Open in app ↗

Search

Write

✦ Member-only story

Essential Guidelines for Writing Functions in Python

Unlock the Secrets of Efficient Function Design, Error Handling, and Performance Optimization in Python

Ravi | Python ✍ · Follow

Published in DataDrivenInvestor · 12 min read · Mar 25, 2024

67

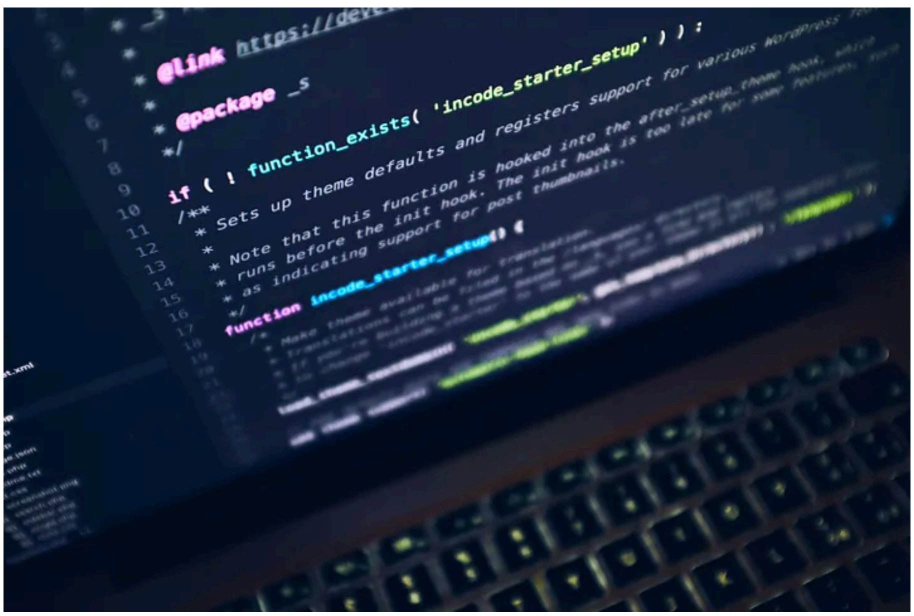


Photo by Luca Bravo on Unsplash

Functions are the building blocks of any Python program, and writing good functions is crucial for creating maintainable, reusable, and efficient code. Well-designed functions not only make the code easier to understand and debug but also promote code reusability and modularity. In this article, we will explore the essential guidelines for writing functions in Python, covering various aspects such as naming conventions, documentation, design principles, parameter handling, error handling, testing, optimization, and best practices. By following these guidelines, you can elevate your Python programming skills and write more professional and robust code.

Mastering Function Interfaces in Python: Unleashing the Power of Type Hints and Protocols

A Deep Dive into Implementing, Leveraging, and Advancing Function Interfaces for Cleaner, More Maintainable Code

ravi-m.medium.com

Naming and Documentation

This section emphasizes the significance of clear and descriptive names for functions, which facilitate understanding and maintenance. It also covers the essentials of documenting functions through docstrings, comments, and annotations to ensure that the code is self-explanatory and accessible to others.

Descriptive Function Names

When naming your functions, it's essential to choose descriptive and meaningful names that clearly convey the purpose of the function. Follow the Python naming convention of using lowercase letters and underscores for function names (e.g., `calculate_average`, `process_data`). Avoid using abbreviations or cryptic names that can make the code harder to understand.

```
def calculate_average(numbers):
    """Calculate the average of a list of numbers."""
    total = sum(numbers)
    count = len(numbers)
    return total / count
```

Docstrings

Docstrings are string literals that appear as the first statement in a function. They provide a brief description of what the function does, its parameters, and its return value. Docstrings serve as documentation for your functions and make it easier for other developers (including yourself) to understand how to use them.

```
def calculate_average(numbers):
    """
    Calculate the average of a list of numbers.

    Args:
        numbers (list): A list of numbers.

    Returns:
        float: The average of the numbers.
    """
    total = sum(numbers)
    count = len(numbers)
    return total / count
```

Comments and Annotations

In addition to docstrings, you can use comments and type annotations to provide additional information about your functions. Comments should be used sparingly and only when necessary to clarify complex or non-obvious code. Type annotations, introduced in Python 3.5, allow you to specify the expected types of function parameters and return values.

```
def calculate_average(numbers: list[float]) -> float:
    """Calculate the average of a list of numbers."""
    total = sum(numbers)
    count = len(numbers)
    return total / count
```

Function Design Principles

The principles discussed here focus on creating functions with a single responsibility, keeping them small and focused, and avoiding the use of global variables. It also touches on designing functions that are easy to extend and modify, which is crucial for long-term code maintenance.

Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a function should have only one reason to change. In other words, a function should do one thing and do it well. By following the SRP, you create functions that are focused, easier to understand, and less prone to bugs.

```
def read_data_from_file(file_path):
    """Read data from a file."""
    with open(file_path, 'r') as file:
        data = file.read()
    return data

def process_data(data):
    """Process the data."""
    processed_data = data.upper()
    return processed_data

def write_data_to_file(file_path, data):
    """Write data to a file."""
    with open(file_path, 'w') as file:
        file.write(data)
```

Keep Functions Small and Focused

Functions should be small and focused, typically consisting of no more than a few dozen lines of code. Small functions are easier to understand, test, and maintain. If a function starts to grow too large, consider breaking it down into smaller, more manageable functions.

```
def calculate_statistics(numbers):
    """Calculate various statistics of a list of numbers."""
    average = calculate_average(numbers)
    median = calculate_median(numbers)
    standard_deviation = calculate_standard_deviation(numbers)
    return average, median, standard_deviation

def calculate_average(numbers):
    """Calculate the average of a list of numbers."""
    total = sum(numbers)
    count = len(numbers)
    return total / count

def calculate_median(numbers):
    """Calculate the median of a list of numbers."""
    sorted_numbers = sorted(numbers)
    count = len(sorted_numbers)
    mid = count // 2
    if count % 2 == 0:
        return (sorted_numbers[mid - 1] + sorted_numbers[mid]) / 2
    else:
        return sorted_numbers[mid]

def calculate_standard_deviation(numbers):
    """Calculate the standard deviation of a list of numbers."""
    average = calculate_average(numbers)
    squared_diff = [(x - average) ** 2 for x in numbers]
    variance = sum(squared_diff) / len(numbers)
    return variance ** 0.5
```

Avoid Global Variables

It’s generally a good practice to avoid using global variables within functions. Functions should rely on their parameters and local variables to perform their tasks. Global variables can make the code harder to understand and maintain, as they introduce dependencies and can lead to unexpected behavior.

```
# Avoid global variables
global_counter = 0

def increment_counter():
    global global_counter
    global_counter += 1

# Use local variables and parameters instead
def increment_counter(counter):
    return counter + 1
```

Design for Extension

When designing functions, consider making them extensible by providing parameters or options that allow the caller to customize the function’s behavior. This way, you can reuse the function in different scenarios without modifying its core functionality.

```
def read_data_from_file(file_path, encoding='utf-8'):
    """Read data from a file with a specified encoding."""
    with open(file_path, 'r', encoding=encoding) as file:
        data = file.read()
    return data
```

Parameters and Arguments

It includes using default parameters sparingly, avoiding an excessive number of parameters, effectively utilizing argument unpacking with `*args` and `**kwargs`, and the importance of using immutable objects for default arguments to prevent bugs.

Default Parameters

Default parameters allow you to specify default values for function parameters, making them optional for the caller. Use default parameters sparingly and only when there is a sensible default value. Avoid using mutable objects (e.g., lists or dictionaries) as default parameter values, as they can lead to unexpected behavior.

```
def greet(name, greeting='Hello'):
    """Greet a person with an optional greeting."""
    return f"{greeting}, {name}!"

print(greet("Alice"))  # Output: Hello, Alice!
print(greet("Bob", "Hi"))  # Output: Hi, Bob!
```

Avoid Too Many Parameters

Functions with too many parameters can be difficult to understand and use. If a function requires many parameters, consider grouping related parameters into a single object or using a configuration object.

```
# Avoid too many parameters
def create_user(username, email, password, first_name, last_name, age, address):
    # ...

# Group related parameters into an object
class User:
    def __init__(self, username, email, password, first_name, last_name, age, ad
        self.username = username
        self.email = email
        self.password = password
        self.first_name = first_name
```

```
        self.last_name = last_name
        self.age = age
        self.address = address

def create_user(user):
    # ...
```

Use Argument Unpacking

Python allows you to unpack sequences (e.g., lists or tuples) and dictionaries into individual arguments using the `*` and `**` operators, respectively. Argument unpacking can make your code more concise and readable.

```
def calculate_sum(*numbers):
    """Calculate the sum of any number of arguments."""
    return sum(numbers)

print(calculate_sum(1, 2, 3))  # Output: 6
print(calculate_sum(4, 5, 6, 7))  # Output: 22

def print_user_info(**user_info):
    """Print user information from a dictionary."""
    for key, value in user_info.items():
        print(f"{key}: {value}")

print_user_info(name="Alice", age=25, city="New York")
```

Immutable Default Arguments

When using default parameter values, be cautious with mutable objects such as lists or dictionaries. If a mutable object is used as a default value, it is created only once when the function is defined and shared across all function calls. This can lead to unexpected behavior if the mutable object is modified.

```
# Mutable default argument (problematic)
def append_to_list(item, my_list=[]):
    my_list.append(item)
    return my_list

print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [1, 2]  (Unexpected)

# Immutable default argument (correct)
def append_to_list(item, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(item)
    return my_list

print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [2]  (Expected)
```

Error Handling

Error handling is an essential aspect of robust function design. This section explains the preference for exceptions over error checking, how to handle exceptions gracefully, and the use of assertions to implement internal checks within a function.

Exceptions Over Error Checking

Python encourages the use of exceptions for error handling instead of manual error checking. The `try-except` block allows you to handle exceptions gracefully and provide meaningful error messages to the caller.

```
def divide_numbers(a, b):
    """Divide two numbers."""
    try:
        result = a / b
    except ZeroDivisionError:
        raise ValueError("Cannot divide by zero.")
    return result

print(divide_numbers(10, 2))  # Output: 5.0
print(divide_numbers(10, 0))  # Raises ValueError: Cannot divide by zero.
```

Handle Exceptions Gracefully

When handling exceptions, provide clear and informative error messages that help the caller understand what went wrong and how to fix it. Avoid catching broad exceptions (e.g., `Exception`) unless absolutely necessary.

```
def read_data_from_file(file_path):
    """Read data from a file."""
    try:
        with open(file_path, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        raise FileNotFoundError(f"File not found: {file_path}")
    except PermissionError:
        raise PermissionError(f"Permission denied: {file_path}")
    return data
```

Use Assertions

Assertions are a way to check for conditions that should always be true at a certain point in your code. They help catch logical errors early and provide a clear indication of what went wrong. Use assertions to validate function parameters, preconditions, and postconditions.

```
def calculate_square_root(number):
    """Calculate the square root of a number."""
    assert number >= 0, "Number must be non-negative."
```

```
    return number ** 0.5

print(calculate_square_root(16))  # Output: 4.0
print(calculate_square_root(-1))  # Raises AssertionError: Number must be non-ne
```

Testing and Maintenance

Here, the focus is on the importance of writing unit tests to verify function behavior, testing edge cases to ensure comprehensive coverage, refactoring repeated code to adhere to the DRY principle, and regularly updating functions to maintain efficiency.

Write Unit Tests

Writing unit tests for your functions is essential to ensure their correctness and reliability. Unit tests help catch bugs early, serve as documentation, and provide confidence when refactoring or modifying the code.

```
def is_prime(number):
    """Check if a number is prime."""
    if number < 2:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
    return True

# Unit tests
assert is_prime(2) == True
assert is_prime(7) == True
assert is_prime(12) == False
assert is_prime(0) == False
```

Test Edge Cases

When writing unit tests, it's crucial to test not only the typical scenarios but also the edge cases. Edge cases are the extreme or unusual situations that your function might encounter. Testing edge cases helps uncover hidden bugs and ensures that your function behaves correctly in all possible scenarios.

```
def calculate_average(numbers):
    """Calculate the average of a list of numbers."""
    assert len(numbers) > 0, "List must not be empty."
    total = sum(numbers)
    count = len(numbers)
    return total / count

# Unit tests
assert calculate_average([1, 2, 3]) == 2.0
assert calculate_average([0]) == 0.0
assert calculate_average([-1, 1]) == 0.0
```

```
try:
    calculate_average([])
except AssertionError as e:
    assert str(e) == "List must not be empty."
```

Refactor Repeated Code

If you find yourself writing similar code multiple times, consider refactoring it into a separate function. This follows the DRY (Don't Repeat Yourself) principle, which promotes code reuse and maintainability.

```
def calculate_circle_area(radius):
    """Calculate the area of a circle."""
    return 3.14159 * radius ** 2

def calculate_circle_circumference(radius):
    """Calculate the circumference of a circle."""
    return 2 * 3.14159 * radius

# Refactored code
def calculate_circle_area(radius):
    """Calculate the area of a circle."""
    return PI * radius ** 2

def calculate_circle_circumference(radius):
    """Calculate the circumference of a circle."""
    return 2 * PI * radius

PI = 3.14159
```

Regularly Refactor

As your codebase evolves and grows, it's important to regularly refactor your functions to keep them up-to-date and efficient. Refactoring involves improving the code's structure and design without changing its external behavior. Regular refactoring helps maintain code quality, readability, and performance.

```
# Before refactoring
def process_data(data):
    result = []
    for item in data:
        if item % 2 == 0:
            result.append(item * 2)
        else:
            result.append(item * 3)
    return result

# After refactoring
def process_item(item):
    if item % 2 == 0:
        return item * 2
    else:
        return item * 3

def process_data(data):
    return [process_item(item) for item in data]
```

Performance and Optimization

This part highlights the need to profile functions before optimizing them, balancing readability with performance, and using generators for memory efficiency when dealing with large data sets.

Profile Before Optimizing

Before optimizing your functions for performance, it's essential to profile your code to identify the actual performance bottlenecks. Premature optimization can lead to unnecessary complexity and may not yield significant performance gains.

```
import cProfile

def fibonacci(n):
    """Calculate the nth Fibonacci number."""
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

cProfile.run('fibonacci(30)')
```

Optimize for Readability vs. Performance

When optimizing your functions, strike a balance between readability and performance. In most cases, prioritize code readability and clarity over minor performance gains. Only optimize when necessary and when the performance benefit outweighs the cost of reduced readability.

```
# Optimized for readability
def is_prime(number):
    """Check if a number is prime."""
    if number < 2:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
    return True

# Optimized for performance
def is_prime(number):
    """Check if a number is prime."""
    if number < 2:
        return False
    if number == 2:
        return True
    if number % 2 == 0:
        return False
    for i in range(3, int(number ** 0.5) + 1, 2):
        if number % i == 0:
            return False
    return True
```

Use Generators for Large Data Sets

When working with large data sets, consider using generators instead of storing all the data in memory. Generators allow you to generate values on-the-fly, reducing memory consumption and improving performance.

```
def generate_even_numbers(n):
    """Generate even numbers up to n."""
    for i in range(2, n + 1, 2):
        yield i

even_numbers = generate_even_numbers(100)
for number in even_numbers:
    print(number)
```

Best Practices and Idioms

Best practices include adhering to PEP 8 for consistent coding style, embracing the guiding principles of Python for simplicity and readability, avoiding magic numbers by using named constants, and limiting the use of nested functions for better code structure.

Follow PEP 8

PEP 8 is the official style guide for Python code. It provides guidelines for code formatting, naming conventions, and best practices. Following PEP 8 makes your code more consistent, readable, and maintainable.

```
# PEP 8 compliant
def calculate_sum(a, b):
    """Calculate the sum of two numbers."""
    return a + b
```

The Zen of Python

The Zen of Python, also known as PEP 20, is a collection of guiding principles for writing Python code. It emphasizes simplicity, readability, and explicitness. Keep these principles in mind when writing your functions.

```
import this

"""
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

```
Sparse is better than dense.
Readability counts.
...
"""
```

Avoid Magic Numbers

Magic numbers are literal values that appear in the code without any context or explanation. They make the code harder to understand and maintain. Instead of using magic numbers, use named constants to provide meaningful names and improve code readability.

```
# Avoid magic numbers
def calculate_area(radius):
    return 3.14159 * radius ** 2

# Use named constants
PI = 3.14159

def calculate_area(radius):
    return PI * radius ** 2
```

Limit the Use of Nested Functions

While Python allows the definition of nested functions (functions inside other functions), it's generally a good practice to limit their use. Nested functions can make the code harder to read and understand. Instead, prefer to define functions at the top level of the module.

```
# Avoid deeply nested functions
def outer_function():
    def inner_function1():
        def inner_function2():
            # ...

# Prefer top-level functions
def inner_function2():
    # ...

def inner_function1():
    # ...

def outer_function():
    # ...
```

Advanced Practices

Advanced practices involve using decorators judiciously to enhance functions, preferring helper functions over complex expressions for readability, being cautious with recursion due to Python's limitations, and using context managers for effective resource management.

Use Decorators Wisely

Decorators are a powerful feature in Python that allow you to modify the behavior of functions without changing their code. They can be used for logging, timing, authentication, and more. However, use decorators wisely and avoid overusing them, as they can make the code harder to understand and debug.

```
def log_execution_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time of {func.__name__}: {end_time - start_time:.5f} s
        return result
    return wrapper

@log_execution_time
def process_data(data):
    # ...
```

Prefer Helper Functions Over Complex Expressions

When faced with complex expressions or calculations, consider extracting them into separate helper functions. This improves code readability and reusability.

```
# Complex expression
def calculate_price(base_price, tax_rate, discount):
    return base_price * (1 + tax_rate) * (1 - discount)

# Helper functions
def apply_tax(price, tax_rate):
    return price * (1 + tax_rate)

def apply_discount(price, discount):
    return price * (1 - discount)

def calculate_price(base_price, tax_rate, discount):
    price_with_tax = apply_tax(base_price, tax_rate)
    final_price = apply_discount(price_with_tax, discount)
    return final_price
```

Be Cautious with Recursion

Recursion is a programming technique where a function calls itself to solve a problem. While recursion can lead to elegant and concise solutions, be cautious when using it in Python. Python has a limited recursion depth (default is 1000), and exceeding it can result in a "maximum recursion depth exceeded" error. For deep recursions, consider using iterative approaches or increasing the recursion limit.

```
# Recursive function
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Iterative alternative
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Use Context Managers

Context managers provide a convenient way to manage resources such as files, database connections, or locks. They ensure that the resources are properly acquired and released, even in the presence of exceptions. Use the `with` statement to work with context managers and ensure proper resource management.

```
# Without context manager
file = open("data.txt", "r")
try:
    data = file.read()
finally:
    file.close()

# With context manager
with open("data.txt", "r") as file:
    data = file.read()
```

Conclusion

Writing good functions is an essential skill for any Python programmer. By following the guidelines discussed in this article, you can create functions that are readable, maintainable, and efficient. Remember to choose descriptive names, document your functions, keep them small and focused, handle errors gracefully, write tests, and continuously refactor your code.

As you write more Python code, you'll develop a better understanding of these guidelines and when to apply them.

Happy coding!

Visit us at *DataDrivenInvestor.com*

Subscribe to DDIntel *here*.

DDI Services: https://linktr.ee/datadriveninvestor

Featured Article:

Introducing Singapore's Fund Platform | DataDrivenInvestor
Accelerating Fund's Setup, Launch, and Operational Efficiency Introduction Singapore's ambition to lead the global...
www.datadriveninvestor.com

Join our creator ecosystem *here*.

DDI Official Telegram Channel: https://t.me/+tafUp6ecEys4YjQ1

Follow us on *LinkedIn*, *Twitter*, *YouTube*, and *Facebook*.

Python | Python Programming | Software Development | Coding | Programming

More from the list: "Reading list"
Curated by @reenum

Jon Anderson
Python & Baseball Savant —Lesson #2
1 min read · Jul 8, 2021

Ben F... in Towards Data S...
Anatomy of a Polars Query: A Syntax...
· 7 min read · Mar 19, 2024

Harish M... in Python Pan...
How to Import Data From Multiple Text Files in...
· 5 min read · Jan 1, 2021

Foc ana
11 m

View list

Written by Ravi | Python ✍️

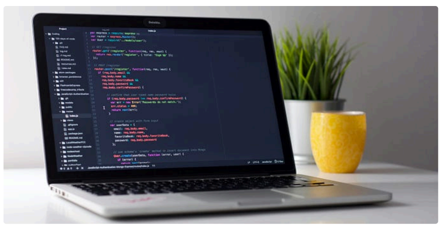
Follow

2.9K Followers · Writer for DataDrivenInvestor

💻 Software Engineer | Tech Explorer | Tech Storyteller | Astronomy Enthusiast | Writing About the Art of Coding (Python) | Books | #Programming #Python

---

More from Ravi | Python ✍️ and DataDrivenInvestor



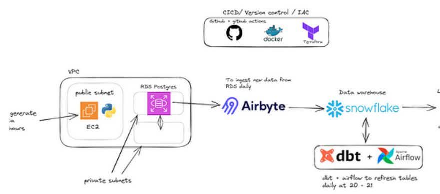
Ravi | Python ✍️ in Python in Plain English

Mastering Pythonic Code: 60 Examples and Best Practices for...

60 Examples of Best Practices for Writing Clean, Efficient, and Maintainable Python...

✦ · 10 min read · Mar 19, 2024

319



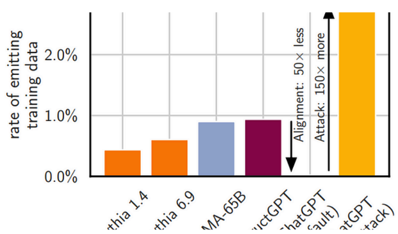
Dorian Teffo in DataDrivenInvestor

How I Built This Data Platform in One Week

This will certainly be my longest project (and the most expensive, so please like and...

7 min read · Mar 22, 2024

741 15



Devansh in DataDrivenInvestor

Google extracted ChatGPT's Training Data using a silly trick.

Scalable Extraction of Training Data from (Production) Language Models

13 min read · Jan 7, 2024

3.2K 22

Ravi | Python ✍️ in Python in Plain English

Mastering Python: 100 Advanced Python Cheatsheets for Developers

Boost Your Python Proficiency with 100 Comprehensive Code Examples and...

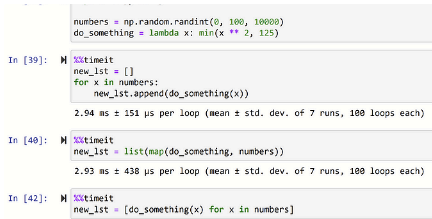
✦ · 17 min read · Feb 4, 2024

582

See all from Ravi | Python ✍️

See all from DataDrivenInvestor

Recommended from Medium



Florian Huber

A common Python myth, or why you don't need map, filter, and...

I very frequently teach Python. A lot. I teach Python to bachelor students, to master...

3 min read · Jan 25, 2024

173 5

Neelam Yadav

Ultimate Python Cheat Sheet: Practical Python For Everyday...

This Cheat Sheet is a product of necessity. Tasked with re-engaging with a Python...

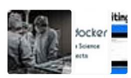
33 min read · Apr 13, 2024

506 1

Lists

General Coding Knowledge

20 stories · 1130 saves

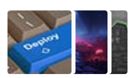


Coding & Development

11 stories · 572 saves

Stories to Help You Grow as a Software Developer

19 stories · 990 saves



Predictive Modeling w/ Python

20 stories · 1113 saves

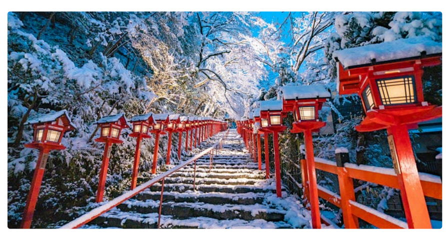
Jake Page

The guide to Git I never had.

🩺 Doctors have stethoscopes.

13 min read · Apr 11, 2024

2.1K 22



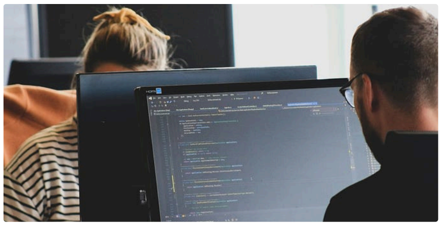
Yang Zhou in TechToFreedom

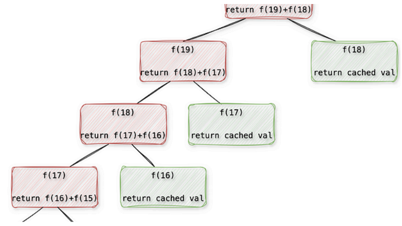
9 Advanced Python Type Hints That Will Improve Your Code...

Elegance, one step further

✦ · 9 min read · Apr 14, 2024

396 3





Builescu Daniel in Python in Plain English

Don't Repeat My Mistakes: Advanced Python Pitfalls to Avoid

Level up your Python: Lessons from an ex-Googler

· 8 min read · Apr 12, 2024

620 3

Christopher Tao in Towards Data Science

How to Use Python Built-In Decoration to Improve...

How to implement a caching mechanism in Python, and when not to use it?

· 9 min read · 6 days ago

898 2

See more recommendations