Python Function Design and Best Practices

What Is Function Design?

Writing a function is not only about making code work.

Good function design means creating functions that are:

- easy to understand
- easy to test
- easy to reuse
- easy to modify
- predictable
- focused on one clear responsibility

Compare:

```
def process_user(user):
    # validate user
    # save user
    # send email
    # generate report
    # update dashboard
    ...
```

This function is responsible for too many things.

A better design separates those responsibilities:

```
def validate_user(user):
    ...
```

```
def save_user(user):
    ...

def send_welcome_email(user):
    ...

def generate_user_report(user):
    ...
```

Each function now has a clear purpose.

Why Function Design Matters

Poorly designed functions create problems as projects grow.

A function may become:

- difficult to understand
- difficult to test
- difficult to reuse
- difficult to debug
- difficult to change safely

Good function design reduces these problems.

Think of a function as a small machine:

```
Input
  
[ Function ]
  
```

`Output`

A well-designed function has a clear answer to:

> What goes in, what happens, and what comes out?

Principle 1: Give Every Function One Clear Responsibility

A function should ideally do one meaningful job.

Instead of:

```
def process_order(order):
    calculate_total(order)
    save_order(order)
    send_email(order)
    update_inventory(order)
```

you can separate the responsibilities:

```
def calculate_order_total(order):
    ...

def save_order(order):
    ...

def send_order_confirmation(order):
    ...

def update_inventory(order):
```

```
    ...
```

Then a higher-level function can coordinate them:

```
def process_order(order):
    total = calculate_order_total(order)
    save_order(order)
    update_inventory(order)
    send_order_confirmation(order)

    return total
```

The main function coordinates the process while smaller functions handle individual tasks.

Single Responsibility

A useful question is:

> Can I describe what this function does in one clear sentence?

Good:

```
def calculate_total(items):
    """Calculate the total price of items."""
```

Good:

```
def send_email(email, message):
    """Send an email message."""
```

Less clear:

```
def process_everything(data):
    ...
```

If you need a long explanation to describe a function, it may be doing too much.

Principle 2: Give Functions Clear Names

A function name should communicate its purpose.

Good:

```
calculate_total()
get_user()
validate_email()
send_message()
save_resource()
filter_products()
```

Poor:

```
do_it()
process()
handle()
thing()
data()
function1()
```

The name should help someone understand the code before reading its implementation.

Use Verbs for Functions

Functions usually represent actions, so verbs are useful.

`get_user()`
`create_account()`
`delete_file()`
`calculate_tax()`
`validate_password()`

Compare:

`user()`

with:

`get_user()`

The second name tells us what the function actually does.

Principle 3: Keep Functions Focused

A function does not have to be extremely short.

The goal is not:

> Make every function five lines long.

The goal is:

> Keep each function focused on one meaningful responsibility.

For example:

```
def calculate_discount(price, percentage):
    discount = price * percentage / 100
    return price - discount
```

This function has one clear job.

Avoid Giant Functions

Consider:

```
def create_account(user):
    # validate name
    # validate email
    # validate password
    # connect to database
    # save user
    # create session
    # send email
    # generate notification
    # update analytics
    # ...
```

This becomes difficult to maintain.

Instead:

```
def create_account(user):
    validate_user(user)
    save_user(user)
    create_session(user)
    send_welcome_email(user)
    track_signup(user)
```

The high-level flow becomes much easier to understand.

Principle 4: Choose Clear Parameters

Parameters should communicate what the function needs.

Good:

```
def calculate_shipping(weight, distance):
    ...
```

Less useful:

```
def calculate_shipping(x, y):
    ...
```

When someone calls:

```
calculate_shipping(5, 20)
```

the first version is easier to understand.

Avoid Too Many Parameters

This can become difficult to use:

```
def create_user(
    name,
    email,
    age,
    country,
    city,
    phone,
    address,
    role,
    active
):
    ...
```

Too many parameters may indicate that a data structure or class would be more appropriate.

For example:

```
user = {
    "name": "Hafsa",
    "email": "hafsa@example.com",
    "age": 25,
    "country": "Pakistan",
    "city": "Lahore"
}
```

Then:

```
def create_user(user):
```

```
    ...
```

The exact design depends on the application, but the principle is:

> A function should require the information it actually needs.

Principle 5: Use Parameters Instead of Hardcoding Values

Avoid:

```
def calculate_discount(price):
    return price * 0.80
```

What does `0.80` mean?

A better design:

```
def calculate_discount(price, discount):
    return price * (1 - discount / 100)
```

Now:

```
calculate_discount(1000, 20)
```

The function is reusable.

Hardcoded Logic vs Reusable Logic

Poor:

```
def calculate_student_fee():
    return 50000
```

Better:

```
def calculate_fee(base_fee, discount):
    return base_fee - (base_fee * discount / 100)
```

Now the same function can work with different values.

Principle 6: Return Values Instead of Printing

A reusable function usually should return data rather than print it.

Less reusable:

```
def calculate_total(a, b):
    print(a + b)
```

Better:

```
def calculate_total(a, b):
    return a + b
```

Now the caller decides what to do with the result:

```
total = calculate_total(100, 50)

print(total)
```

Or:

```
total = calculate_total(100, 50)

if total > 100:
    print("Large order")
```

Or:

```
total = calculate_total(100, 50)

save_total(total)
```

The function is more flexible.

`return` vs `print`

Remember:

`print()`

displays something.

`return`

gives a value back to the caller.

Example:

```
def add(a, b):
    return a + b
```

Then:

```
result = add(5, 3)
```

Now `result` contains:

```
8
```

Principle 7: Avoid Unnecessary Side Effects

A side effect occurs when a function changes something outside its local calculation.

Examples include:

- modifying global variables
- writing to files
- changing database records
- sending emails
- printing output
- changing shared objects

For example:

```
total = 0

def add_to_total(value):
    global total
    total += value
```

This function changes external state.

A more predictable function is:

```
def add(a, b):
  return a + b
```

The result depends only on its inputs.

Pure Functions

A pure function generally:

1. produces the same result for the same input
2. does not modify external state

Example:

```
def square(number):
  return number * number
```

If you call:

```
square(5)
```

you always get:

```
25
```

Pure functions are often easier to test and reason about.

Not Every Function Must Be Pure

Real applications need side effects.

For example:

```
def save_user(user):
    database.save(user)
```

Saving to a database is a side effect, but it is a legitimate responsibility for this function.

The goal is not to eliminate side effects.

The goal is to keep them **clear and controlled**.

Principle 8: Avoid Global Variables

Global state can make functions difficult to understand.

Example:

```
tax_rate = 0.10

def calculate_total(price):
    return price + price * tax_rate
```

The function depends on a value defined somewhere else.

A clearer version can be:

```
def calculate_total(price, tax_rate):
    return price + price * tax_rate
```

Now the dependency is explicit.

Principle 9: Use Default Arguments Carefully

Default arguments can make functions convenient.

```
def greet(name, message="Hello"):
    return f"{message}, {name}!"
```

Now:

```
greet("Hafsa")
```

produces:

```
Hello, Hafsa!
```

And:

```
greet("Hafsa", "Welcome")
```

produces:

```
Welcome, Hafsa!
```

Use defaults when there is a sensible default behavior.

Be Careful With Mutable Defaults

Avoid:

```
def add_item(item, items=[]):
    items.append(item)
    return items
```

The same list can be reused across calls.

Instead:

```
def add_item(item, items=None):
    if items is None:
        items = []

    items.append(item)
    return items
```

This creates a new list when no list is provided.

Principle 10: Keep Input and Output Predictable

A function should behave in a way users of the function can reasonably expect.

For example:

```
def calculate_total(items):
    return sum(items)
```

If it sometimes returns a number and sometimes returns a string:

```
def calculate_total(items):
    if not items:
        return "No items"

    return sum(items)
```

the caller has to handle different types.

A more predictable design might be:

```
def calculate_total(items):
    return sum(items)
```

An empty list naturally produces:

```
0
```

Predictable outputs make functions easier to use.

Principle 11: Validate Important Inputs

Functions should protect themselves from invalid input when necessary.

Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("b cannot be zero")
```

```
    return a / b
```

Now invalid input produces a clear error.

Validation Should Match the Responsibility

Do not make every function validate everything.

For example:

```
def calculate_square(number):
    return number * number
```

There may be no need for complicated validation.

But if a function requires an email:

```
def register_user(email):
    if "@" not in email:
        raise ValueError("Invalid email address")

    ...
```

Validation should be appropriate to the function's role.

Principle 12: Use Exceptions for Exceptional Situations

Avoid using special return values to hide errors.

Less clear:

```
def divide(a, b):
    if b == 0:
        return None

    return a / b
```

The caller has to know that `None` means an error.

A clearer design may be:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")

    return a / b
```

Now the error is explicit.

Principle 13: Avoid Deep Nesting

Consider:

```
def process_user(user):
    if user:
        if user["active"]:
            if user["email"]:
                if "@" in user["email"]:
                    return "Valid"
```

This is difficult to read.

You can simplify it with early returns:

```
def process_user(user):
    if not user:
        return False

    if not user["active"]:
        return False

    if not user["email"]:
        return False

    if "@" not in user["email"]:
        return False

    return True
```

The logic becomes easier to follow.

Early Returns

An early return exits a function as soon as a condition is met.

Example:

```
def get_discount(user):
    if not user:
        return 0
```

```
    if not user["active"]:
        return 0

    return 10
```

This can be easier to understand than deeply nested conditions.

Principle 14: Avoid Duplicate Logic

Suppose you write:

```
def calculate_student_total(price):
    return price + price * 0.10

def calculate_teacher_total(price):
    return price + price * 0.10
```

The same logic appears twice.

Create one reusable function:

```
def calculate_total(price, tax):
    return price + price * tax
```

Then:

```
calculate_total(1000, 0.10)
```

Avoiding unnecessary duplication makes future changes easier.

Don't Over-Abstract

There is also a danger of creating too many tiny functions.

For example:

```
def add(a, b):
    return a + b

def calculate_sum(a, b):
    return add(a, b)

def get_result(a, b):
    return calculate_sum(a, b)
```

This creates unnecessary layers.

Good design is not:

> More functions = better code.

Good design is:

> Functions should provide useful boundaries around meaningful responsibilities.

Principle 15: Keep Abstraction at the Right Level

Consider:

```
def process_order(order):
    validate_order(order)
    calculate_total(order)
    save_order(order)
```

This is useful because each function represents a meaningful operation.

But creating functions such as:

```
def get_price(order):
    ...

def store_price(price):
    ...

def return_price(price):
    ...
```

may add complexity without providing useful abstraction.

Ask:

> Does this function make the system easier to understand?

If not, the abstraction may not be necessary.

Principle 16: Use Docstrings

A well-designed function should be understandable.

Docstrings help communicate its purpose.

```
def calculate_total(items):
    """
    Calculate the total price of all items.

    Args:
        items: A list of item prices.

    Returns:
        The total price.
    """
    return sum(items)
```

Documentation is especially important for reusable or public functions.

Principle 17: Use Type Hints

Type hints communicate expected data types.

```
def add(a: int, b: int) -> int:
    return a + b
```

This tells the reader:

```
a □ integer
b □ integer
return □ integer
```

More descriptive code:

```
def calculate_total(prices: list[float]) -> float:
    return sum(prices)
```

Type hints improve readability and can help development tools detect mistakes.

Principle 18: Don't Make Functions Do Hidden Things

Consider:

```
def calculate_total(items):
    send_email()
    save_to_database()
    return sum(items)
```

The function name suggests calculation, but it secretly performs other operations.

This makes the function surprising.

Better:

```
def calculate_total(items):
    return sum(items)
```

And separately:

```
def save_order(order):
    ...
```

The function's name and behavior should agree.

Principle 19: Design Functions Around Meaningful Data

Suppose you have:

```
def calculate_total(price, shipping, tax, discount):
    ...
```

This may be perfectly reasonable.

But if the same group of values is repeatedly passed around, a data structure may make the design clearer:

```
order = {
    "price": 1000,
    "shipping": 200,
    "tax": 100,
    "discount": 50
}
```

Then:

```
def calculate_order_total(order):
    ...
```

The appropriate design depends on the application.

The key idea is:

> Group related information when it represents one meaningful concept.

Function Composition

Well-designed functions can be combined.

Example:

```
def get_prices(items):
    return [item["price"] for item in items]


def calculate_total(prices):
    return sum(prices)


def apply_discount(total, discount):
    return total * (1 - discount / 100)
```

Now:

```
prices = get_prices(items)
total = calculate_total(prices)
final_total = apply_discount(total, 10)
```

Each function performs one useful step.

Composition in One Flow

The same process can be organized into a higher-level function:

```
def calculate_final_total(items, discount):
    prices = get_prices(items)
```

```
    total = calculate_total(prices)
    return apply_discount(total, discount)
```

This is a powerful design pattern:

Small focused functions

Composition

Larger useful behavior

Real World Example: haas.dev Resources

Imagine a resource system with:

```
resources = [
    {
        "title": "Python Functions",
        "category": "Python",
        "free": True
    },
    {
        "title": "React Fundamentals",
        "category": "React",
        "free": False
    },
    {
        "title": "Python Lists",
        "category": "Python",
        "free": True
    }
]
```

Instead of one large function:

```
def process_resources(resources, keyword):
    # filter
    # search
    # sort
    # format
    # calculate count
    ...
```

create focused functions:

```
def filter_free_resources(resources):
    """Return resources that are free."""
    return [
        resource
        for resource in resources
        if resource["free"]
    ]


def search_resources(resources, keyword):
    """Return resources whose titles contain the keyword."""
    keyword = keyword.lower()

    return [
        resource
        for resource in resources
        if keyword in resource["title"].lower()
    ]


def sort_resources(resources):
    """Return resources sorted alphabetically by title."""
```

```
    return sorted(resources, key=lambda resource: resource["title"])
```

Now each function can be tested independently.

Testing Becomes Easier

Focused functions are easier to test.

For example:

```
def calculate_total(prices):
    return sum(prices)
```

You can test:

```
assert calculate_total([10, 20, 30]) == 60
assert calculate_total([]) == 0
```

Because the function has one responsibility, the expected behavior is clear.

Function Design and Testing

Poor design:

```
def process_order(order):
    # calculate
    # save database
    # send email
    # update inventory
```

```
    # ...
```

Testing this function requires many systems.

Better:

```
def calculate_order_total(order):
    ...

def update_inventory(order):
    ...

def send_confirmation(order):
    ...
```

Each part can be tested separately.

Function Design and Debugging

Suppose something goes wrong in:

```
def process_order(order):
    validate_order(order)
    calculate_total(order)
    save_order(order)
    send_confirmation(order)
```

The problem can be isolated to a specific operation.

If everything exists inside one giant function, debugging becomes harder.

Small meaningful boundaries make problems easier to locate.

Function Design and Reusability

A reusable function should not depend on unnecessary external information.

Poor:

```
tax_rate = 0.10

def calculate_tax(price):
    return price * tax_rate
```

Better:

```
def calculate_tax(price, tax_rate):
    return price * tax_rate
```

Now the function can work in different situations.

A Function Design Checklist

Before considering a function finished, ask:

Purpose

- Does it have one clear responsibility?
- Can I describe its job in one sentence?

Naming

- Does the name clearly describe the action?
- Is it specific enough?

Parameters

- Does it receive only the data it needs?
- Are the parameter names clear?
- Are there too many parameters?

Output

- Does it return a useful result?
- Is the return type predictable?

Side Effects

- Does it modify external state?
- Are those side effects intentional and obvious?

Errors

- What happens with invalid input?
- Should the function raise an exception?

Reusability

- Is important logic hardcoded unnecessarily?
- Can the function be reused elsewhere?

Readability

- Is the implementation easy to follow?
- Is there unnecessary nesting or duplication?

Documentation

- Does the function need a docstring?
- Would type hints improve clarity?

A Function Design Workflow

Use this process when creating a new function.

Step 1: Define the Job

Write one sentence:

`Calculate the total price of an order.`

Step 2: Identify Inputs

Ask:

`What information does the function need?`

Example:

`items`

Step 3: Identify the Output

Ask:

`What should the function give back?`

Example:

total price

Step 4: Define Edge Cases

Ask:

What happens if the input is empty?
What happens if the value is invalid?

Step 5: Write the Function

```
def calculate_total(items):
    return sum(items)
```

Step 6: Add Documentation if Needed

```
def calculate_total(items):
    """Return the total price of all items."""
    return sum(items)
```

Step 7: Test It

```
assert calculate_total([10, 20, 30]) == 60
```

Before and After

Poor Design

```
def process_user(user):
    if user:
        if user["email"]:
            if "@" in user["email"]:
                user["name"] = user["name"].strip()
```

```
        user["active"] = True
        print("User is valid")
        save_to_database(user)
        send_email(user)
```

Problems:

- validation
- data modification
- printing
- database work
- email sending

all happen in one function.

Improved Design

```
def validate_email(email):
    """Return True if the email appears valid."""
    return bool(email and "@" in email)


def clean_name(name):
    """Return a cleaned user name."""
    return name.strip()


def activate_user(user):
    """Mark a user as active."""
    user["active"] = True
```

```
def process_user(user):
    """Validate and prepare a user for registration."""
    if not validate_email(user["email"]):
        raise ValueError("Invalid email")

    user["name"] = clean_name(user["name"])
    activate_user(user)

    return user
```

Database and email operations can remain separate:

```
save_to_database(user)
send_email(user)
```

The responsibilities are now clearer.

When a Function Is Too Large

There is no universal number of lines that makes a function too large.

Instead, look for warning signs:

- multiple unrelated responsibilities
- repeated blocks of logic
- deeply nested conditions
- many parameters
- many temporary variables
- difficult testing
- difficult naming

- comments explaining large sections of the function
- frequent changes to unrelated parts

These signs suggest the function may need to be redesigned.

When a Function Is Too Small

Small functions are not automatically good.

This:

```
def get_name(user):
    return user["name"]
```

may be useful if it represents a meaningful abstraction.

But creating functions for every single trivial operation can make code harder to follow.

For example:

```
def add(a, b):
    return a + b

def call_add(a, b):
    return add(a, b)

def get_sum(a, b):
    return call_add(a, b)
```

There is no useful reason for all three layers.

Good function design balances:

```
Readability
+
Reusability
+
Simplicity
```

Five Important Rules

Remember these five rules when designing functions:

1. One clear responsibility

```
calculate_total()
```

2. Clear names

```
validate_email()
```

3. Explicit inputs and outputs

```
calculate_tax(price, rate)
```

4. Minimize unnecessary side effects

```
return total
```

instead of unexpectedly changing global state.

5. Keep behavior predictable

The function should do what its name and documentation suggest.

Mini Project: Order Processing System

Create a small order system using focused functions.

Step 1: Calculate total

```
def calculate_total(items):
    """Return the total price of all items."""
    return sum(items)
```

Step 2: Apply discount

```
def apply_discount(total, percentage):
    """Return the total after applying a discount."""
    return total * (1 - percentage / 100)
```

Step 3: Add tax

```
def add_tax(total, rate):
    """Return the total after applying tax."""
    return total * (1 + rate)
```

Step 4: Combine them

```
def calculate_final_price(items, discount, tax):
    """
    Calculate the final order price.

    Applies a discount and then adds tax.
    """
    total = calculate_total(items)
    total = apply_discount(total, discount)
    total = add_tax(total, tax)

```

```
    return total
```

Test:

```
items = [100, 200, 300]

final_price = calculate_final_price(
    items,
    discount=10,
    tax=0.05
)

print(final_price)
```

Notice how the larger operation is built from smaller, focused functions.

---

5 Minute Challenge

Create a function:

```
calculate_student_result(marks)
```

Requirements:

- calculate the average
- determine whether the student passed
- use at least two smaller helper functions
- return the result
- add useful docstrings
- avoid global variables

Possible structure:

```
def calculate_average(marks):
    """Return the average of the marks."""
    ...


def is_passing(average):
    """Return True if the average meets the passing score."""
    ...


def calculate_student_result(marks):
    """Calculate a student's average and pass status."""
    ...
```

The goal is not just to make it work.

The goal is to practice **designing functions with clear responsibilities**.

Exercises

Exercise 1: Refactor a Large Function

Take this:

```
def process_product(product):
    # validate product
    # calculate discount
    # calculate tax
    # save product
```

```
    # send notification
    ...
```

Break it into smaller meaningful functions.

Exercise 2: Improve Function Names

Improve these names:

```
def do_data():
    ...


def process():
    ...


def handle_user():
    ...


def thing():
    ...
```

Give each function a name that clearly communicates its responsibility.

Exercise 3: Return Instead of Print

Rewrite:

```
def calculate_total(a, b):
    print(a + b)
```

so that another function can use the result.

Exercise 4: Remove Global State

Rewrite:

```
tax_rate = 0.10

def calculate_tax(price):
    return price * tax_rate
```

so that the dependency is explicit.

Exercise 5: Reduce Duplication

Refactor:

```
def student_total(price):
    return price + price * 0.10


def teacher_total(price):
    return price + price * 0.10
```

into a reusable design.

Mini Quiz

1. What is a major goal of good function design?

A. Making every function as long as possible
B. Giving functions clear and focused responsibilities
C. Avoiding all parameters
D. Using global variables

Answer: B

2. Which is the clearest function name?

A. `do_it()`
B. `process()`
C. `calculate_total()`
D. `thing()`

Answer: C

3. Why are return values often preferred over printing inside reusable functions?

A. Printing is slower
B. Returned values can be used by other code
C. `return` automatically saves data
D. `print()` cannot work inside functions

Answer: B

4. What is a pure function generally expected to do?

A. Change global variables
B. Send emails
C. Produce the same output for the same input without external side effects
D. Always print something

Answer: C

5. What can too many parameters indicate?

A. The function is always correct
B. The function may be handling too much data or responsibility
C. Python does not allow parameters
D. The function should use more global variables

Answer: B

Interview Questions

1. What makes a function well designed?

A well-designed function generally has a clear responsibility, meaningful name, explicit inputs and outputs, predictable behavior, and minimal unnecessary side effects.

2. What does single responsibility mean for functions?

It means a function should focus on one meaningful responsibility rather than combining several unrelated tasks.

3. Why should functions generally return values instead of printing them?

Returning values allows the caller to decide what to do with the result. This makes the function more reusable and testable.

4. What is a pure function?

A pure function produces the same result for the same inputs and does not modify external state.

5. Why should global variables generally be avoided?

They create hidden dependencies and make functions harder to understand, test, and reuse.

6. What is function composition?

Function composition means building larger behavior by combining smaller functions with clear responsibilities.

7. How can you identify a function that is doing too much?

Warning signs include multiple unrelated responsibilities, deep nesting, many parameters, repeated code, difficult testing, and complicated explanations.

8. Should every function be very short?

No. Length alone is not the goal. A function should be as simple as reasonably possible while maintaining a meaningful responsibility.

Function Design Cheat Sheet

Good Function
□
□□□ One clear responsibility
□□□ Clear name
□□□ Meaningful parameters
□□□ Predictable return value
□□□ Minimal unnecessary side effects
□□□ Appropriate validation

□□□ Useful error handling
□□□ No unnecessary duplication
□□□ Useful abstraction
□□□ Docstring when appropriate
□□□ Type hints when useful

Good

```
def calculate_total(prices: list[float]) -> float:
    """Return the total price."""
    return sum(prices)
```

Less Good

```
total = 0

def process(data):
    global total

    # validate
    # calculate
    # save
    # print
    # send email
    # update database
    ...
```

Key Takeaways

- Function design is about more than making code execute correctly.
- Give each function a clear and meaningful responsibility.
- Use names that communicate what the function does.
- Pass dependencies explicitly instead of relying on global state.

- Prefer returning values when building reusable functions.
- Keep side effects intentional and easy to identify.
- Avoid unnecessary parameters.
- Avoid duplicate logic.
- Use early returns when they make conditional logic clearer.
- Validate important inputs.
- Use exceptions when invalid situations need to be reported.
- Use docstrings and type hints when they improve understanding.
- Do not create unnecessary abstractions.
- Do not make every function extremely short just for the sake of being short.
- Combine small, meaningful functions to build larger workflows.
- Good function design makes code easier to read, test, debug, reuse, and maintain.

Visit haas.dev for more resources and guides.

haas.dev

https://dev-roast-app.vercel.app