The Most Common Python Code Smells and How to Fix Them

6 min read · Dec 28, 2022

Ryk Kiel Follow

Photo by Waldemar Brandt on Unsplash

Code smells are indicators of poor design or implementation in code. They often indicate that there is a problem with the code, but they are not necessarily errors

that will cause the code to fail. Rather, they are warning signs that the code may be difficult to maintain or may not be as efficient as it could be.

It is important to regularly review and refactor code to identify and resolve code smells. This can improve the quality and maintainability of the codebase and make it easier for developers to work with.

Here are some common code smells that can occur in Python, along with examples and techniques for resolving them:

1. Long Methods

Long methods are those that have a large number of lines of code or that perform multiple tasks. They can be difficult to understand and maintain because it is difficult to see the overall structure and purpose of the method at a glance.

Here is an example of a long method in Python:

```
def calculate_total_cost(items):
    total_cost = 0
    for item in items:
        if item.type == "clothing":
            total_cost += item.price * 1.07
        elif item.type == "electronics":
            total_cost += item.price * 1.15
        else:
            total_cost += item.price
    return total_cost
```

To resolve this code smell, try breaking the long method into smaller, more focused methods. This can make the code easier to understand and maintain because each method has a specific purpose and is shorter in length.

For example, the following refactored code separates the calculation of the total cost for each item type into separate methods:

```
def calculate_total_cost(items):
    total_cost = 0
    for item in items:
        total_cost += calculate_item_cost(item)
    return total_cost

def calculate_item_cost(item):
    if item.type == "clothing":
        return calculate_clothing_cost(item)
    elif item.type == "electronics":
        return calculate_electronics_cost(item)
    return item.price

def calculate_clothing_cost(item):
    return item.price * 1.07

def calculate_electronics_cost(item):
    return item.price * 1.15
```

2. Duplicate Code

Duplicate code is code that is copied and pasted multiple times within the same codebase. It can be difficult to maintain because changes to the code must be made in multiple places, and it can also make the codebase larger and slower to run.

Here is an example of duplicate code in Python:

```
def calculate_total_cost(items):
    total_cost = 0
    for item in items:
        if item.type == "clothing":
            total_cost += item.price * 1.07
        elif item.type == "electronics":
            total_cost += item.price * 1.15
        else:
            total_cost += item.price
    return total_cost

def calculate_total_discount(items):
    total_discount = 0
```

```
    for item in items:
        if item.type == "clothing":
            total_discount += item.price * 0.1
        elif item.type == "electronics":
            total_discount += item.price * 0.2
        else:
            total_discount += item.price * 0.05
    return total_discount
```

To resolve this code smell, try refactoring the code to use a function or method to avoid duplication. This can make the code easier to maintain because changes only need to be made in one place, and it can also make the codebase smaller and faster to run.

For example, the following refactored code uses a function to calculate the discount rate for each item type, rather than duplicating the calculation in multiple places:

```
def calculate_total_cost(items):
    total_cost = 0
    for item in items:
        total_cost += item.price
    return total_cost

def calculate_total_discount(items):
    total_discount = 0
    for item in items:
        total_discount += calculate_item_discount(item)
    return total_discount

def calculate_item_discount(item):
    if item.type == "clothing":
        return item.price * 0.1
    elif item.type == "electronics":
        return item.price * 0.2
    return item.price * 0.05
```

3. Large Classes

Large classes are those that have a large number of methods or attributes. They can be difficult to understand and maintain because it is difficult to see the overall

structure and purpose of the class at a glance.

Here is an example of a large class in Python:

```
class Customer:
    def __init__(self, name, email, phone, address):
        self.name = name
        self.email = email
        self.phone = phone
        self.address = address

    def place_order(self, items):
        order = Order(self, items)
        order.process()
        return order

    def pay_invoice(self, invoice):
        invoice.pay()

    def update_contact_info(self, email=None, phone=None, address=None):
        if email is not None:
            self.email = email
        if phone is not None:
            self.phone = phone
        if address is not None:
            self.address = address
```

To resolve this code smell, try breaking the large class into smaller, more focused classes. This can make the code easier to understand and maintain because each class has a specific purpose and is shorter in length.

For example, the following refactored code separates the customer's contact information into a separate class:

```
class Customer:
    def __init__(self, name, contact_info):
        self.name = name
        self.contact_info = contact_info
```

```
        def place_order(self, items):
            order = Order(self, items)
            order.process()
            return order

        def pay_invoice(self, invoice):
            invoice.pay()

    class ContactInfo:
        def __init__(self, email, phone, address):
            self.email = email
            self.phone = phone
            self.address = address

        def update(self, email=None, phone=None, address=None):
            if email is not None:
                self.email = email
            if phone is not None:
                self.phone = phone
            if address is not None:
                self.address = address
```

4. Long Parameter Lists

Long parameter lists are those that have a large number of parameters. They can be difficult to understand and maintain because it is difficult to see the purpose and meaning of each parameter at a glance.

Here is an example of a long parameter list in Python:

```
def calculate_cost(type, quantity, price, tax_rate, discount_rate, shipping_cos
    cost = quantity * price
    cost += cost * tax_rate
    cost -= cost * discount_rate
    cost += shipping_cost
    return cost
```

To resolve this code smell, try using an object to group related parameters together. This can make the code easier to understand and maintain because the purpose and meaning of each parameter is clearer.

For example, the following refactored code uses a `CostParameters` object to group the cost-related parameters together:

```
class CostParameters:
    def __init__(self, type, quantity, price, tax_rate, discount_rate, shipping
        self.type = type
        self.quantity = quantity
        self.price = price
        self.tax_rate = tax_rate
        self.discount_rate = discount_rate
        self.shipping_cost = shipping_cost

def calculate_cost(cost_parameters):
    cost = cost_parameters.quantity * cost_parameters.price
    cost += cost * cost_parameters.tax_rate
    cost -= cost * cost_parameters.discount_rate
    cost += cost_parameters.shipping_cost
    return cost
```

5. Misplaced Responsibility

Misplaced responsibility is when a class or method is responsible for tasks that are not related to its main purpose. This can make the code difficult to understand and maintain because it is unclear why the class or method is performing those tasks.

Here is an example of misplaced responsibility in Python:

```
class Order:
    def __init__(self, items, customer):
        self.items = items
        self.customer = customer

    def process(self):
        # Perform order processing tasks
        ...
        self.customer.send_email("Your order has been processed")
```

In this example, the `Order` class is responsible for both performing order processing tasks and sending an email to the customer. This is an example of misplaced responsibility because sending emails is not related to the main purpose of the `Order` class.

To resolve this code smell, try moving the unrelated tasks to a separate class or method. This can make the code easier to understand and maintain because each class or method has a specific purpose.

For example, the following refactored code moves the email-sending task to a separate `EmailSender` class:

```
class Order:
    def __init__(self, items, customer):
        self.items = items
        self.customer = customer

    def process(self):
        # Perform order processing tasks
        ...

class EmailSender:
    def send_email(self, customer, message):
        # Send email to customer
        ...

email_sender = EmailSender()

order = Order(items, customer)
order.process()
email_sender.send_email(customer, "Your order has been processed")
```

Conclusion

In conclusion, code smells are indicators of poor design or implementation in code and can make code difficult to understand and maintain. It is important to regularly review and refactor code to identify and resolve code smells.

Some common code smells in Python include long methods, duplicate code, large classes, long parameter lists, and misplaced responsibility. Techniques for resolving these code smells include breaking code into smaller, more focused pieces, using functions or methods to avoid duplication, and moving unrelated tasks to separate classes or methods.

By regularly identifying and resolving code smells, developers can improve the quality and maintainability of their codebase and make it easier for themselves and others to work with.

Thanks for reading and happy coding!

Python | Python Programming | Software Development | Software Engineering

Software

Follow

Written by Ryk Kiel

99 followers · 31 following

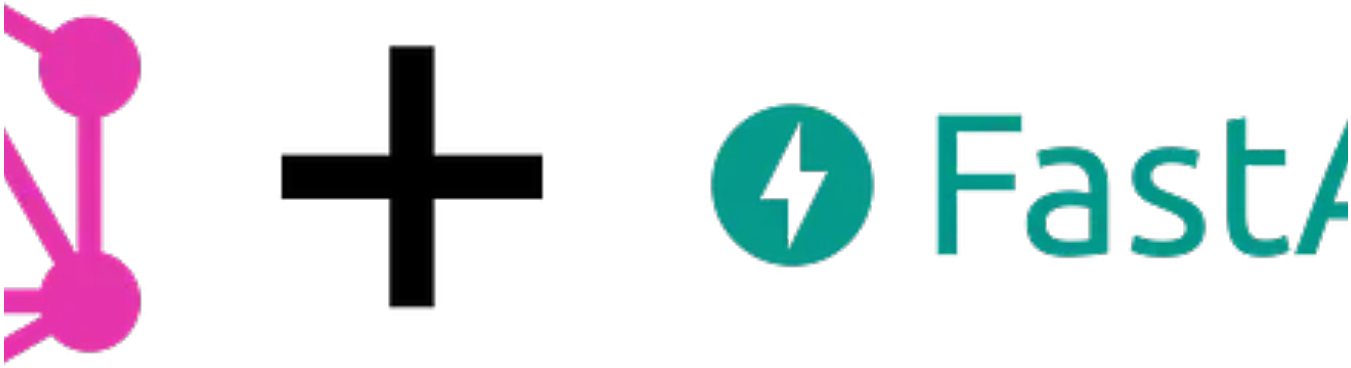
I am a Python lover with a love for problem-solving and creating solutions. I have expertise in web development, data analysis, and machine learning.

No responses yet

Rod Davison

What are your thoughts?

More from Ryk Kiel

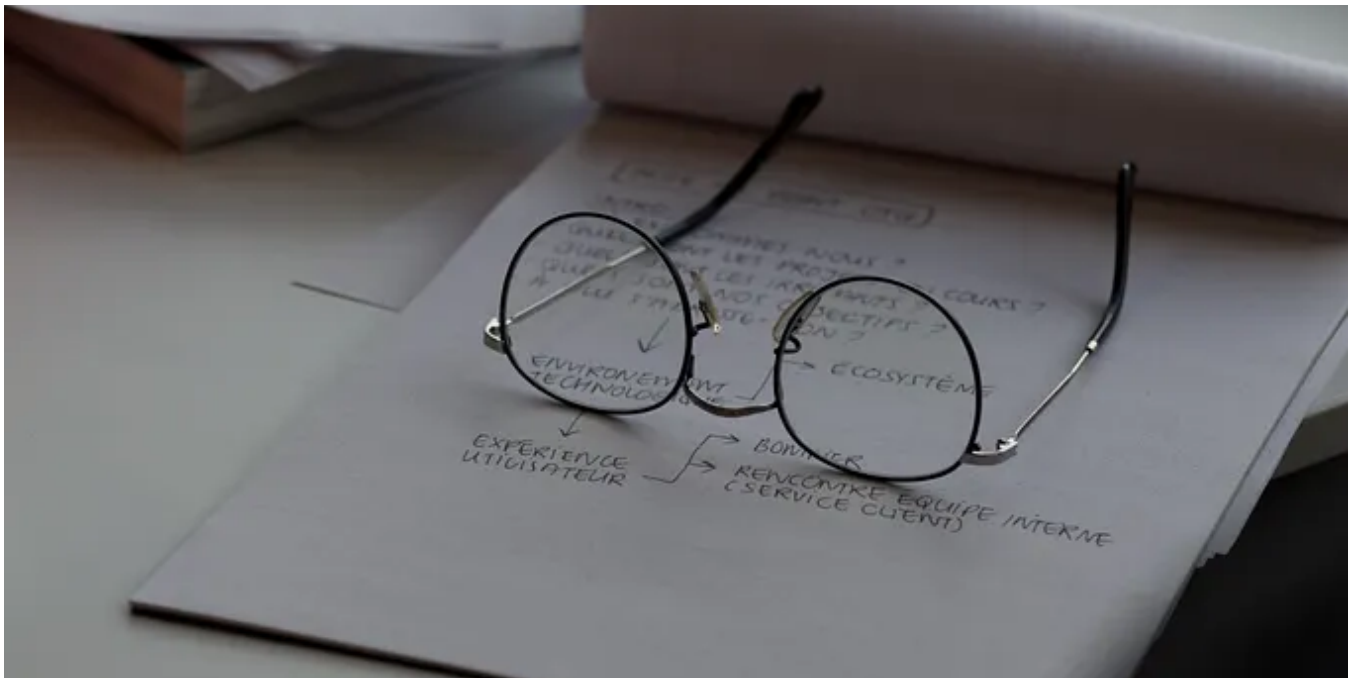


Ryk Kiel

GraphQL and FastAPI: The ultimate combination for building APIs with Python

Getting Started with GraphQL and FastAPI in Python

Jan 5, 2023 103 2



Ryk Kiel

Python Pro Tips: Organizing and Simplifying Your Codebase with Utilities and Helper Functions

Utilities and helper functions are a crucial part of any Python project, as they help to organize and simplify the codebase. When written...

Jan 23, 2023 17 1

Ryk Kiel

Generators in Python: Advantages Over Regular Loops

Python is a powerful programming language that offers various constructs for writing efficient and readable code. One such construct is the...

Feb 13, 2023 28 1

Ryk Kiel

Don’t Let Your Code Get Out of Control: Avoiding Side Effects in Python

As programmers, we’re all familiar with the concept of variables and functions in our code. We use variables to store information, and...

Feb 15, 2023 12

See all from Ryk Kiel

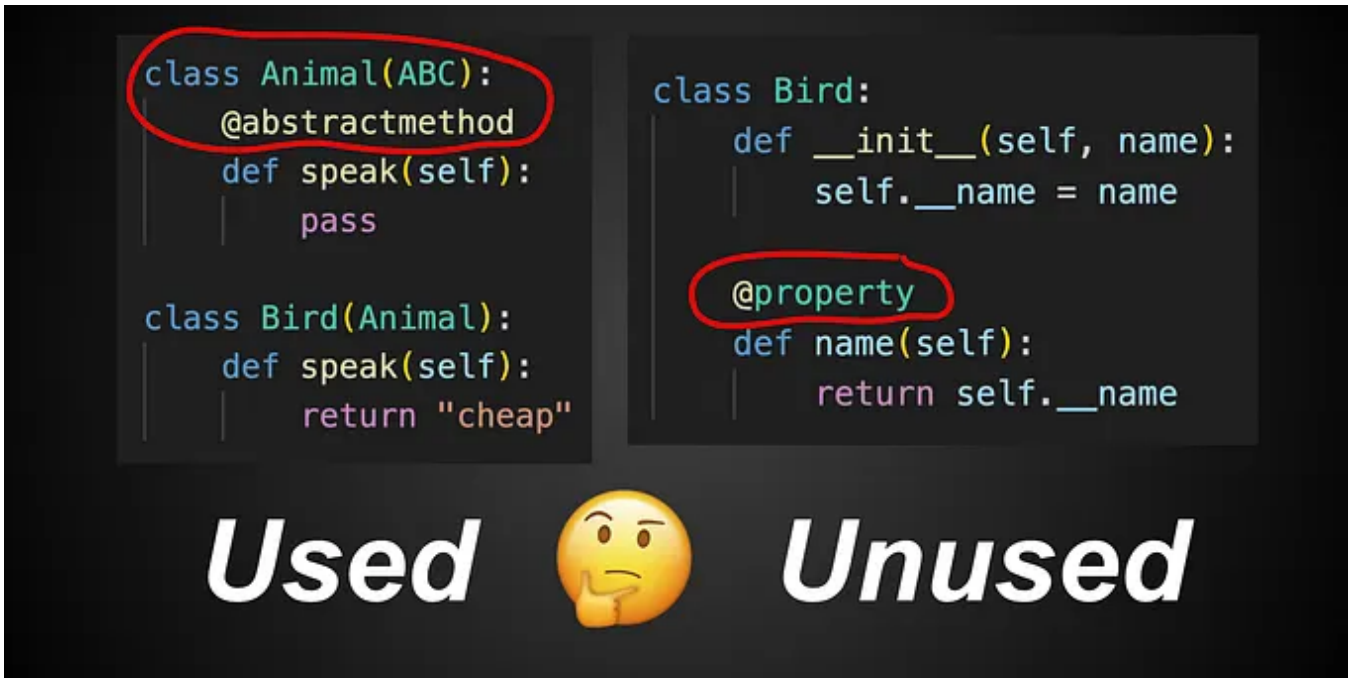
Recommended from Medium

Dikshit Medhi

Understanding Python's Memory Management (Part1): Variable Storage

Python Memory Management part 1

Apr 17 6 1



In Level Up Coding by Liu Zuo Lin

4 Python Class Things We Use In Prod (& 3 We Don't)

Read free: https://www.linkedin.com/pulse/4-python-class-things-we-use-prod-3-dont-the-python-rabbithole-sextc

68 2

Elshad Karimov

🛠️ What is CI/CD in Python Projects?

🔤 First: What does CI/CD even mean?

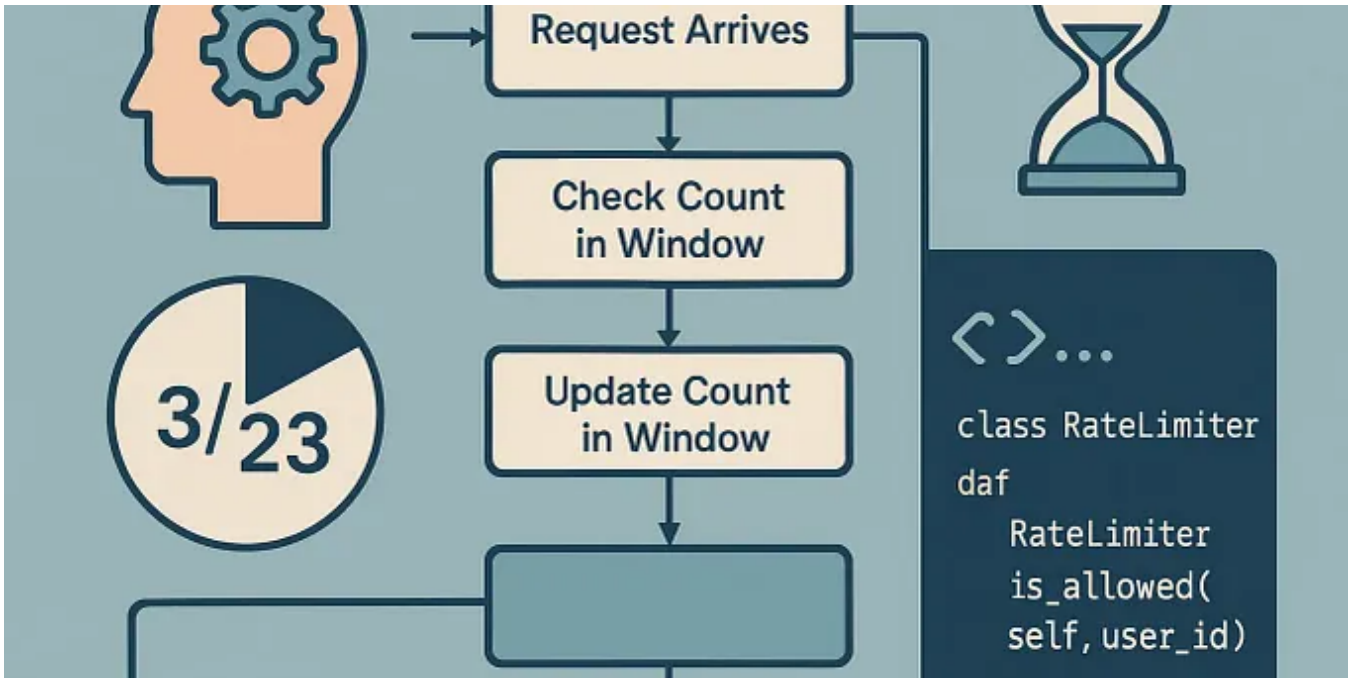
Jul 21 30

Aditya Ghadge

Python Project Structure: Why the ‘src’ Layout Beats Flat Folders (and How to Use My Free Template)

From Kitchen Chaos to Clean Code

May 17 9 1

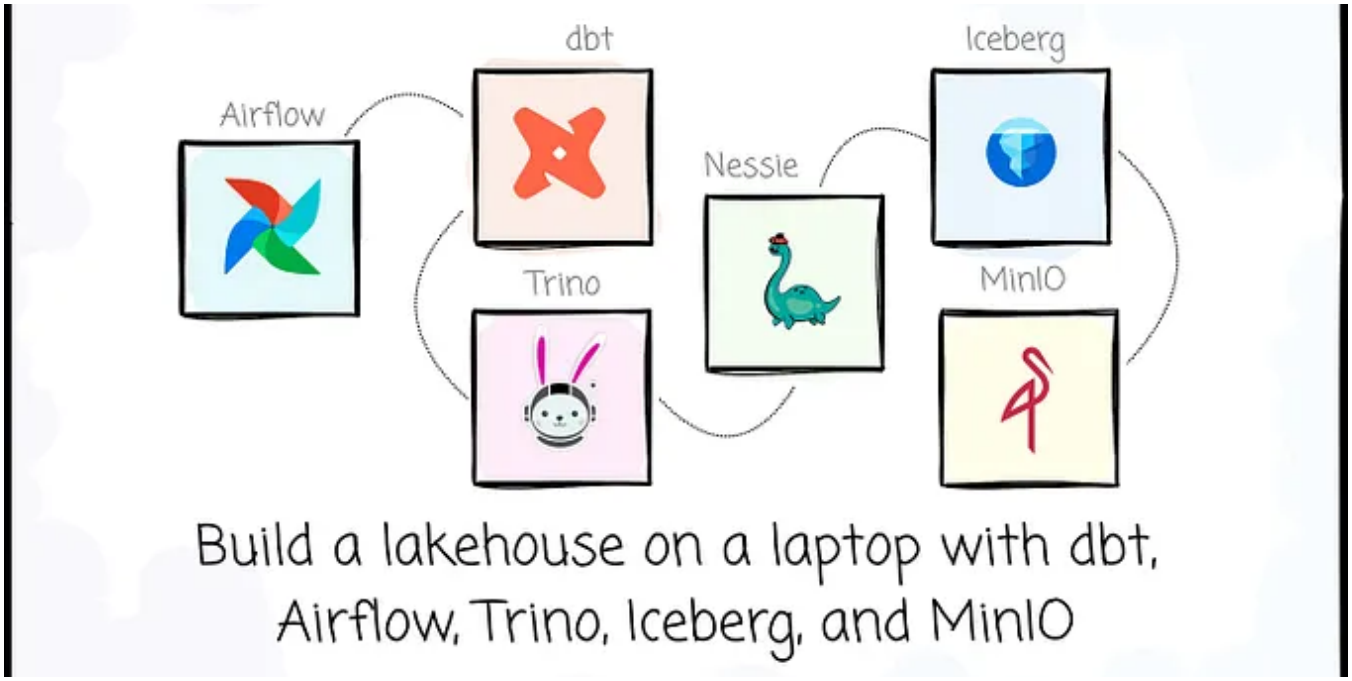


The Latency Gambler

I Interviewed 20+ Engineers. Here's Why Most Can't Code

Over the past year as a Senior Software Engineer at a B2B SaaS company, I've conducted 20+ technical interviews for roles ranging from...

Sep 9 1.3K 51



In Data Engineer Things by Vu Trinh

Build a lakehouse on a laptop with dbt, Airflow, Trino, Iceberg, and MinIO

A pet project for learning data engineering

Sep 14 345 11

See more recommendations