The Great Lie of Python's Type Hints

```
def process_data(user: User):
```

You write them. Your static analyser checks them. You feel safe.
But at runtime, Python completely ignores them.

The Real-World Cost of False Confidence

Silent Failures

Wrong types flow through your system completely undetected.

Delayed Errors

A crash occurs far from where the bad data was introduced.

Debugging Nightmares

Hours spent asking, “Where did this `None` come from?”

Security Vulnerabilities

Unchecked input reaches sensitive business logic.

Contract Violations

Functions return the wrong type, breaking their callers.

Making Type Hints a Runtime Reality

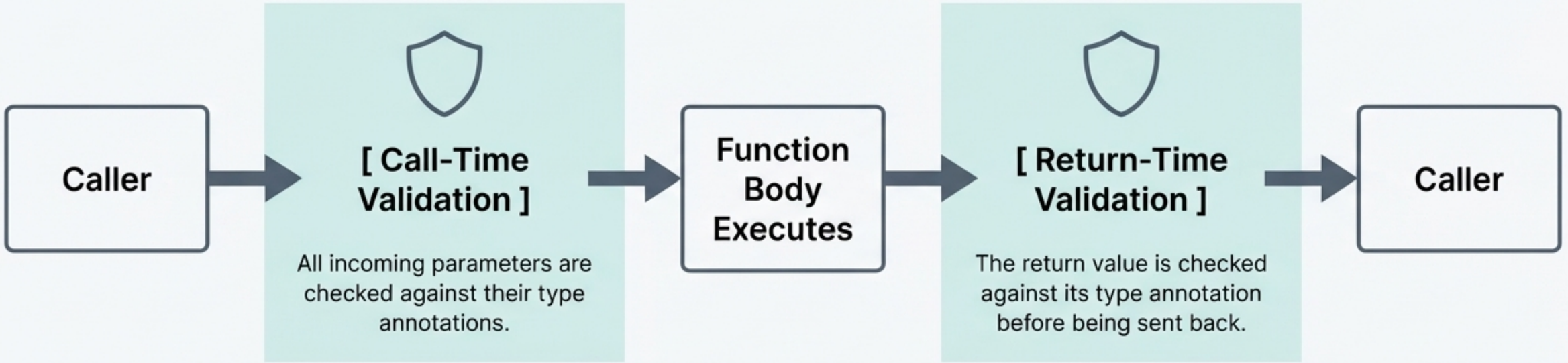
The decorator that provides robust runtime type enforcement for both **parameters** and **return values**. It does what you always thought Python should do.

The Difference is Immediate and Comprehensive

Capability	Python Default	@type_safe
Parameter type checking	❌	✅
Return type checking	❌	✅
List/Dict element validation	❌	✅
Optional via `= None` default	❌	✅
Clear error messages	❌	✅
Collection auto-conversion	❌	✅

The Foundation: Two-Phase Validation

The `@type_safe` decorator validates at two critical points in a function's lifecycle, ensuring that bad data can neither enter nor exit.



Beyond the Surface: Deep Validation for Collections

Standard type hints only check that you have a list. @type_safe checks that **every element** inside that list is the correct type.

Standard Python - The Problem

```
def process_ids(user_ids: list[int]):
    # This will run without error!
    # ... eventually crashing later
    pass

process_ids([101, 102, "103", 104]) # Silent failure
```

@type_safe - The Solution

```
@type_safe
def process_ids(user_ids: list[int]):
    pass

# Raises TypeError instantly!
process_ids([101, 102, "103", 104])
```

A More Pythonic Approach to Optionals

Don’t use Optional[T]. The decorator’s design philosophy promotes a cleaner, more explicit signal for optional parameters.

The Recommended Pattern

```
@type_safe
def find_user(user_id: int, include_profile: bool = None):
    # The default '= None' makes it optional.
    # No 'Optional[bool]' needed.
    ...
```

- `= None` clearly signals optionality to the reader.
- It’s the **caller’s responsibility** to handle a `None` response.
- Less verbose, same behaviour, more Pythonic.

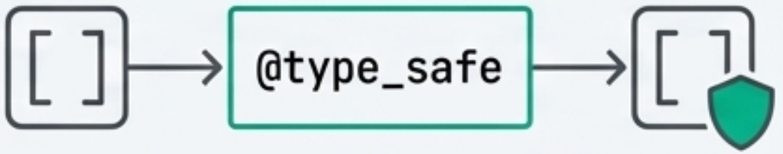
Safety that Flows: Automatic Return Conversion

The decorator doesn't just check your return values; it automatically converts standard collections into their `Type_Safe` equivalents, ensuring safety is maintained throughout the entire call chain without extra code.

```python
@type_safe
def get_active_users() -> Type_Safe__List[User]:
    # This function returns a plain Python list...
    users = [User(), User()]
    return users

# ...but the caller receives a fully validated Type_Safe__List!
safe_user_list = get_active_users()
# safe_user_list.append("not_a_user") would now fail.
```

Supported Conversions



- `list` → `Type_Safe__List`
- `set` → `Type_Safe__Set`
- `dict` → `Type_Safe__Dict`

Stop Guessing: Error Messages That Actually Help

Vague, generic `TypeError`s are a thing of the past. `@type_safe` tells you exactly what went wrong, where it happened, and what it expected.

The Vague `TypeError`

```
TypeError: can only concatenate
str (not "int") to str
```

Where did the `int` come from? Which argument was it?

The Actionable Error

```
TypeError: Parameter 'user_name'
expected <class 'str'> but got
<class 'int'> with value 123
```

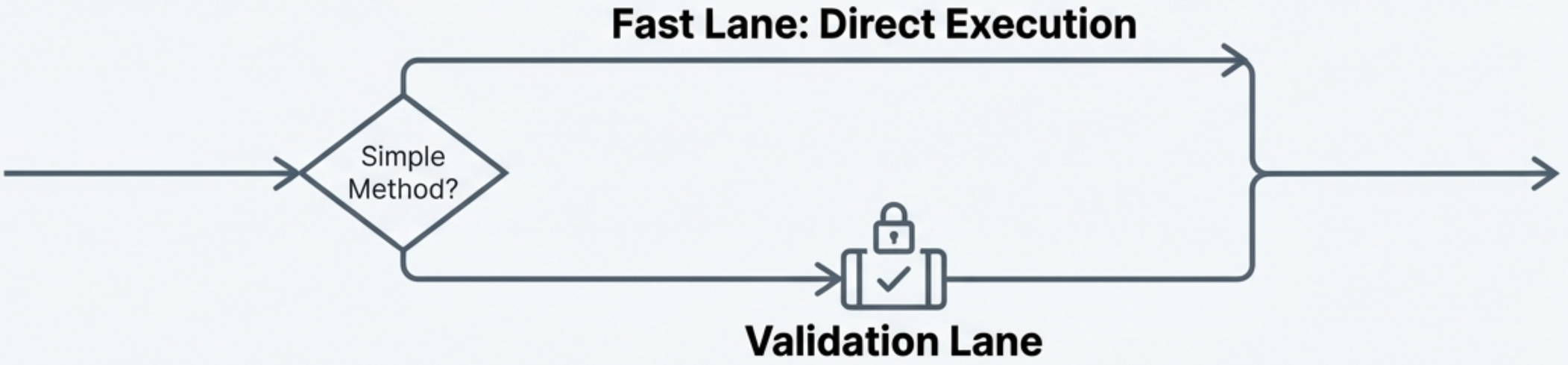
The exact parameter, the expected type, and the received type/value. Debugging solved in seconds.

Designed for Performance Where It Matters

Yes, runtime validation has a cost. The decorator is engineered to minimise it, especially in common scenarios.

The Optimisation: Direct Execution

The decorator intelligently detects simple methods (e.g., those with no parameters or only `self`). For these, it pre-calculates that no validation is needed and bypasses most of the overhead, allowing for near-native execution speed.



Guidance: Apply `@type_safe` to your public API methods and boundaries. For performance-critical, internal helper functions, you can choose to omit it.

How to Use `@type_safe` Like an Expert

Decorate Your Boundaries

Apply `@type_safe` to public API methods—where untrusted data enters your system.

Always Annotate Returns

A function without a return annotation cannot have its output validated.

Use `Safe_*` Types

For parameters needing domain validation (e.g., `Safe_String`), let the types do the work.

Let Auto-Conversion Work For You

Annotate returns with `Type_Safe` collections and let the decorator handle the conversion.

Use `Type[T]` for Factories

When a parameter expects a class itself, not an instance, use `Type[T]`.

Your `@type_safe` Checklist

- [ ] Import the decorator.
- [ ] Add `@type_safe` above your method definition.
- [ ] Annotate **ALL** parameters with their types.
- [ ] Annotate the return type using `-> Type`.
- [ ] Use `param: T = None` for optional parameters (NOT `Optional[T]`).
- [ ] Trust `None` is always a valid return; the caller must handle it.
- [ ] Use `List[T]` and `Dict[K, V]` to validate all elements.
- [ ] Trust auto-conversion for `Type_Safe` collection returns.
- [ ] Expect clear, actionable error messages.

Write safer, more predictable Python. Today.