✨ Member-only story

6 Pythonic Ways to Replace if-else Statements

Avoid if-else the Pythonic Ways

Bobby · Follow

Published in Level Up Coding
6 min read · Feb 26

Listen Share

If-else statements are a fundamental control structure in programming. However, code that relies heavily on if-else statements is less readable and also less maintainable. Thankfully, with Python, we can easily avoid if-else statements. In this blog post, we will explore six Pythonic ways for avoiding if-else statements.

Photo by Kenny Eliason on Unsplash

Should if-elif-else be Completely Avoided?

Short answer: **No.** When the conditional logic is **simple**, we can use `if-else`.

```
def number_classification(number):
    if number > 0:
        return "Positive"
    elif number < 0:
        return "Negative"
    else:
        return "Zero"
```

In this example, we want to classify if a number is positive, negative, or zero. The logic is simple and thus, it is fine to use `if-elif-else`. By the way, the `else` block here can be gotten rid of.

1. Unnecessary else

```
def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False
```

The code above checks if a number is even. Nothing is wrong with its logic. However, the `else` block is not necessary and can be easily gotten rid of.

```
def is_even(number):
    if number % 2 == 0:
        return True
    return False
```

In many cases, the `else` block is not needed at all. If a certain condition is met, just return something imm

Another case w assign value to variables. Assigning a default value can simplify your code a lot.

```
# No, we can do better than this
if condition:
    number = 1
else:
    number = 2


# Just use a default vlue
number = 2
if condition:
    number = 1
```

2. Value Assignments

It is quite common for new programmers to use `if-else` to assign a new value to a variable depending on the given input.

```
def classify_temperature(temperature):
    message = ''
    if temperature >= 26:
        message = "Hot"
    elif temperature >= 15:
        message = "Warm"
    else:
        message = "Cold"
    return message
```

In this example, we have a function that takes a temperature in Celcius and classifies whether the temperature is hot, warm, or cold. This function is very simple and readable but it can still be improved. There is no need for the `elif` and `else` block.

```
def classify_temperature(temperature):
    if temperature >= 26:
        return "Hot"
    if temperature >= 15:
```

```
        return "Warm"
    return
```


3. Guard Clauses

Guard clauses are also known as early returns. Guard clauses are a way to simplify control flow in a function or method by returning or raising an exception as soon as an **early exit condition** is met. This can help eliminate the need for `else` statements and make the code more readable, maintainable, and also testable. In my opinion, it makes no sense to let a function continue if it is provided with invalid inputs.

The early return code looks like this:

```
def my_function():
    if not condition:
        return  # or raise an Exception
    # Do something

def my_function_reverse_condition():
    if condition:
        # Do something
    return # or raise an Exception
```

In the example below, the version with guard clauses is more readable as each condition is checked and handled separately. This version also does not require several `else` .

```
# Using guard clauses
def validate_input(input_list):
    if not isinstance(input_list, list):
        raise TypeError("Input must be a list.")
    if not input_list:
        raise ValueError("Input cannot be an empty list.")
    if not all(isinstance(item, float) for item in input_list):
        raise ValueError("All items in the list must be floats.")
    return True


# Using if-else statements
def validate_input(input_list):
    if isinstance(input_list, list):
        if input_list:
            if all(isinstance(item, float) for item in input_list):
                return True
            else:
                raise ValueError("All items in the list must be floats.")
        else:
```

```
            raise ValueError("Input cannot be an empty list.")
    else:
        rai
```


4. Dictionary

Python dictionary is a real painkiller when we need to work with a large number of `if-else` statements. With a Python dictionary, we can map each input value to its corresponding action or value.

```
# lots of if-elif
def determine_favorite_fruit(color):
    if color == "red":
        return "Strawberry"
    elif color == "yellow":
        return "Banana"
    elif color == "green":
        return "Honeydew"
    return "unknown"


# Dictionary is used
def determine_favorite_fruit_with_dict(color):
    color_to_fruit = {
        "red": "Strawberry",
        "yellow": "Banana",
        "green": "Honeydew",
    }
    return color_to_fruit.get(color, "unknown")
```

Using Python dictionaries greatly improves our code readability. Additionally, it is also more flexible. There are cases when we need to add more cases to our code. With `if-elif-else` approach, we need to add another block of code to the method. With the dictionary approach, we just need to modify our dictionary without affecting the rest of the code. Another benefit is improved efficiency because, in Python, dictionary lookups are usually faster than iterating through a series of `if-else` conditions.

5. Match

Starting from Python 3.10, we can use `match` to perform pattern matching. Thanks to this feature, we can say goodbye to long chains of `if-elif` .

```
match value:
    case pattern_1:
        # code to execute if value matches pattern_1
```

```
    case pattern_2:
        # c
    ...
    case pa
        # c
    case _:
        # code to execute if value matches pattern_n
```


The _ (underscore) is a wildcard pattern that matches any value.

```
def use_if_else(value):
    if value == 1:
        print("Value is 1")
    elif value == 2:
        print("Value is 2")
    elif value == 3:
        print("Value is 3")
    else:
        print("Value is something else")


def use_match(value):
    match value:
        case 1:
            print("Value is 1")
        case 2:
            print("Value is 2")
        case 3:
            print("Value is 3")
        case _:
            print("Value is something else")
```

In comparison to the `if-elif` method, `match` makes the code more readable.

6. Strategy Pattern

The strategy pattern is a design pattern in software engineering that allows the behavior of a class to be selected at runtime.

Example: Our website allows customers to use different payment methods, such as Credit cards and Paypal. Below is an example of **not** using the strategy pattern. Quite many `if-elif-else` are used:

```
def pay_with_credit_card(amount):
    print(f"Paying {amount} using credit card.")
```

```
def pay_with_paypal(amount):
    print(f

def pay(amo
    if meth
        pay_with_credit_card(amount)
    elif method == "paypal":
        pay_with_paypal(amount)
    else:
        raise ValueError("Unsupported payment method")
```

If `stripe` is also allowed as a payment method, we must modify the `pay` method. Even more `if-elif-else` . Another reason why this practice is bad: it violates the Open-Closed Principle (SOLID Principles).

```
def pay_with_credit_card(amount):
    print(f"Paying {amount} using credit card.")

def pay_with_paypal(amount):
    print(f"Paying {amount} using PayPal.")

def pay_with_stripe(amount):
    print(f"Paying {amount} using Stripe.")

def pay(amount, method):
    if method == "credit_card":
        pay_with_credit_card(amount)
    elif method == "paypal":
        pay_with_paypal(amount)
    elif method == "stripe":
        pay_with_stripe(amount)
    else:
        raise ValueError("Unsupported payment method")
```

We can achieve better maintainability while limiting `if-elif-else` with the Strategy Pattern. When the `pay` method is called on the context `PaymentContext` , it delegates the payment to the current payment strategy.

```
from abc import ABC, abstractmethod


# Define the abstract Strategy class
class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount):
        pass
```

```
# Define th
class Credi
    def pay
        print(f"Paying {amount} using credit card.")


class PayPalPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paying {amount} using PayPal.")


class StripePayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paying {amount} using Stripe.")


# Define the Context class that will use the Strategy pattern
class PaymentContext:
    def __init__(self, payment_strategy: PaymentStrategy):
        self.payment_strategy = payment_strategy

    def set_payment_strategy(self, payment_strategy: PaymentStrategy):
        self.payment_strategy = payment_strategy

    def pay(self, amount):
        self.payment_strategy.pay(amount)


# Example usage
payment_context = PaymentContext(CreditCardPayment())
payment_context.pay(100)  # Output: Paying 100 using credit card.

payment_context.set_payment_strategy(PayPalPayment())
payment_context.pay(50)  # Output: Paying 50 using PayPal.

payment_context.set_payment_strategy(StripePayment())
payment_context.pay(150)  # Output: Paying 150 using Stripe.
```


🔔 Want more articles like this? Sign up here.

Thanks for reading. I hope these 6 tricks would be useful to you.

Thanks for being a part of our community! Before you go:

- 👏 Clap for

- 📰 View more content in the Level Up Coding publication
- 💰 Free coding interview course ⇒ View Course
- 🔔 Follow us: Twitter | LinkedIn | Newsletter

🚀👉 **Join the Level Up talent collective and find an amazing job**

Python Coding Programming Learn To Code Python Programming

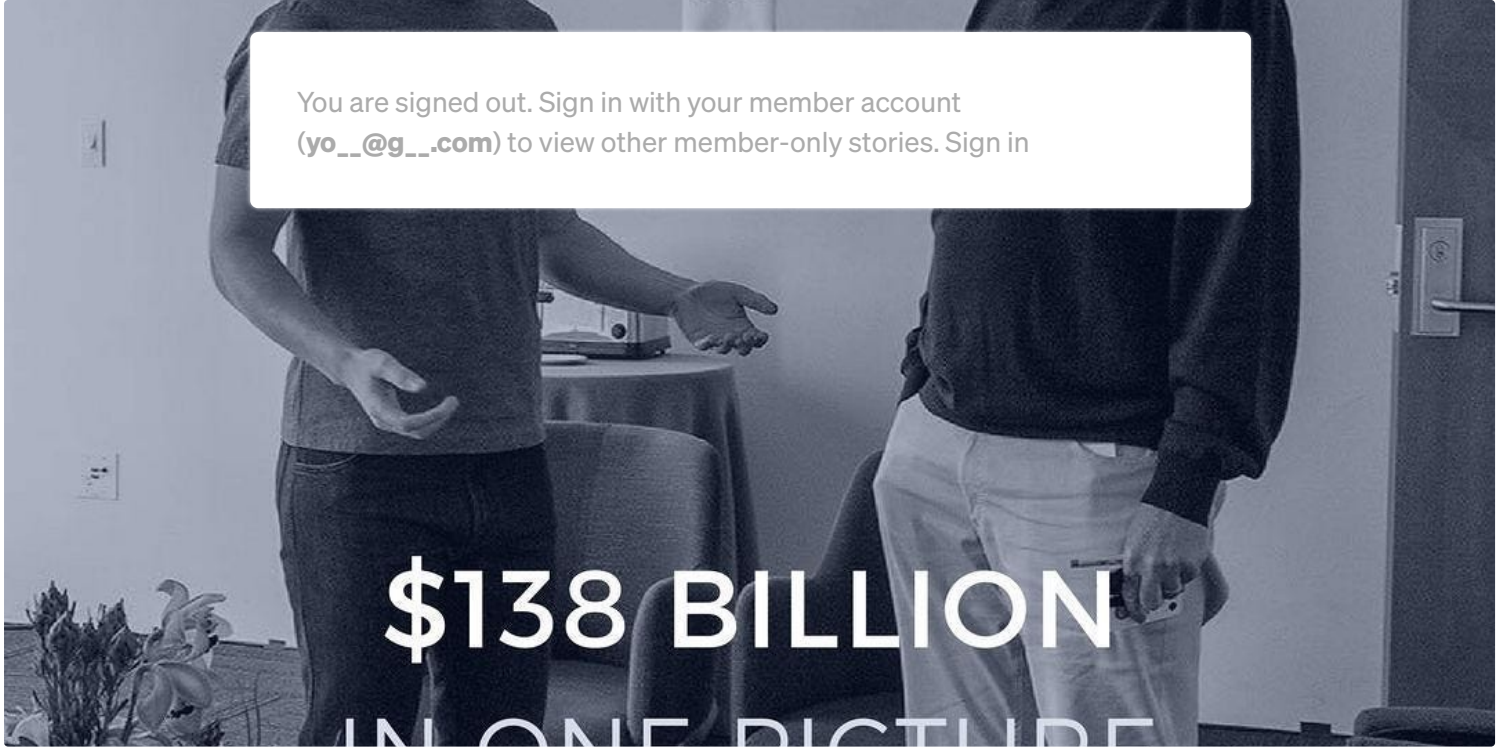
Follow

Written by Bobby

473 Followers · Writer for Level Up Coding

I write about programming

More from Bobby and Level Up Coding



Bobby

What screams “I’m from the upper class”?—Top characteristics of high-class people

Upper class—this is a term about a noble social class with a lot of admiration. These people are very rich and normally, their immense...

· 5 min read · Dec 21, 2018

131

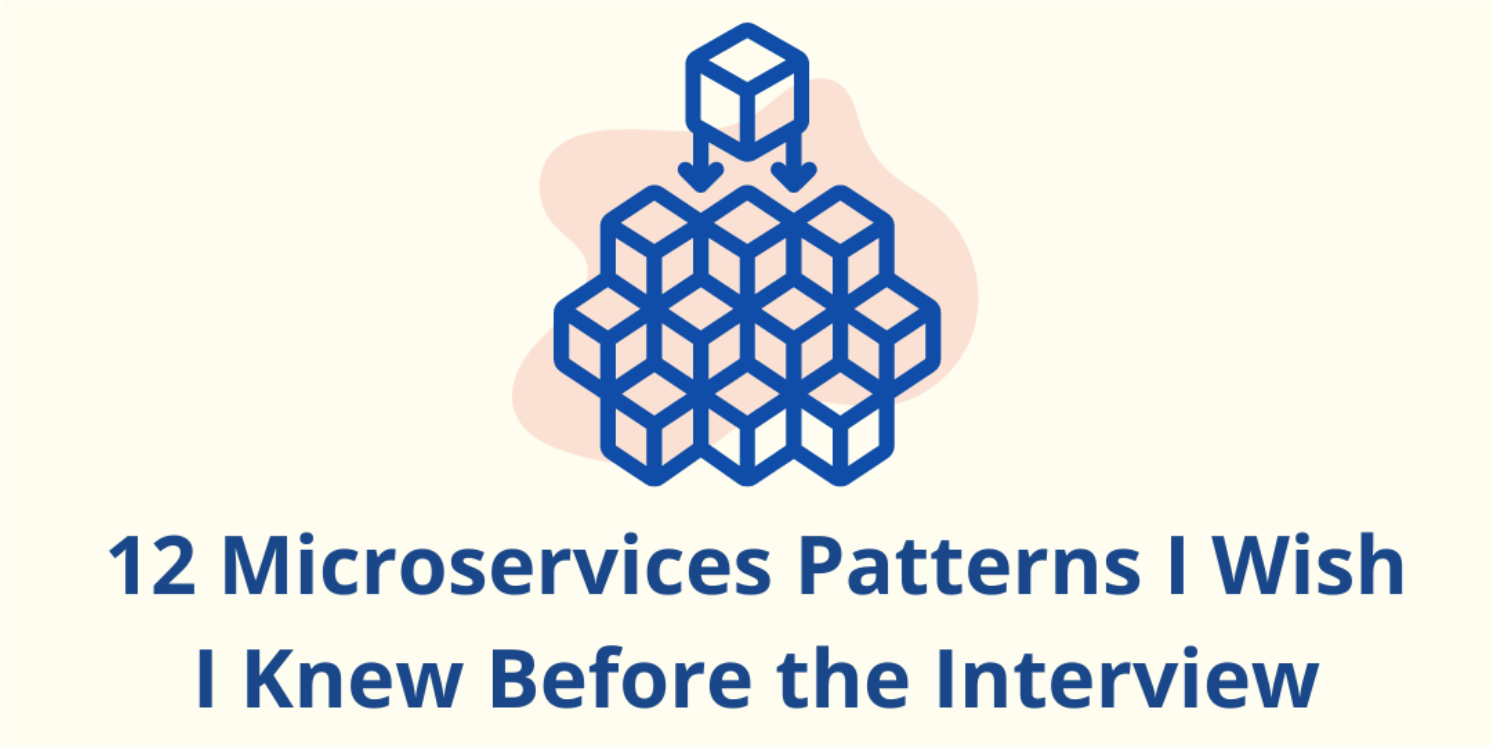
Nitin Sharma in Level Up Coding

AutoGPT is Taking Over the Internet: Here Are the Incredible Use Cases That Will Blow Your Mir

From acting as an

· 6 min read · Apr 25

2.4K 31



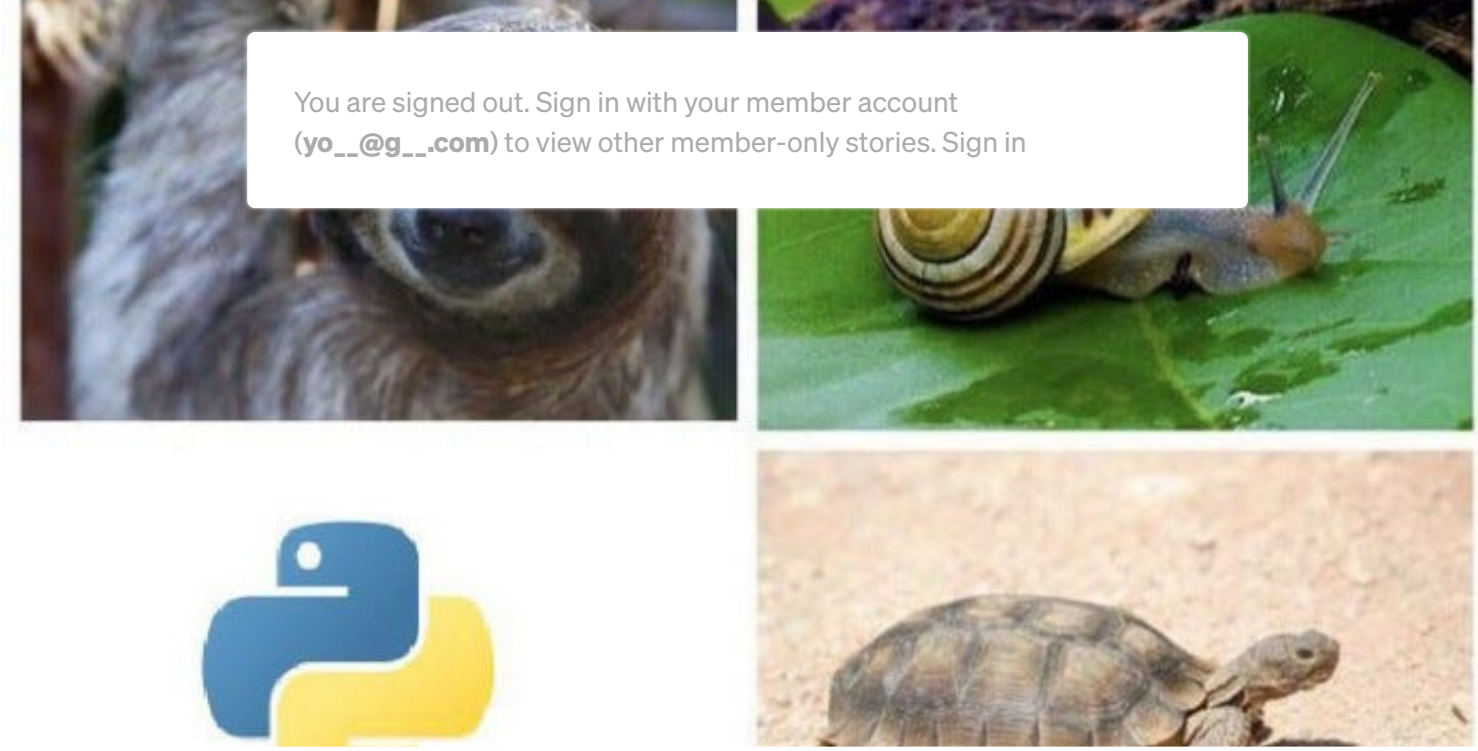
Arslan Ahmad in Level Up Coding

12 Microservices Patterns I Wish I Knew Before the System Design Interview

Mastering the Art of Scalable and Resilient Systems with Essential Microservices Design Patterns

· 13 min read · May 16

1.1K 9

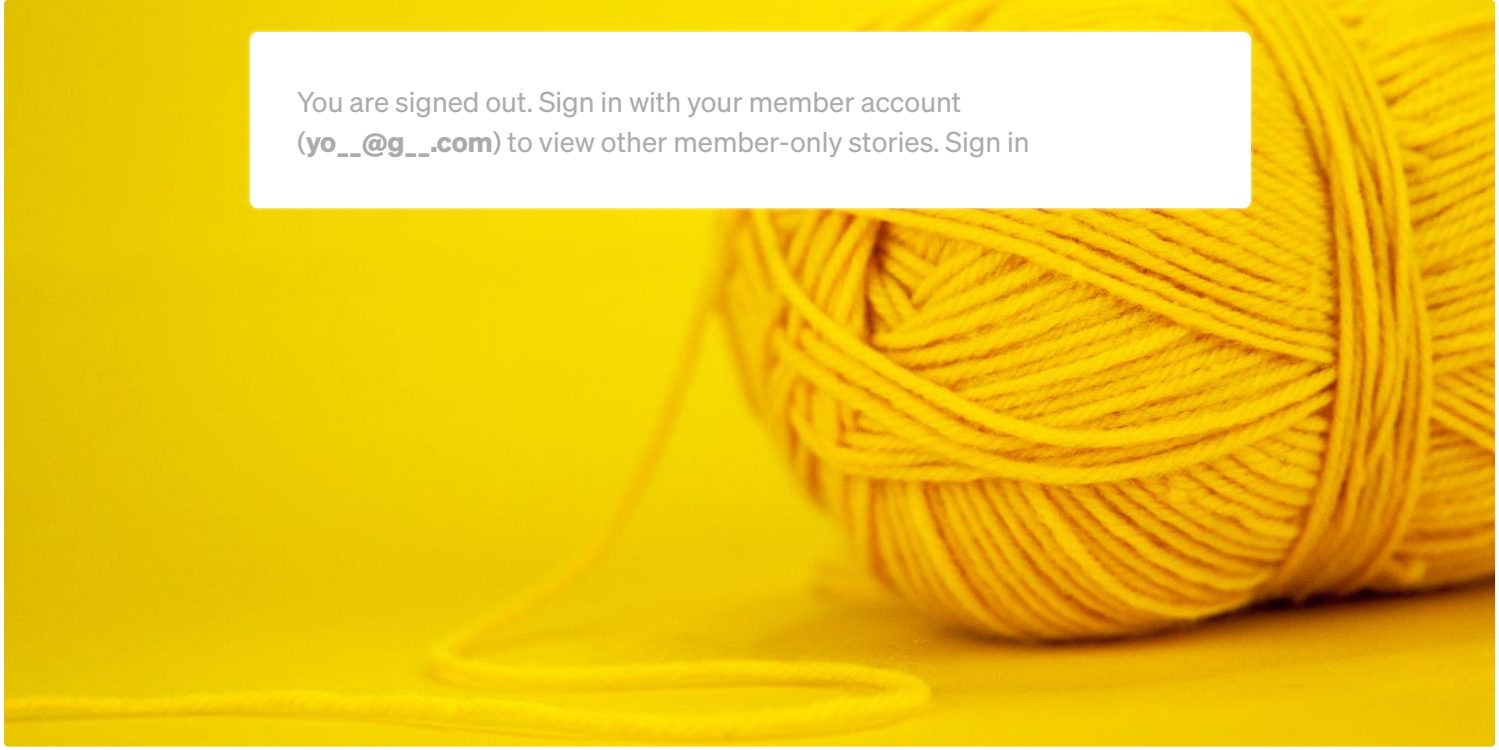


Bobby

Why Python is Popular Despite Being (Super) Slow

Python is one of the most widely used programming languages, and it has been around for more than 28 years now. One common question arises...

· 5 min read · Jan 4, 2019



Martin Heinz in Better Programming

Real Multithreading is Coming to Python—Learn How You Can Use It Now

True multi-core concurrency is coming to Python in 3.12 release and here's how you can use it right now using sub-interpreter API

· 6 min read · May 14

801 9

Gabe Araujo, M.Sc. in Level Up Coding

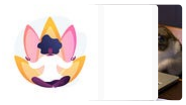
🐼 Introducing PandasAI: The Generative AI Python Library 🐼

Pandas AI is an ad[illegible]alysis and manipulation tool

✦ · 9 min read · M

636 6

Lists

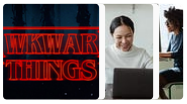


Stories to Help You Grow as a Software Developer

19 stories · 61 saves

Leadership

30 stories · 24 saves

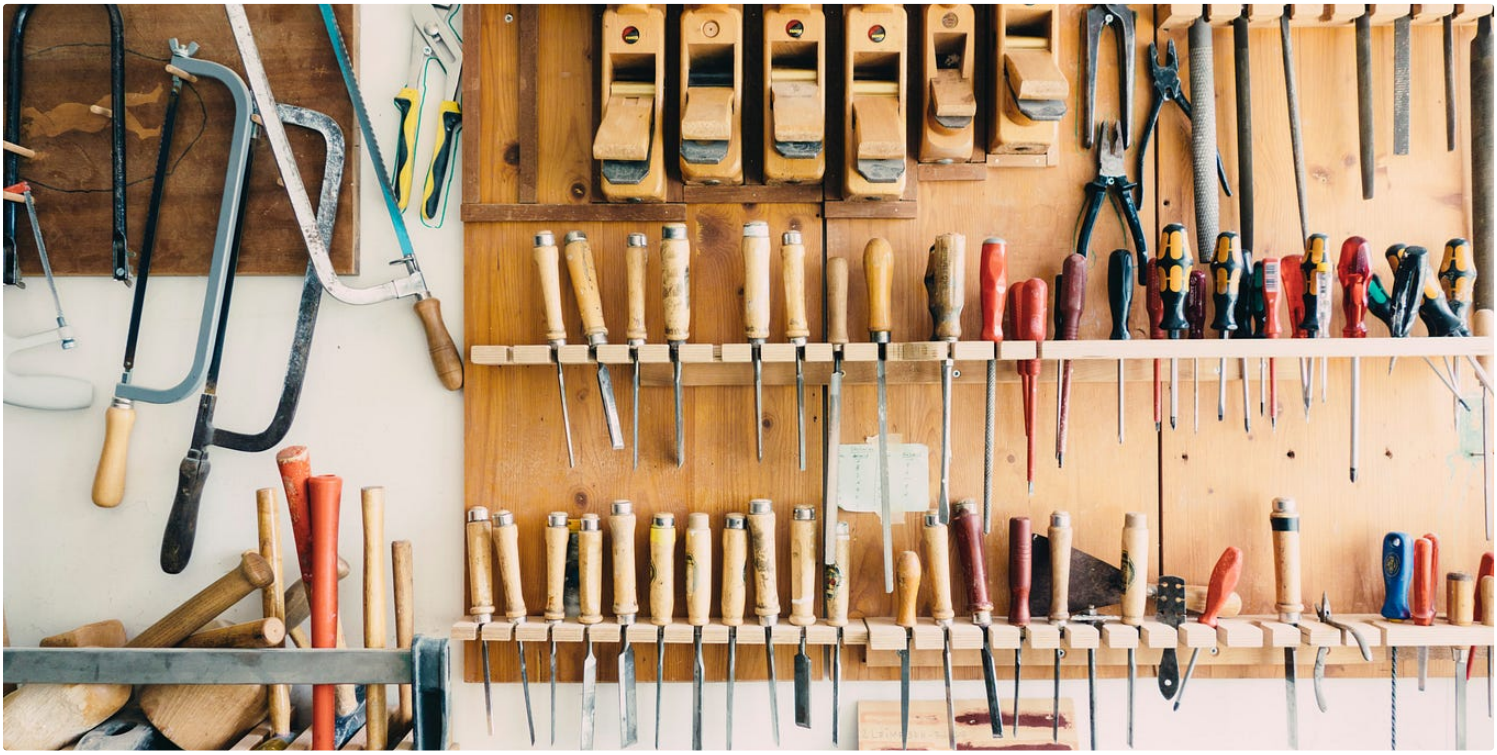


How to Run More Meaningful 1:1 Meetings

11 stories · 29 saves

Stories to Help You Level-Up at Work

19 stories · 50 saves

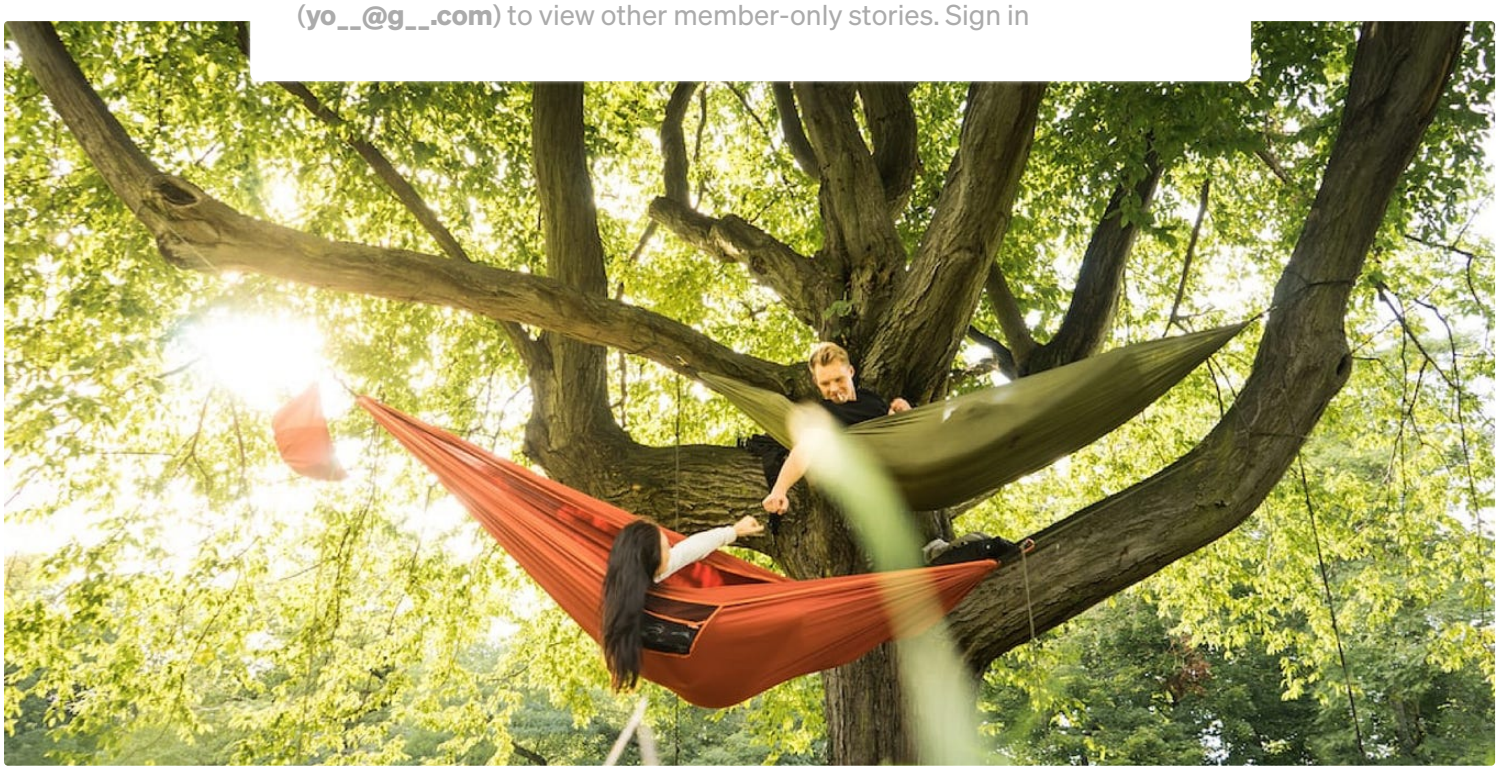


Yancy Dennis in Python in Plain English

Use Python's Functools

A useful Python module that provides tools for working with func

✦ · 3 min read · Jan 4

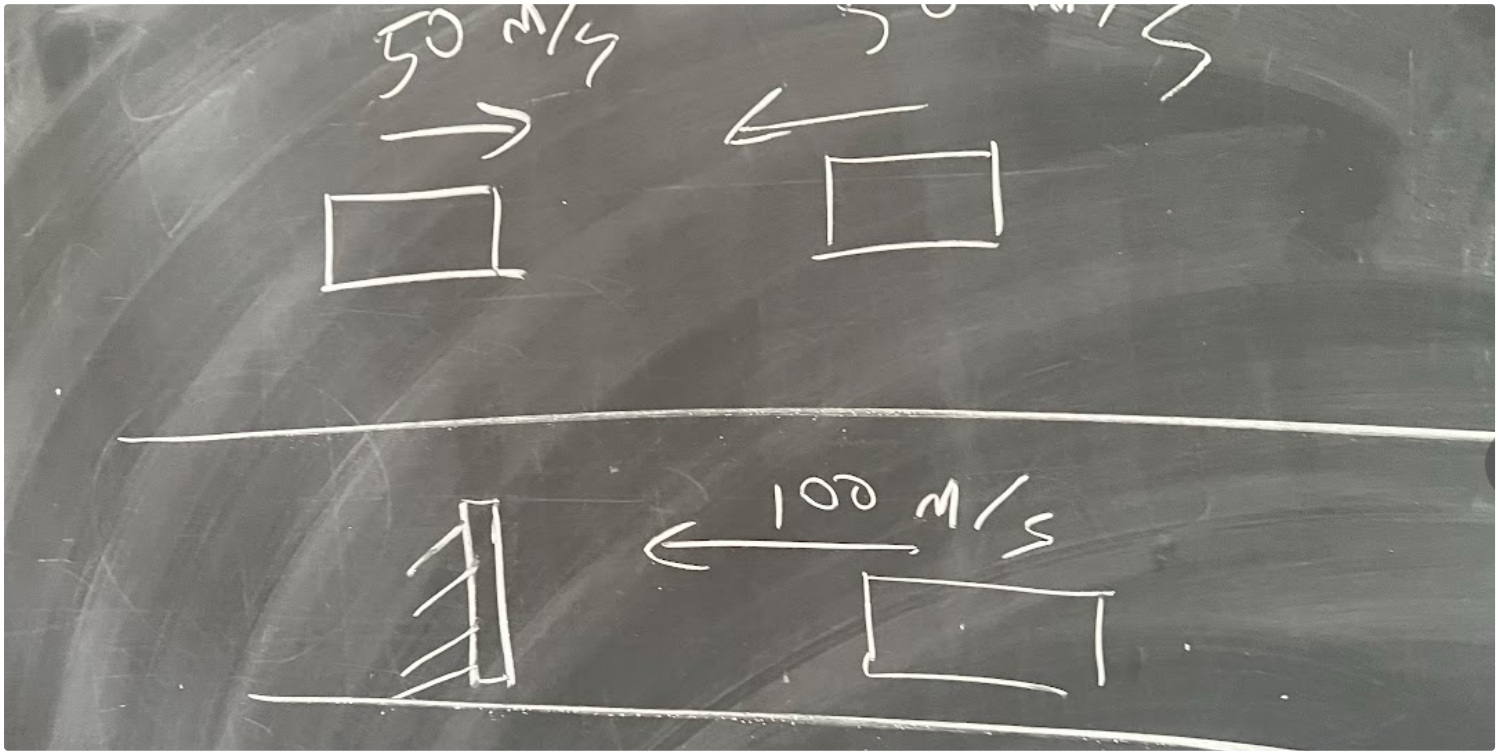


Gabe Araujo, M.Sc. in Level Up Coding

🧬 How I Used Python to Make Everyday Tasks Easier

Hey there! As a busy person with a lot on my plate, I'm always looking for ways to make my life easier.

· 8 min read · Apr 30



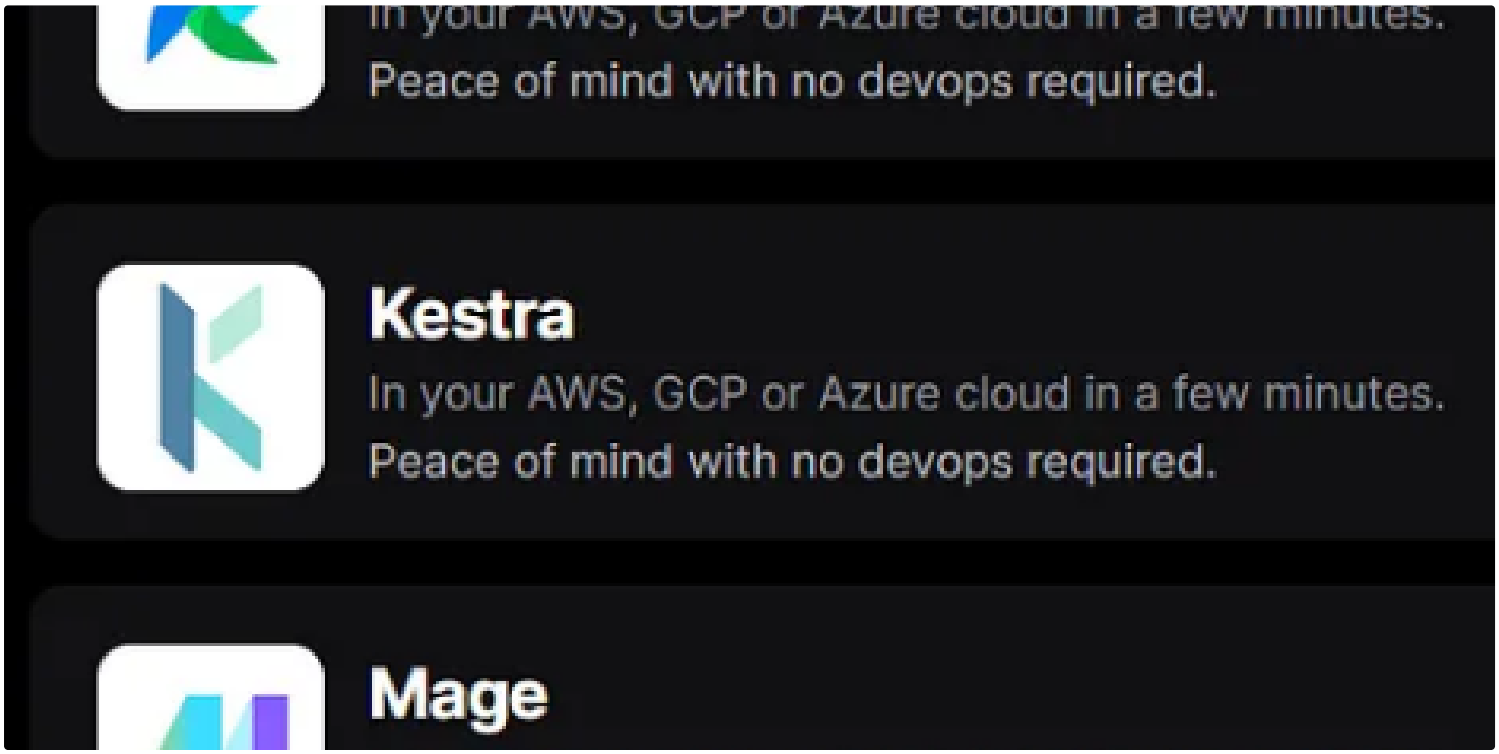
Rhett Allain

Is a 50 mph-5

A long time ago tl ars. Suppose you have two equal m...

9 min read · May 19

537 10



Gabe Araujo, M.Sc. in Level Up Coding

Airflow vs. Mage vs. Kestra

Navigating the Workflow Seas: Choosing the Right Tool for Efficient Data Management

10 min read · May 17

295

See more recommendations