

Skills for today's data-driven world
Earn your Northwestern master's degree online. Northwestern | DATA SCIENCE
Events Earn your degree entirely online. Resources Cheat Sheets Recommendations Tech Briefs Advertise

Stop Writing Messy Python: A Clean Code Crash Course

Writing Python that works is easy. But writing Python that's clean, readable, and maintainable? That's what this crash course is for.

By **Bala Priya C**, KDnuggets Contributing Editor & Technical Content Specialist on June 11, 2025 in **Python**



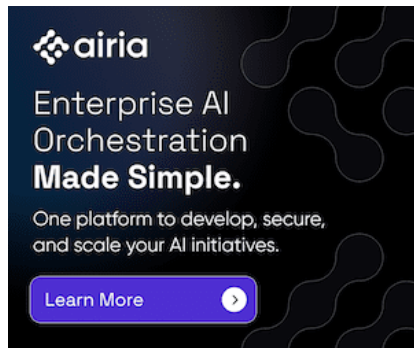
Image by Author | Ideogram

If you've been coding in Python for a while, you've probably mastered the basics, built a few projects. And now you're looking at your code thinking: "This works, but... it's not exactly something I'd proudly show in a code review." We've all been there.

Latest Posts

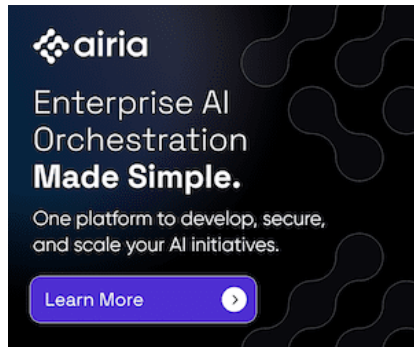
- Bridging the Gap: New Datasets Push Recommender Research Toward Real-World Scale
- Top 7 MCP Clients for AI Tooling
- Why You Need RAG to Stay Relevant as a Data Scientist
- Stop Writing Messy Python: A Clean Code Crash Course
- Selling Your Side Project? 10 Marketplaces Data Scientists Need to Know
- Integrating DuckDB & Python: An Analytics Guide

Top Posts



[Schedule a demo today.](#)

But as you keep coding, writing clean code becomes as important as writing functional code. In this article, I've compiled practical techniques that can help you go from "it runs, don't touch it" to "this is actually maintainable."



[Schedule a demo today.](#)

 [Link to the code on GitHub](#)

1. Model Data Explicitly. Don't Pass Around Dicts

Dictionaries are super flexible in Python and that's precisely the problem. When you pass around raw dictionaries throughout your code, you're inviting typos, key errors, and confusion about what data should actually be present.

Instead of this:

```
def process_user(user_dict):
    if user_dict['status'] == 'active': # What if 'status' is missing?
        send_email(user_dict['email']) # What if it's 'mail' in some places?

    # Is it 'name', 'full_name', or 'username'? Who knows!
```

7 Cool Python Projects to Automate the Boring Stuff

Run the Full DeepSeek-R1-0528 Model Locally

5 Error Handling Patterns in Python (Beyond Try-Except)

Building Data Science Projects Using AI: A Vibe Coding Guide

Top 5 Alternative Data Career Paths and How to Learn Them for Free

Unlocking Your Data to AI Platform: Generative AI for Multimodal Analytics

10 Awesome OCR Models for 2025

10 Generative AI Key Concepts Explained

5 Powerful Ways to Use Claude 4

Implementing Machine Learning Pipelines with Apache Spark



Get the FREE ebook 'The Great Big Natural Language Processing Primer' and 'The Complete Collection of Data Science Cheat Sheets' along with the leading newsletter on Data Science, Machine Learning, AI & Analytics straight to your inbox.

Your Email

SIGN UP

By subscribing you accept KDnuggets Privacy Policy

```
log_activity(f"Processed {user_dict['name']}")
```

This code is not robust because it assumes dictionary keys exist without validation. It offers no protection against typos or missing keys, which will cause `KeyError` exceptions at runtime. There's also no documentation of what fields are expected.

Do this:

```
from dataclasses import dataclass
from typing import Optional

@dataclass
class User:
    id: int
    email: str
    full_name: str
    status: str
    last_login: Optional[datetime] = None

def process_user(user: User):
    if user.status == 'active':
        send_email(user.email)
        log_activity(f"Processed {user.full_name}")
```

Python's `@dataclass` decorator gives you a clean, explicit structure with minimal boilerplate. Your IDE can now provide autocomplete for attributes, and you'll get immediate errors if required fields are missing.

For more complex validation, consider Pydantic:

```
from pydantic import BaseModel, EmailStr, validator

class User(BaseModel):
    id: int
    email: EmailStr # Validates email format
    full_name: str
    status: str

    @validator('status')
    def status_must_be_valid(cls, v):
        if v not in {'active', 'inactive', 'pending'}:
            raise ValueError('Must be active, inactive or pending')
        return v
```

Now your data validates itself, catches errors early, and documents expectations clearly.

2. Use Enums for Known Choices

String literals are prone to typos and provide no IDE autocomplete. The validation only happens at runtime.

Instead of this:

```
def process_order(order, status):
```

```

    if status == 'pending':
        # process logic
    elif status == 'shipped':
        # different logic
    elif status == 'delivered':
        # more logic
    else:
        raise ValueError(f"Invalid status: {status}")

# Later in your code...
process_order(order, 'shiped') # Typo! But no IDE warning

```

Do this:

```

from enum import Enum, auto

class OrderStatus(Enum):
    PENDING = 'pending'
    SHIPPED = 'shipped'
    DELIVERED = 'delivered'

def process_order(order, status: OrderStatus):
    if status == OrderStatus.PENDING:
        # process logic
    elif status == OrderStatus.SHIPPED:
        # different logic
    elif status == OrderStatus.DELIVERED:
        # more logic

# Later in your code...
process_order(order, OrderStatus.SHIPPED) # IDE autocomplete helps!

```

When you're dealing with a fixed set of options, an Enum makes your code more robust and self-documenting.

With enums:

- Your IDE provides autocomplete suggestions
- Typos become (almost) impossible
- You can iterate through all possible values when needed

Enum creates a set of named constants. The type hint `status: OrderStatus` documents the expected parameter type. Using `OrderStatus.SHIPPED` instead of a string literal allows IDE autocomplete and catches typos at development time.

3. Use Keyword-Only Arguments for Clarity

Python's flexible argument system is powerful, but it can lead to confusion when function calls have multiple optional parameters.

Instead of this:

```

def create_user(name, email, admin=False, notify=True, temporary=False):
    # Implementation

# Later in code...

```

```
create_user("John Smith", "john@example.com", True, False)
```

Wait, what do those booleans mean again?

When called with positional arguments, it's unclear what the boolean values represent without checking the function definition. Is True for admin, notify, or something else?

Do this:

```
def create_user(name, email, *, admin=False, notify=True, temporary=False):
    # Implementation

# Now you must use keywords for optional args
create_user("John Smith", "john@example.com", admin=True, notify=False)
```

The *, syntax forces all arguments after it to be specified by keyword. This makes your function calls self-documenting and prevents the "mystery boolean" problem where readers can't tell what True or False refers to without reading the function definition.

This pattern is especially useful in API calls and the like, where you want to ensure clarity at the call site.

4. Use Pathlib Over os.path

Python's os.path module is functional but clunky. The newer pathlib module provides an object-oriented approach that's more intuitive and less error-prone.

Instead of this:

```
import os

data_dir = os.path.join('data', 'processed')
if not os.path.exists(data_dir):
    os.makedirs(data_dir)

filepath = os.path.join(data_dir, 'output.csv')
with open(filepath, 'w') as f:
    f.write('results\n')

# Check if we have a JSON file with the same name
json_path = os.path.splitext(filepath)[0] + '.json'
if os.path.exists(json_path):
    with open(json_path) as f:
        data = json.load(f)
```

This uses string manipulation with os.path.join() and os.path.splitext() for path handling. Path operations are scattered across different functions. The code is verbose and less intuitive.

Do this:

```
from pathlib import Path
```

```

data_dir = Path('data') / 'processed'
data_dir.mkdir(parents=True, exist_ok=True)

filepath = data_dir / 'output.csv'
filepath.write_text('results\n')

# Check if we have a JSON file with the same name
json_path = filepath.with_suffix('.json')
if json_path.exists():
    data = json.loads(json_path.read_text())

```

Why pathlib is better:

- Path joining with / is more intuitive
- Methods like `mkdir()`, `exists()`, and `read_text()` are attached to the path object
- Operations like changing extensions (`with_suffix()`) are more semantic

Pathlib handles the subtleties of path manipulation across different operating systems. This makes your code more portable and robust.

5. Fail Fast with Guard Clauses

Deeply nested if-statements are often hard to understand and maintain. Using early returns — guard clauses — leads to more readable code.

Instead of this:

```

def process_payment(order, user):
    if order.is_valid:
        if user.has_payment_method:
            payment_method = user.get_payment_method()
            if payment_method.has_sufficient_funds(order.total):
                try:
                    payment_method.charge(order.total)
                    order.mark_as_paid()
                    send_receipt(user, order)
                    return True
                except PaymentError as e:
                    log_error(e)
                    return False
            else:
                log_error("Insufficient funds")
                return False
        else:
            log_error("No payment method")
            return False
    else:
        log_error("Invalid order")
        return False

```

Deep nesting is hard to follow. Each conditional block requires tracking multiple branches simultaneously.

Do this:

```
def process_payment(order, user):
    # Guard clauses: check preconditions first
    if not order.is_valid:
        log_error("Invalid order")
        return False

    if not user.has_payment_method:
        log_error("No payment method")
        return False

    payment_method = user.get_payment_method()
    if not payment_method.has_sufficient_funds(order.total):
        log_error("Insufficient funds")
        return False

    # Main logic comes after all validations
    try:
        payment_method.charge(order.total)
        order.mark_as_paid()
        send_receipt(user, order)
        return True
    except PaymentError as e:
        log_error(e)
        return False
```

Guard clauses handle error cases up front, reducing indentation levels. Each condition is checked sequentially, making the flow easier to follow. The main logic comes at the end, clearly separated from error handling.

This approach scales much better as your logic grows in complexity.

6. Don't Overuse List Comprehensions

List comprehensions are one of Python's most elegant features, but they become unreadable when overloaded with complex conditions or transformations.

Instead of this:

```
# Hard to parse at a glance
active_premium_emails = [user['email'] for user in users_list
                          if user['status'] == 'active' and
                          user['subscription'] == 'premium' and
                          user['email_verified'] and
                          not user['email'] in blacklisted_domains]
```

This list comprehension packs too much logic into one line. It's hard to read and debug. Multiple conditions are chained together, making it difficult to understand the filter criteria.

Do this:

Here are better alternatives.

Option 1: Function with a descriptive name

Extracts the complex condition into a named function with a descriptive name. The list comprehension is now much clearer, focusing on what it's doing (extracting emails) rather

than how it's filtering.

```
def is_valid_premium_user(user):
    return (user['status'] == 'active' and
            user['subscription'] == 'premium' and
            user['email_verified'] and
            not user['email'] in blacklisted_domains)

active_premium_emails = [user['email'] for user in users_list if is_valid_prem
```

Option 2: Traditional loop when logic is complex

Uses a traditional loop with early continues for clarity. Each condition is checked separately, making it easy to debug which condition might be failing. The transformation logic is also clearly separated.

```
active_premium_emails = []
for user in users_list:
    # Complex filtering logic
    if user['status'] != 'active':
        continue
    if user['subscription'] != 'premium':
        continue
    if not user['email_verified']:
        continue
    if user['email'] in blacklisted_domains:
        continue

    # Complex transformation logic
    email = user['email'].lower().strip()
    active_premium_emails.append(email)
```

List comprehensions should make your code more readable, not less. When the logic gets complex:

- Break complex conditions into named functions
- Consider using a regular loop with early continues
- Split complex operations into multiple steps

Remember, the goal is readability.

7. Write Reusable Pure Functions

A function is a pure function if it produces the same output for the same inputs always.

Also, it has no side effects.

Instead of this:

```
total_price = 0 # Global state

def add_item_price(item_name, quantity):
    global total_price
    # Look up price from global inventory
    price = inventory.get_item_price(item_name)
    # Apply discount
```



```

    if settings.discount_enabled:
        price *= 0.9
    # Update global state
    total_price += price * quantity

# Later in code...
add_item_price('widget', 5)
add_item_price('gadget', 3)
print(f"Total: ${total_price:.2f}")

```

This uses global state (`total_price`) which makes testing difficult.

The function has side effects (modifying global state) and depends on external state (inventory and settings). This makes it unpredictable and hard to reuse.

Do this:

```

def calculate_item_price(item, price, quantity, discount=0):
    """Calculate final price for a quantity of items with optional discount.

    Args:
        item: Item identifier (for logging)
        price: Base unit price
        quantity: Number of items
        discount: Discount as decimal

    Returns:
        Final price after discounts
    """
    discounted_price = price * (1 - discount)
    return discounted_price * quantity

def calculate_order_total(items, discount=0):
    """Calculate total price for a collection of items.

    Args:
        items: List of (item_name, price, quantity) tuples
        discount: Order-level discount

    Returns:
        Total price after all discounts
    """
    return sum(
        calculate_item_price(item, price, quantity, discount)
        for item, price, quantity in items
    )

# Later in code...
order_items = [
    ('widget', inventory.get_item_price('widget'), 5),
    ('gadget', inventory.get_item_price('gadget'), 3),
]

total = calculate_order_total(order_items,
                             discount=0.1 if settings.discount_enabled else 0)
print(f"Total: ${total:.2f}")

```

The following version uses pure functions that take all dependencies as parameters.

8. Write Docstrings for Public Functions and Classes

Documentation isn't (and shouldn't be) an afterthought. It's a core part of maintainable

code. Good docstrings explain not just what functions do, but why they exist and how to use them correctly.

Instead of this:

```
def celsius_to_fahrenheit(celsius):  
    """Convert Celsius to Fahrenheit."""  
    return celsius * 9/5 + 32
```

This is a minimal docstring that only repeats the function name. Provides no information about parameters, return values, or edge cases.

Do this:

```
def celsius_to_fahrenheit(celsius):  
    """  
    Convert temperature from Celsius to Fahrenheit.  
    The formula used is:  $F = C \times (9/5) + 32$   
    Args:  
    celsius: Temperature in degrees Celsius (can be float or int)  
    Returns:  
    Temperature converted to degrees Fahrenheit  
    Example:  
    >>> celsius_to_fahrenheit(0)  
    32.0  
    >>> celsius_to_fahrenheit(100)  
    212.0  
    >>> celsius_to_fahrenheit(-40)  
    -40.0  
    """  
    return celsius * 9/5 + 32
```

A good docstring:

- Documents parameters and return values
- Notes any exceptions that might be raised
- Provides usage examples

Your docstrings serve as executable documentation that stays in sync with your code.

9. Automate Linting and Formatting

Don't rely on manual inspection to catch style issues and common bugs. Automated tools can handle the tedious work of ensuring code quality and consistency.

You can try setting up these linting and formatting tools:

1. **Black** - Code formatter
2. **Ruff** - Fast linter
3. **mypy** - Static type checker

4. **isort** - Import organizer

Integrate them using pre-commit hooks to automatically check and format code before each commit:

1. Install pre-commit: `pip install pre-commit`
2. Create a `.pre-commit-config.yaml` file with the tools configured
3. Run `pre-commit install` to activate

This setup ensures consistent code style and catches errors early without manual effort.

You can check [7 Tools To Help Write Better Python Code](#) to know more on this.

10. Avoid Catch-All except

Generic exception handlers hide bugs and make debugging difficult. They catch everything, including syntax errors, memory errors, and keyboard interrupts.

Instead of this:

```
try:
    user_data = get_user_from_api(user_id)
    process_user_data(user_data)
    save_to_database(user_data)
except:
    # What failed? We'll never know!
    logger.error("Something went wrong")
```

This uses a bare exception to handle:

- Programming errors (like syntax errors)
- System errors (like MemoryError)
- Keyboard interrupts (Ctrl+C)
- Expected errors (like network timeouts)

This makes debugging extremely difficult, as all errors are treated the same.

Do this:

```
try:
    user_data = get_user_from_api(user_id)
    process_user_data(user_data)
    save_to_database(user_data)
except ConnectionError as e:
    logger.error(f"API connection failed: {e}")
    # Handle API connection issues
except ValueError as e:
    logger.error(f"Invalid user data received: {e}")
    # Handle validation issues
except DatabaseError as e:
    logger.error(f"Database error: {e}")
    # Handle database issues
```

```
except Exception as e:
    # Last resort for unexpected errors
    logger.critical(f"Unexpected error processing user {user_id}: {e}",
                    exc_info=True)
    # Possibly re-raise or handle generically
    raise
```

Catches specific exceptions that can be expected and handled appropriately. Each exception type has its own error message and handling strategy.

The final `except Exception` catches unexpected errors, logs them with full traceback (`exc_info=True`), and re-raises them to avoid silently ignoring serious issues.

If you do need a catch-all handler for some reason, use `except Exception as e:` rather than a bare `except:`, and always log the full exception details with `exc_info=True`.

Wrapping Up

I hope you get to use at least some of these practices in your code. Start implementing them in your projects.

You'll find your code becoming more maintainable, more testable, and easier to reason about.

Next time you're tempted to take a shortcut, remember: **code is read many more times than it's written**. Happy clean coding?

Bala Priya C is a developer and technical writer from India. She likes working at the intersection of math, programming, data science, and content creation. Her areas of interest and expertise include DevOps, data science, and natural language processing. She enjoys reading, writing, coding, and coffee! Currently, she's working on learning and sharing her knowledge with the developer community by authoring tutorials, how-to guides, opinion pieces, and more. Bala also creates engaging resource overviews and coding tutorials.

More On This Topic

- [Free Intermediate Python Programming Crash Course](#)
- [Unlock Your Potential with This FREE DevOps Crash Course](#)
- [How to Write Clean Python Code as a Beginner](#)
- [Mastering Python: 7 Strategies for Writing Clear, Organized, and...](#)
- [Tips for Writing Better Unit Tests for Your Python Code](#)

- [Data Cleaning in SQL: How To Prepare Messy Data for Analysis](#)



Get the FREE ebook 'The Great Big Natural Language Processing Primer' and 'The Complete Collection of Data Science Cheat Sheets' along with the leading newsletter on Data Science, Machine Learning, AI & Analytics straight to your inbox.


Your Email


SIGN UP


By subscribing you accept KDnuggets Privacy Policy


What do you think?


3 Responses



Upvote


Funny


Love


Surprised


Angry


Sad

1 Comment

Login ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

♡

Share

BestNewestOldest

V

VladimirJosephStephanOrlovsky

an hour ago

—

🚩

celsius_to_fahrenheit(-888)

00ReplyShare ▾

[<= Previous post](#)

[Next post =>](#)