decorator That Fills In Missing Ordering Methods

The `@total_ordering` decorator from the `functools` module is used to generate the missing comparison methods for a Python class based on the ones that are defined.

Here's an example:

```
from functools import total_ordering


@total_ordering
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __eq__(self, other):
        return self.grade == other.grade

    def __lt__(self, other):
        return self.grade < other.grade


student1 = Student("Alice", 85)
student2 = Student("Bob", 75)
student3 = Student("Charlie", 85)

print(student1 < student2)  # False
print(student1 > student2)  # True
print(student1 == student3)  # True
print(student1 <= student3) # True
print(student3 >= student2) # True
```

As the above code illustrated, there are no definitions for `__ge__`, `__gt__`, and `__le__` methods in the `Student` class. However, thanks to the `@total_ordering` decorator, the results of our comparisons between different instances are all correct.

The benefits of this decorator are obvious:

- It can make your code cleaner and save your time. Since you don't need to write all the comparison methods.
- Some old classes may not define enough comparison methods. It's safer to add the `@total_ordering` decorator to it for further usage.

3. @contextmanager: Make a Customized Context Manager

Python has a context manager mechanism to help you manage resources properly.

Mostly, we just need to use the `with` statements:

```python
with open("test.txt",'w') as f:
    f.write("Yang is writing!")
```

As the above code shows, we can open a file using the `with` statement so it will be closed automatically after being written. We don't need to call the `f.close()` function explicitly to close the file.

Sometimes, we need to define a customized context manager for some special requirements. In this case, the @contextmanager decorator is our friend.

For instance, the following code implements a simple customized context manager which can print corresponding information when a file is opening or closing.

```python
from contextlib import contextmanager

@contextmanager
def file_manager(filename, mode):
    print("The file is opening...")
    file = open(filename,mode)
    yield file
    print("The file is closing...")
    file.close()

with file_manager('test.txt', 'w') as f:
    f.write('Yang is writing!')
# The file is opening...
# The file is closing...
```

4. @property: Setting Up Getters and Setters for Python Classes

Getters and setters are important concepts in object-oriented programming(OOP).

For each instance variable of a class, a getter method returns its value while a setter method sets or updates its value. Given this, getters and setters are also known as accessors and mutators, respectively.

They are used **to protect your data from being accessed or modified directly and unexpectedly.**

Different OOP languages have different mechanisms to define getters and setters. In Python, we can simply use the `@property` decorator.

```python
class Student:
    def __init__(self):
        self._score = 0

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, s):
        if 0 <= s <= 100:
            self._score = s
        else:
            raise ValueError('The score must be between 0 ~ 100!')

Yang = Student()

Yang.score=99
print(Yang.score)
# 99

Yang.score = 999
# ValueError: The score must be between 0 ~ 100!
```

As the above example shows, the `score` variable cannot be set as 999, which is a meaningless number. Because we restricted its acceptable range inside the setter function using the `@property` decorator.

Without doubt, adding this setter can successfully avoid unexpected bugs or results.

5. @cached_property: Cache the Result of a Method as an Attribute

Python 3.8 introduced a new powerful decorator to the `functool` module — `@cached_property`. It can transform a method of a class into a property whose value is computed once and then cached as a normal attribute for the life of the instance.

Here's an example:

```python
from functools import cached_property


class Circle:
    def __init__(self, radius):
        self.radius = radius

    @cached_property
    def area(self):
        return 3.14 * self.radius ** 2


circle = Circle(10)
print(circle.area)
# prints 314.0
print(circle.area)
# returns the cached result (314.0) directly
```

In the above code, we decorated the `area` method through the `@cached_property`. So there are no repetitive calculations for `circle.area` of the same unchanged instance.

6. @classmethod: Define Class Methods in a Python Class

Inside a Python class, there are 3 possible types of methods:

- **Instance methods:** methods that are *bound to an instance*. They can access and modify the instance data. An instance method is called on an instance of the class, and it can access the instance data through the `self` parameter.
- **Class methods:** methods that are *bound to the class*. They can't modify the instance data. A class method is called on the class itself, and it receives the class as the first parameter, which is conventionally named `cls`.
- **Static methods:** methods that are *not bound to the instance or the class.*

The instance methods can be defined as normal Python functions as long as its first parameter is `self`. However, to define a class method, we need to use the `@classmethod` decorator.

To demonstrate, the following example defines a class method which can be used to get a `Circle` instance through a diameter:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @classmethod
    def from_diameter(cls, diameter):
        return cls(diameter / 2)

    @property
    def diameter(self):
        return self.radius * 2

    @diameter.setter
    def diameter(self, diameter):
        self.radius = diameter / 2


c = Circle.from_diameter(8)
print(c.radius)  # 4.0
print(c.diameter)  # 8.0
```

7. @staticmethod: Define Static Methods in a Python Class

As mentioned, static methods are not bound to an instance or a class. They are included in a class simply because they logically belong there.

Static methods are commonly used in utility classes that perform a group of related tasks, such as mathematical calculations. By organizing related functions into static methods within a class, our code will become more organized and easier to understand.

To define a static method, we just need to use the @staticmethod decorator. Let's see an example:

```
class Student:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self.nickname = None

    def set_nickname(self, name):
        self.nickname = name
```

```
print(Student.suitable_age(99)) # False
print(Student.suitable_age(27)) # True
print(Student('yang', 'zhou').suitable_age(27)) # True
```

8. @dataclass: Define Special Classes With Less Code

The `@dataclass` decorator (introduced in Python 3.7) can automatically generate several special methods for a class, such as `__init__`, `__repr__`, `__eq__`, `__lt__`, and so on.

Therefore, it can save us lots of time from writing these basic methods. If a class is primarily used to store data, the `@dataclass` decorator is our best friend.

To demonstrate, the following example just defines two data fields of a class named `Point`. Thanks to the `@dataclass` decorator, it's enough to be used:

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float

point = Point(1.0, 2.0)
print(point)
# Point(x=1.0, y=2.0)
```

9. @atexit.register: Register a Function To Be Executed Upon Normal Program Termination

The `@register` decorator from the `atexit` module can allow us to execute a function when the Python interpreter is exiting.

This decorator is very useful for performing final tasks, such as releasing resources or just saying goodbye! 👋

Here is an example:

```
import atexit

@atexit.register
def goodbye():
    print("Bye bye!")

print("Hello Yang!")
```

The outputs are:

```
Hello Yang!
Bye bye!
```

As the example shows, due to the use of the `@register` decorator, the terminal printed “Bye bye!” even if we didn’t call the `goodbye` function explicitly.

Thanks for reading. ❤️

Join Medium through my referral link to access millions of great articles:

Join Medium with my referral link - Yang Zhou

Read every story from Yang Zhou (and thousands of other writers on Medium). Your membership fee directly supports Yang…

yangzhou1993.medium.com

Reference:

7 Levels of Using Decorators in Python

Master the most magical feature of Python

medium.com

Programming Python Coding Technology Software Development