Member-only story

# Pytest Mocking Cheat Sheet

## Mocking made simple, patching made easy

Kay Jan Wong · Follow

Published in Towards Data Science · 11 min read · Mar 14, 2024



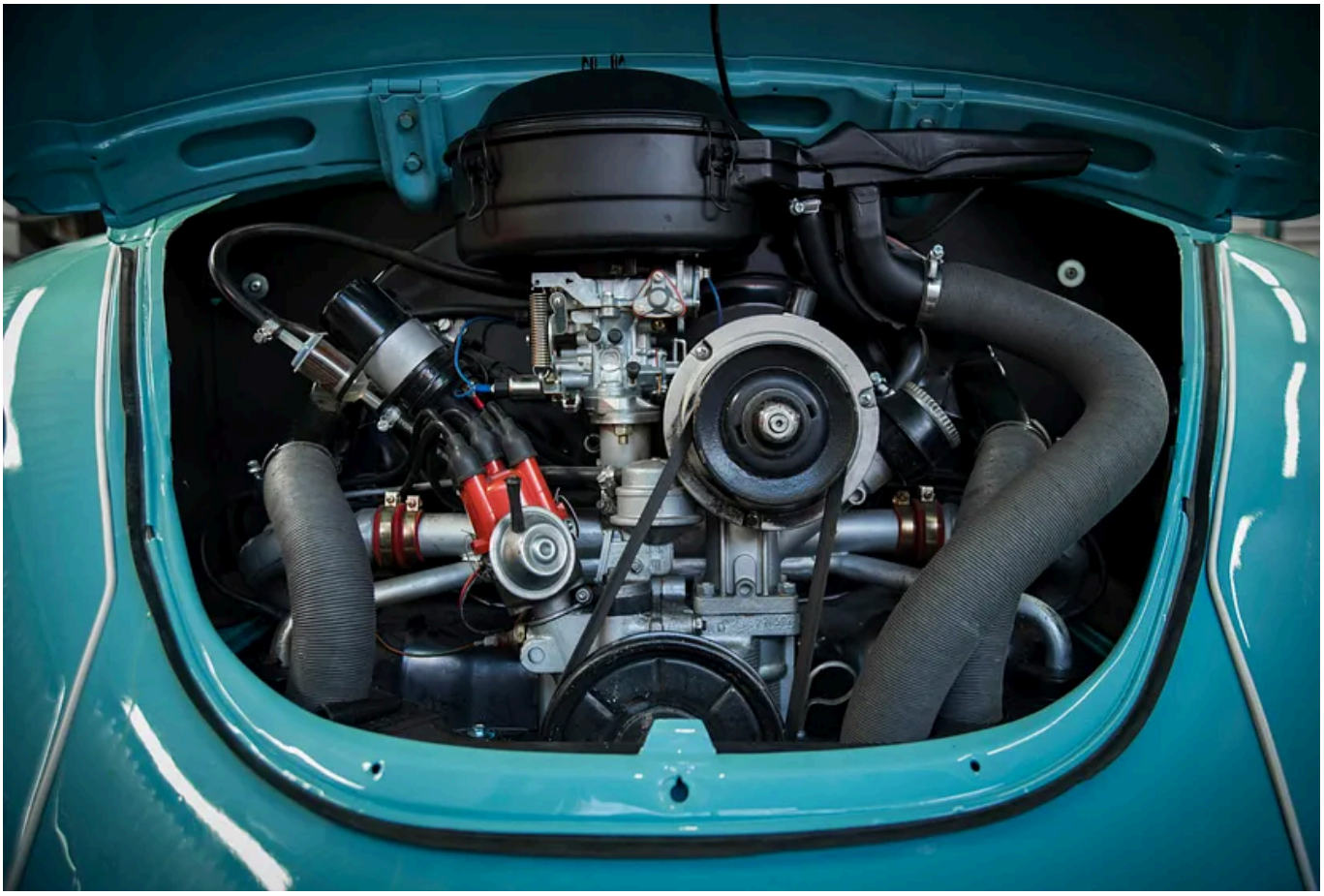


Photo by Habib Beaini on Unsplash

The previous article I wrote, which received over 100K views on Medium, provided a broad overview of `pytest`, covering topics on the setup, execution, and advanced features such as marking, mocking, and fixture usage. To quote from the previous article,

> *Mocking is used in unit tests to replace the return value of a function. It is useful to replace operations that should not be run in a testing environment, for instance, to replace operations that connect to a database and load data when the testing environment does not have the same data access.*

There is so much more to the topic of mocking. This article will show examples of how to mock constants, functions, an *initialized instance* of a class, private methods, magic methods, environment variables, external modules, fixtures, and even user inputs! Since mocking replaces actual values with inserted "fake" values, this article will also touch on **meaningful assertions** to use hand-in-hand with mocking.



## Table of Contents

- Tips and Tricks
- Examples (mock constants, function, class, and more)

## Purpose of Mocking

> *Why mock and what should you mock?*

Instead of listing the instances where you should implement mocking, it will be more intuitive to understand the purpose of unit tests and the use cases of mocking will naturally follow. Unit tests should be

- **Independent**: It should test the function and not its dependencies. With this consideration, the interface and wrappers should be tested, and external packages should be mocked if necessary
- **Fast**: Long-running functions or API calls can be mocked if they are not the subject of interest, especially if they also violate the independent criteria
- **Repeatable**: File read from a database should be mocked as the tests should not depend on the file contents, which may change over time as well

There are a few good practices to note

- If a **chunk of code needs to be replaced** with a final mocked value, consider refactoring your code since it is likely that the code is violating the Single Responsibility Principle

- If mocking is used for **replacing randomness** in a function, consider setting seed with `random.seed(0)` instead
- **Mock where it is used, not where it is defined.** This is especially important when mocking *initialized* class instances or *imported* modules since the item to mock is most likely already defined

## Mocking Packages

> *Comparing the trinity of `monkeypatch`, `pytest-mock`, and `mock`*

Three popular packages perform mocking, which have different installation steps and usage.

- **`monkeypatch`**: Requires installation with `pip install pytest`, used as a fixture called `monkeypatch`
- **`pytest-mock`**: Requires installation with `pip install pytest-mock`, used as a fixture called `mocker`
- **`mock`**: No installation required, used as an import such as `from unittest.mock import create_autospec, patch, Mock, MagicMock`

Below are examples of mocking a function using the different packages:

```
# Within src/example.py
def function_to_mock():
    return "original value"
```

```
def function_to_test():
    return function_to_mock()

# Within tests/src/test_example.py
from src.example import function_to_test
from unittest.mock import patch


def test_function_to_test_monkeypatch(monkeypatch):
    def mock_function_to_mock():
        return "mocked value"

    monkeypatch.setattr("src.example.function_to_mock", mock_function_to_mock)
    assert function_to_test() == "mocked value"


def test_function_to_test_pytest_mock(mocker):
    mocker.patch("src.example.function_to_mock", return_value="mocked value")
    assert function_to_test() == "mocked value"


@patch("src.example.function_to_mock")
def test_function_to_test_mock(mock_function_to_mock):
    mock_function_to_mock.return_value = "mocked value"
    assert function_to_test() == "mocked value"
```

In the example above, we are trying to test `function_to_test` which has a dependency on `function_to_mock`.

- `monkeypatch` defines another mocked function and performs one-to-one replacement
- `pytest-mock` sets what should be the replaced return value directly
- `mock` sets what should be the replaced return value directly

The detailed comparison is summarized well in this GitHub issue comment, which concluded that it comes down to personal preference. I prefer using `mock` as it is comprehensive and easy to understand and implement after

knowing a few tricks. In later sections, I will touch on the tricks and all of the code examples will be using `mock`.

## Implementations

> *Using context manager, decorator, or inline statement*

For `unittest.mock.patch`, it can be used as a decorator using `@patch`, as shown in the previous section, or as a context manager using the `with` keyword. Another method includes using it as an inline statement such as `mocker.patch()` in the previous section.

```
from unittest.mock import patch


# Decorator
@patch("src.example.function_to_mock")
def test_function_to_test_mock(mock_function_to_mock):
    mock_function_to_mock.return_value = "mocked value"
    assert function_to_test() == "mocked value"


# Context Manager
def test_function_to_test_mock_context_manager():
    with patch("src.example.function_to_mock", return_value="mocked value"):
        assert function_to_test() == "mocked value"
```

There is no *best* way to implement `patch`, but based on the different situations, decorators are a cleaner way to **mock multiple functions**, although bear in mind that the order of decorators matters. Decorators are

executed in last-to-first order and it matters when parameters need to be mocked in the correct order. Context managers are good for cases when a **fixture needs to be mocked** since decorators sit outside the function and are unable to access the fixture.

## Assertions for Mocking

> *Meaningful assertions to use when inputs are mocked*

Asserting the final output of the function, as shown in the previous section, does not *necessarily* mean that the tested function is using the mocked value instead of the original value. More definite assertions can be made to ensure that the mocked function was called and what arguments it was called with.

### Assertions for mocked function being called

In the order of most generic to most specific,

- **`.assert_called()`** : asserts that it was ever called
- **`.assert_called_once()`** : asserts that it was called once
- **`.call_count`** : get the number of times it was called

This is not recommended to use but is included for completeness.

- **`.assert_not_called()`** : asserts that it was not called (dangerous to use)

### Assertions for arguments to mocked function

In the order of most generic to most specific,

- **`.assert_any_call`**`(*args, **kwargs)` : asserts that it was ever called with the argument
- **`.assert_called_with`**`(*args, **kwargs)` : asserts that it was called with the argument, if it was called multiple times, this checks the last argument it was called with
- **`.assert_called_once_with`**`(*args, **kwargs)` : asserts that it was called once with the argument

### Assertion for arguments to mocked function over multiple calls

In the order of most generic to most specific,

- **`.assert_has_calls`**`(calls)` : asserts that certain calls exist
- **`.call_args_list`** : get the list of calls, able to check all calls

Below are examples of how the assertions can be used,

```
# Within src/example.py
def function_to_mock(num: int):
    return num * 2


def function_to_test(num1: int, num2: int):
    return function_to_mock(num=num1) + function_to_mock(num=num2)

# Within tests/src/test_example.py
from src.example import function_to_test
from unittest.mock import call, patch


@patch("src.example.function_to_mock")
def test_sample_func_mock(mock_function_to_mock):
    mock_function_to_mock.return_value = 4

    assert function_to_test(1, 2) == 8
    assert mock_function_to_mock.call_count == 2
```

```
    mock_function_to_mock.assert_called_with(num=2)
    assert mock_function_to_mock.call_args_list == [call(num=1), call(num=2)]
```

## Tips and Tricks

> *Master mocking!*

Below are some of the initial confusion I had when performing mocking, and some tips and tricks that helped me implement mocking in multiple situations.

### Returning different values across multiple calls

> *In some cases, the mocked function could be called multiple times and a different mocked value needs to be returned across the multiple calls*

For example, a function that loads data might be called multiple times to load different datasets. In this case, `.side_effect = [mock_value1, mock_value2]` can be used instead of `.return_value = mock_value`. This is generic across `monkeypatch`, `pytest-mock`, and `mock`.

### `Mock` **vs.** `MagicMock`

> *Within* `unittest.mock` *there are* ***`Mock`*** *and* ***`MagicMock`*** *classes*

`MagicMock` is the default option used when implementing mocking. In terms of differences, the short answer is that `MagicMock` is a subclass of `Mock`, and

hence has all the features of `Mock` with added mocking for magic methods. In simple mocking cases, there is not much difference between `Mock` and `MagicMock`, unless magic methods are called on the mocked object.

```
from unittest.mock import Mock, MagicMock

mock_obj = Mock()
mock_obj[1]
# TypeError: 'Mock' object is not subscriptable

magicmock_obj = MagicMock()
magicmock_obj[1]
# <MagicMock name='mock.__getitem__()' id='1839395120288'>
```

### Use `auto_spec` if possible

> *While `MagicMock` is widely used, it can accept any argument and attribute which can be a drawback if you want the mocked object to have the same function signature or class attributes*

In such cases, `auto_spec` solves this drawback.

```
# Within src/example.py
def function_to_mock(num: int):
    return num * 2

# Within tests/src/test_example.py
@patch("src.example.function_to_mock")
def test_sample_func_mock(mock_function_to_mock):
    mock_function_to_mock.return_value = 4
    assert mock_function_to_mock() == 4  # this will pass although its wrong


@patch("src.example.function_to_mock", autospec=True)
def test_sample_func_mock(mock_function_to_mock):
```

```
mock_function_to_mock.return_value = 4
assert mock_function_to_mock() == 4  # this will fail
# TypeError: missing a required argument: 'num'
```

`patch` **vs.** `patch.object`

> *When implementing mocking, it is important to know what you are patching*

If the item to mock is not imported, we can patch it with a string input documenting the full path to the item. However, if the item to mock is imported or initialized, such as a fixture, we should patch the object instead with the object to mock and the attribute to mock.

Both methods can be implemented as a decorator or context manager, and the convention is as such:

- `patch("full_path.folder.file.ClassToMock")`
- `patch.object(class_to_mock, "attribute_to_mock")`

We will see when and how to use the different types of patching in different scenarios in the next section.

## Examples

### 1. Mock constants, environment variables, and external modules

In the example below, we will define global constants outside the function and local constants within the function. The constants can be hardcoded or

derived from environment variables.

```
# Within src/example.py
import os

GLOBAL_CONSTANT = 2
GLOBAL_ENV_VAR = os.getenv("GLOBAL_ENV_VAR", 3)


def function_to_test():
    LOCAL_ENV_VAR = os.getenv("LOCAL_ENV_VAR", 4)
    return GLOBAL_CONSTANT * GLOBAL_ENV_VAR * LOCAL_ENV_VAR

# Within tests/src/test_example.py
from src.example import function_to_test
from unittest.mock import patch


@patch("src.example.GLOBAL_CONSTANT", 1)
@patch("src.example.GLOBAL_ENV_VAR", 1)
@patch("src.example.os.getenv", return_value=1)
def test_sample_func_mock(mock_getenv):
    assert function_to_test() == 1
```

For constants or environment variables defined outside the function to test, they can be patched directly with `@patch("folder.file.CONSTANT")`. Take note that the patched item is a constant and does not need to specify `.return_value` nor have the mocked item appear as an argument.

Unfortunately, constants within a function to test cannot be patched, unless they are derived from another function, in this case, the external module `os.getenv`. Take note that instead of patching `os.getenv` directly, we should patch `src.example.os.getenv` which is the imported module within the file. This follows from the rule to **mock where it is used, not where it is defined.**

## 2. Mock functions

From the good practices mentioned, it could be a signal of code smell when there are multiple items to mock. In this example, we abstract out the computation to another function `function_to_mock`. From there, we can **mock the abstracted function** directly with the regular `patch` implementation.

```
# Within src/example.py
import os

GLOBAL_CONSTANT = 2
GLOBAL_ENV_VAR = os.getenv("GLOBAL_ENV_VAR", 3)

def function_to_mock():
    LOCAL_ENV_VAR = os.getenv("LOCAL_ENV_VAR", 4)
    return GLOBAL_CONSTANT * GLOBAL_ENV_VAR * LOCAL_ENV_VAR


def function_to_test():
    return function_to_mock()

# Within tests/src/test_example.py
from src.example import function_to_test
from unittest.mock import patch


@patch("src.example.function_to_mock", return_value=1)
def test_sample_func_mock(mock_function_to_mock):
    assert function_to_test() == 1
```

### 3. Mock imported functions

In a more complex folder structure, the function to mock can be from another file. This introduces complexity as the dependent function is implicitly imported when we import the function to test. The code in the previous section is still valid as the rule remains where we should **mock where it is used, not where it is defined.**

```
# Within src/another_file.py
import os

GLOBAL_CONSTANT = 2
GLOBAL_ENV_VAR = os.getenv("GLOBAL_ENV_VAR", 3)

def function_to_mock():
    LOCAL_ENV_VAR = os.getenv("LOCAL_ENV_VAR", 4)
    return GLOBAL_CONSTANT * GLOBAL_ENV_VAR * LOCAL_ENV_VAR

# Within src/example.py
from src.another_file import function_to_mock


def function_to_test():
    return function_to_mock()

# Within tests/src/test_example.py
from src.example import function_to_test  # implicitly imported function_to_mock
from unittest.mock import patch

# DO NOT DO THIS
@patch("src.another_file.function_to_mock", return_value=1)
def test_sample_func_mock(mock_function_to_mock):
    assert function_to_test() == 24  # mocking did not work
```

## 4. Mock an initialized instance of a class

We can mock the class variable, property, method, private method, and even magic method of a *class instance*. In the example below, we define a class `ClassToMock` and **mock the class when it is used in another function.**

```
# Within src/example.py
class ClassToMock:
    def __init__(self):
        self.variable_to_mock = "original variable"
        self._property_to_mock = "original property"

    @property
    def property_to_mock(self):
```

```
            return self._property_to_mock

        def method_to_mock(self):
            return "original method"

        def call_private_method(self):
            return self.__private_method_to_mock()

        def __private_method_to_mock(self):
            return "original private method"

        def __str__(self):
            return "original magic method"


    def get_class_variable():
        class_to_mock = ClassToMock()
        return class_to_mock.variable_to_mock  # "original variable"


    def get_class_property():
        class_to_mock = ClassToMock()
        return class_to_mock.property_to_mock  # "original property"


    def get_class_method():
        class_to_mock = ClassToMock()
        return class_to_mock.method_to_mock()  # "original method"


    def get_class_private_method():
        class_to_mock = ClassToMock()
        return class_to_mock.call_private_method()  # "original private method"


    def get_class_magic_method():
        class_to_mock = ClassToMock()
        return str(class_to_mock)  # "original magic method"
```

When mocking a class variable and property, we first mock the class and set the return value of the mocked class for the variable and property.

```
# Within tests/src/test_example.py
from src.example import get_class_variable, get_class_property
from unittest.mock import patch


@patch("src.example.ClassToMock")
def test_get_class_variable(mock_class):
    mock_class.return_value.variable_to_mock = "mocked variable"
    assert get_class_variable() == "mocked variable"


@patch("src.example.ClassToMock")
def test_get_class_property(mock_class):
    mock_class.return_value.property_to_mock = "mocked property"
    assert get_class_property() == "mocked property"
```

Mocking a class method and its magic method are similar to mocking functions in the section above. Due to how Python handles private methods, mocking private methods has a slightly different full path as shown below.

```
# Within tests/src/test_example.py
from src.example import (
    get_class_method,
    get_class_private_method,
    get_class_magic_method,
)
from unittest.mock import patch


@patch("src.example.ClassToMock.method_to_mock", return_value="mocked method")
def test_get_class_method(mock_class_method):
    assert get_class_method() == "mocked method"


@patch("src.example.ClassToMock.__str__", return_value="mocked magic method")
def test_get_class_magic_method(mock_magic_method):
    assert get_class_magic_method() == "mocked magic method"


@patch(
```

```
    "src.example.ClassToMock._ClassToMock__private_method_to_mock",
    return_value="mocked private method",
)
def test_get_class_private_method(mock_private_method):
    assert get_class_private_method() == "mocked private method"
```

## 5. Mock fixtures

In the earlier two parts, we mock functions and classes where they are used with `patch`. For fixtures, we need to **patch the fixture directly**, and this requires `patch.object`. In addition, the fixture is added as an argument to the test function, therefore patching has to be done either inline or using a context manager instead.

In the code snippet below, we define the class to mock and initialize the class within the fixture.

```
# Within src/example.py
class ClassToMock:
    def __init__(self):
        self.variable_to_mock = "original variable"
        self._property_to_mock = "original property"

    def get_variable_to_mock(self):
        return self.variable_to_mock

    @property
    def property_to_mock(self):
        return self._property_to_mock

    def method_to_mock(self):
        return "original method"

    def call_private_method(self):
        return self.__private_method_to_mock()

    def __private_method_to_mock(self):
        return "original private method"
```

```
    def __str__(self):
        return "original magic method"

# Within tests/conftest.py (Define fixture)
import pytest

from src.example import ClassToMock


@pytest.fixture
def class_to_mock():
    return ClassToMock()
```

After patching the variable and property of the class fixture, I will assert that other methods have not changed and are working as intended. This is because we do not want to accidentally convert the whole fixture into a `MagicMock` object.

```
# Within tests/src/test_example.py
from unittest.mock import patch


def test_class_as_fixture_mock_variable(class_to_mock):
    with patch.object(class_to_mock, "variable_to_mock", "mocked variable"):
        # Access variable directly
        assert class_to_mock.variable_to_mock == "mocked variable"
        # Function that accesses variable
        assert class_to_mock.get_variable_to_mock() == "mocked variable"
        # Check if other properties are not affected
        assert class_to_mock.property_to_mock == "original property"


def test_class_as_fixture_mock_property(class_to_mock):
    with patch.object(class_to_mock, "property_to_mock", "mocked property"):
        assert class_to_mock.property_to_mock == "mocked property"
        assert class_to_mock.variable_to_mock == "original variable"
```

Patching the fixture class method, magic method, and private method follow the same principles.

```
# Within tests/src/test_example.py
from unittest.mock import patch, MagicMock


def test_class_as_fixture_mock_method(class_to_mock):
    with patch.object(class_to_mock, "method_to_mock", return_value="mocked meth
        assert class_to_mock.method_to_mock() == "mocked method"
        assert class_to_mock.variable_to_mock == "original variable"


def test_class_as_fixture_mock_magic_method(class_to_mock):
    class_to_mock.__str__ = MagicMock(return_value="mocked magic method")
    assert class_to_mock.__str__() == "mocked magic method"
    assert str(class_to_mock) == "original magic method"  # unable to replace th
    assert class_to_mock.variable_to_mock == "original variable"


def test_class_as_fixture_mock_private_method(class_to_mock):
    with patch.object(class_to_mock, "_ClassToMock__private_method_to_mock") as
        mock_private_method.return_value = "mocked private method"
        assert class_to_mock.call_private_method() == "mocked private method"
        assert class_to_mock.variable_to_mock == "original variable"
```

## 6. Mock user inputs

In interactive programs where users are supposed to key in inputs, we can **mock the user inputs** and assert the program code output. This usage is closer to integration testing than unit testing.

```
# Within src/example.py
def run_program():
    user_input_name = input("What is your name?")
    user_input_hometown = input("Where are you from?")
    return f"Hello {user_input_name} from {user_input_hometown}!"
```

```python
# Within tests/src/test_example.py
from src.example import run_program
from unittest.mock import patch


def test_run_program():
    with patch("builtins.input", side_effect=["John", "New York"]):
        assert run_program() == "Hello John from New York!"
```

Mocking used to feel very daunting — what to mock, how to mock, what to use, where to begin, why does it not work? After mocking various data types and noticing the pattern, this made it easier as if there is a mocking cheat sheet. Hope you have learnt more and noticed certain patterns in mocking as well, happy mocking!

## Related Links

- `unittest.mock` Documentation: https://docs.python.org/3/library/unittest.mock.html

### References

- Comparing `monkeypatch`, `mock`, and `pytest-mock`: https://github.com/pytest-dev/pytest/issues/4576#issuecomment-449864333
- Mock vs. MagicMock: https://stackoverflow.com/questions/17181687/mock-vs-magicmock

Mocking Unittest Pytest Python Tips And Tricks

## Written by Kay Jan Wong

Follow

2.1K Followers · Writer for Towards Data Science

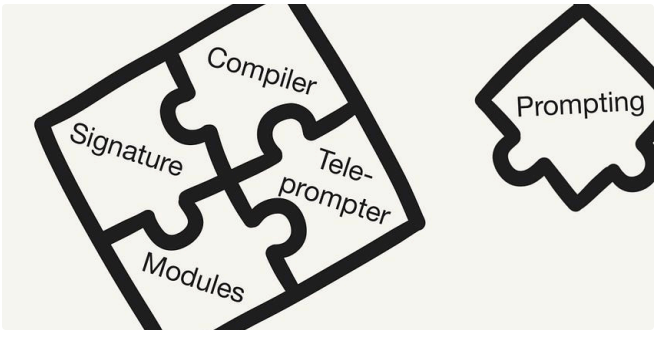
Data Scientist, Machine Learning Engineer, Software Developer, Programmer | Someone who loves coding, and believes coding should make our lives easier

### More from Kay Jan Wong and Towards Data Science

Kay Jan Wong in Towards Data Science

**Python Tree Implementation with BigTree**

Integrating trees with Python lists, dictionaries, and pandas DataFrames




Leonie Monigatti in Towards Data Science

**Intro to DSPy: Goodbye Prompting, Hello Programming!**

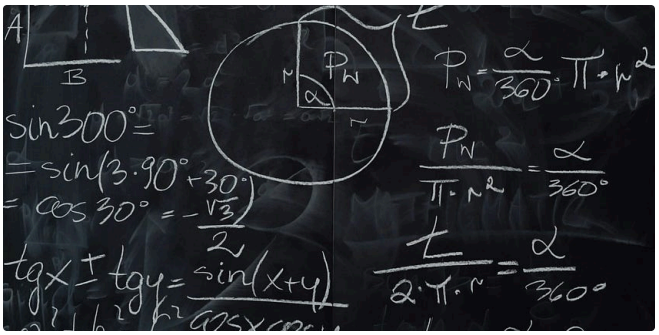
How the DSPy framework solves the fragility problem in LLM-based applications by...

11 min read · Nov 8, 2022

117 1

✦ · 13 min read · Feb 27, 2024

3.5K 10



Egor Howell in Towards Data Science

## How to Learn the Math Needed for Data Science

A breakdown of the three fundamental math fields required for data science: statistics,...

✦ · 8 min read · Mar 4, 2024

1.6K 9

Kay Jan Wong in Towards Data Science

## Multithreading and Multiprocessing in 10 Minutes

Multitasking made easy with Python examples

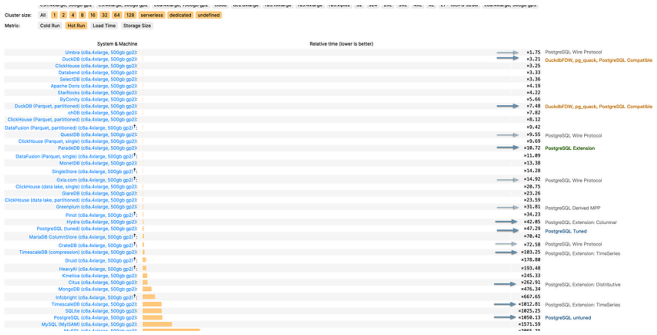
✦ · 4 min read · Mar 24, 2022

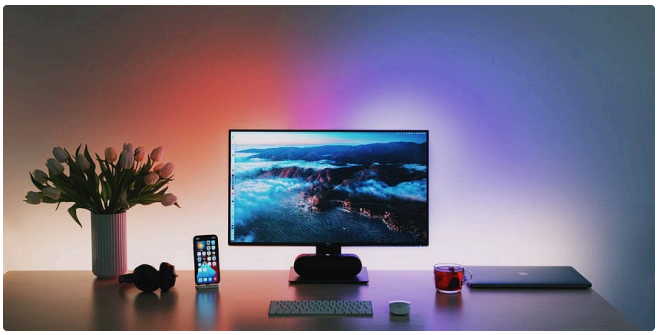
271 1

See all from Kay Jan Wong

See all from Towards Data Science

# Recommended from Medium





Vonng

## Postgres is eating the database world

PostgreSQL isn't just a simple relational database; it's a data management framewor...

11 min read · Mar 15, 2024

1.5K 14

Arunn Thevapalan in Towards Data Science

## Building Your First Desktop Application using PySide6 [A Dat...

Surprise, surprise. It's not as hard as I thought it would be.

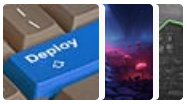
· 14 min read · Mar 16, 2024

976 7

### Lists

**Coding & Development**

11 stories · 522 saves



**Predictive Modeling w/ Python**

20 stories · 1034 saves

**Practical Guides to Machine Learning**

10 stories · 1236 saves

**ChatGPT prompts**

47 stories · 1329 saves


```
*__, a, b, *_ = [1, 2, 3, 4, 5, 6]
print(__, _)
```

What does this print?

- A) Syntax error
- B) `[1] [4, 5, 6]`
- C) `[1, 2] [5, 6]`
- D) `[1, 2, 3] [6]`
- E) `<generator object <genexpr> at 0x1003847c0>`

Builescu Daniel in Python in Plain English

## Your Python Code is a Secret Mess (and Interviewers Can Smell It)

Level up your Python: Code that gets you hired.

· 7 min read · Mar 15, 2024

441 6

Liu Zuo Lin

## You're Decent At Python If You Can Answer These 7 Questions...

# No cheating pls!!

· 6 min read · Mar 6, 2024

1.3K 11


Serhii Olendarenko

## Why C++ is a bad language

Or at least why is it considered to be so?

12 min read · Mar 10, 2024

196 4


DavidW (skyDragon) in overcast blog

## 13 Docker Tricks You Didn't Know

Docker has become an indispensable tool in the development, testing, and deployment...

22 min read · Mar 14, 2024

775 5

See more recommendations