mypy and Static Type Checking Concepts in Python

What Is Static Type Checking?

Static type checking means analyzing Python code **before running it** to find places where values are being used with incompatible types.

Python is dynamically typed, so this is valid:

```
value = 10
value = "hello"
```

Python does not require you to declare the type of `value`.

Type hints let us describe our intended types:

```
value: int = 10
```

A static type checker can then inspect the code and detect when our usage does not match those expectations.

One of the most popular tools for this is **mypy**.

What Is mypy?

`mypy` is a static type checker for Python.

It reads your Python source code and its type annotations and reports potential type-related problems.

For example:

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

This is fine:

```
result = add(10, 20)
```

But:

```
result = add("10", 20)
```

can be reported by mypy because `"10"` is a `str`, while the function expects an `int`.

The important point is:

```
Python
  ↓
```

```
runs your program

mypy
  ↓
analyzes your code
```

They perform different jobs.

Static vs Dynamic

Understanding this distinction is essential.

Dynamic Typing

Python determines and uses types while the program runs.

```
value = 10
print(value + 5)
```

At runtime:

```
value → int
```

Python knows the actual type of the object.

Static Type Checking

A tool analyzes the code without executing the program.

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

add("10", 20)
```

A static checker can see that the call violates the declared function contract.

Type Hints Are the Information

Consider:

```
def calculate_total(
    price: float,
    quantity: int
) -> float:
    return price * quantity
```

The annotations tell the checker:

```
price    → float
quantity → int
return   → float
```

mypy uses this information to reason about your code.

Without type hints, static analysis has less information available.

---

mypy Does Not Run Your Program

Suppose you have:

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

result = add("10", 20)
```

Running:

```
python app.py
```

and running:

```
mypy app.py
```

are different operations.

Python executes the application.

mypy analyzes the source code.

A useful mental model is:

```
Source Code
    ↓
 ┌──────────────┐
 │              │
Python         mypy
 │              │
Execution      Analysis
 │              │
Runtime        Type Errors
```

Installing mypy

Install it with:

```
pip install mypy
```

Verify the installation:

```
mypy --version
```

You should see the installed mypy version.

Your First mypy Check

Create:

```
# app.py

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


result = add(10, 20)

print(result)
```

Run:

```
mypy app.py
```

If there are no detected type errors, mypy reports that the file is clean.

Introduce a Type Error

Change:

```
result = add(10, 20)
```

to:

```
result = add("10", 20)
```

Now run:

```
mypy app.py
```

mypy can identify that the first argument has the wrong type.

The exact wording can vary by mypy version, but conceptually the error means:

```
Expected: int
Received: str
```

Why Static Checking Is Useful

Imagine a small project with:

```
app.py
users.py
resources.py
services.py
database.py
```

A function may be used in many places.

Changing:

```
def get_user(user_id: int) -> User:
```

can affect several files.

A static type checker can help identify places where the new contract is being violated.

This becomes increasingly valuable as a project grows.

Type Checking Is About Contracts

Consider:

```
def find_resource(resource_id: str) -> Resource | None:
    ...
```

This creates an informal contract:

```
Input:
resource_id → str

Output:
Resource OR None
```

Another part of the application should respect that contract.

Static checking helps identify code that does not.

Function Argument Checking

Example:

```
def greet(name: str) -> str:
    return f"Hello {name}"
```

Correct:

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

Potential type error:

```
greet(25)
```

mypy can detect the incompatible argument.

Return Type Checking

Consider:

```
def get_age() -> int:
    return "25"
```

The function says it returns an integer:

```
-> int
```

but actually returns a string:

```
"25"
```

mypy can report this inconsistency.

The important concept is:

```
Declared contract
        ↓
       int
        ↓
Actual return
        ↓
       str
        ↓
Potential type error
```

Variable Type Checking

You can explicitly annotate variables:

```
age: int = 25
```

This is compatible.

But:

```
age: int = "25"
```

conflicts with the annotation.

A type checker can identify this.

Type Inference

mypy can often infer types without explicit annotations.

For example:

```
name = "Hafsa"
```

mypy can infer:

```
name → str
```

And:

```
numbers = [1, 2, 3]
```

can be inferred as:

```
list[int]
```

Therefore, you do not need to annotate every local variable.

Why Type Inference Matters

Compare:

```
name: str = "Hafsa"
age: int = 25
numbers: list[int] = [1, 2, 3]
```

with:

```
name = "Hafsa"
age = 25
numbers = [1, 2, 3]
```

The second version is often perfectly clear.

Good type checking does not mean adding annotations everywhere.

It means providing enough information for useful static analysis.

Type Inference Can Be Wrong About Your Intent

Consider:

```
items = []
```

An empty list does not provide much information.

mypy may not know what you intend to put inside it.

You can make the intended type explicit:

```
items: list[str] = []
```

Now the contract is clear.

Type Narrowing

One of the most useful concepts in static type checking is **type narrowing**.

Suppose:

```
def print_name(name: str | None) -> None:
    ...
```

The value can be either:

```
str
```

or:

```
None
```

Inside the function:

```
def print_name(name: str | None) -> None:
    if name is not None:
        print(name.upper())
```

After:

```
if name is not None:
```

mypy can understand that `name` is now a `str`.

This is called type narrowing.

Narrowing With `isinstance`

Consider:

```
def process(value: int | str) -> None:
    if isinstance(value, int):
        print(value + 10)
    else:
        print(value.upper())
```

Inside the first branch:

```
value → int
```

Inside the second:

```
value → str
```

The checker follows the logic of the program.

Narrowing With `is None`

Example:

```
def get_length(value: str | None) -> int:
    if value is None:
        return 0

    return len(value)
```

After the `return`, mypy knows:

```
value → str
```

because the `None` case has already been handled.

`Any`

`Any` is one of the most important concepts in static type checking.

```
from typing import Any

value: Any = "hello"
```

An `Any` value can generally be used as almost any type.

For example:

```
value: Any = "hello"

value.upper()
```

```
value + 10
value.some_unknown_method()
```

A type checker generally cannot provide useful protection for operations involving `Any`.

Why Too Much `Any` Is Dangerous

Consider:

```
def process(data: Any) -> Any:
    ...
```

You have effectively removed much of the static type information.

This can make a project look typed while providing little actual protection.

Prefer specific types whenever possible.

The Gradual Typing Model

Python's type system is gradual.

That means a project can contain:

```
strongly typed code
+
partially typed code
+
dynamically typed code
```

You do not have to convert an entire old project to strict typing in one step.

You can gradually introduce type checking.

`object` vs `Any`

These are not the same.

Consider:

```
value: object
```

`object` means the value could be any Python object.

But unlike `Any`, the checker does not automatically allow every operation.

For example:

```
value: object

```

```
value.upper()
```

mypy can reject this because not every Python object has `.upper()`.

You need to narrow the type first.

This makes `object` safer than `Any` when you genuinely do not know the concrete type.

Strictness Levels

mypy can operate with different levels of strictness.

A project can start with relatively permissive checking and become stricter over time.

Basic:

```
mypy .
```

More strict:

```
mypy --strict .
```

Strict mode enables additional checks.

It can reveal issues that a basic configuration does not report.

Why Not Start With Maximum Strictness?

For a brand-new project, strict checking can be useful.

For a large existing project, immediately enabling every strict rule may produce a large number of errors.

A practical migration strategy is:

```
Existing project
      ↓
Basic checking
      ↓
Fix important errors
      ↓
Add annotations
      ↓
Increase strictness
      ↓
Maintain type safety
```

`mypy.ini`

mypy can be configured through a configuration file.

Example:

```
[mypy]
python_version = 3.12
check_untyped_defs = True
disallow_untyped_defs = True
warn_return_any = True
```

The exact configuration should depend on the project's Python version and requirements.

`pyproject.toml` Configuration

Modern Python projects can also configure mypy through `pyproject.toml`.

Example:

```
[tool.mypy]
python_version = "3.12"
check_untyped_defs = true
disallow_untyped_defs = true
warn_return_any = true
```

This keeps project tooling configuration in one central file.

Checking Multiple Files

Instead of:

```
mypy app.py
```

you can check a package or project:

```
mypy .
```

mypy analyzes the relevant Python files according to its configuration.

For larger projects, this is much more useful than checking one file manually.

Checking a Package

Suppose the project contains:

```
app/
    __init__.py
    models.py
```

```
    services.py
    repositories.py
```

You can check the package:

```
mypy app
```

This allows mypy to analyze relationships across modules.

Import Analysis

Static type checking becomes more valuable when code is divided into modules.

For example:

```
# users.py

class User:
    ...
```

Then:

```
# services.py

from users import User

def get_user() -> User:
    ...
```

mypy can use imported type information to understand the relationship between the modules.

Third Party Libraries

Not every Python package originally contains complete type information.

mypy may need additional information to understand some dependencies.

Some packages provide their own type information.

Others may provide separate stub packages.

Type Stubs

A type stub describes the types available in a module without containing the full implementation.

Stub files commonly use:

```
.pyi
```

For example:

```
library/
    __init__.py
    api.pyi
```

A stub might describe:

```
def calculate(value: int) -> str: ...
```

The actual implementation may live elsewhere.

Type checkers can use these declarations.

`types-` Packages

Some third-party libraries have separate type stub packages.

For example, you may encounter packages named like:

```
pip install types-requests
```

These provide typing information for libraries that need external stubs.

Whether a package needs stubs depends on the package and its typing support.

Missing Imports

You may encounter errors related to missing type information for imported libraries.

For example, mypy may know:

```
import some_library
```

but not have enough information to analyze its types.

The solution may involve:

- installing a typed version
- installing a stub package
- configuring the import
- adding your own stubs
- allowing the import selectively

Do not blindly ignore every missing-import warning.

`# type: ignore`

Sometimes a specific line genuinely cannot be understood by the type checker.

You can suppress a warning:

```
  result = legacy_function()  # type: ignore
```

But this should be used carefully.

A broad ignore can hide real problems.

Prefer a targeted suppression when necessary.

Why Excessive `type: ignore` Is a Problem

Imagine:

```
  value = get_data()  # type: ignore
```

and later:

```
  result = process(value)  # type: ignore
```

Eventually, large parts of the code become invisible to static analysis.

A better approach is to understand why the type checker is complaining and improve the type information where possible.

`cast`

Sometimes you know something about a value that the type checker cannot infer.

Python provides `cast`.

```
  from typing import cast

  value = get_value()

  name = cast(str, value)
```

This tells the type checker:

```
  Treat value as str here.
```

Important:

`cast()` does **not** perform runtime validation or conversion.

It only affects static type checking.

`cast` Is Not Conversion

This:

```
value = cast(int, "123")
```

does not turn the string into an integer.

The value is still:

```
"123"
```

If you need conversion:

```
value = int("123")
```

`cast` and conversion solve completely different problems.

`assert` and Type Narrowing

Assertions can help static type checkers narrow types.

Example:

```
def process(name: str | None) -> str:
    assert name is not None
    return name.upper()
```

After the assertion, the checker knows that `name` is not `None`.

However, remember that `assert` is also a runtime statement.

It can raise `AssertionError` if the condition is false.

`assert_type`

For more advanced typing tests, Python provides tools such as:

```
from typing import assert_type
```

Example:

```
name = "Hafsa"

assert_type(name, str)
```

This can be useful when testing whether a type checker infers the type you expect.

`reveal_type`

mypy provides a very useful debugging feature:

```
reveal_type(value)
```

For example:

```
name = "Hafsa"

reveal_type(name)
```

mypy can report something like:

```
Revealed type is "builtins.str"
```

This helps you understand what mypy believes the type is.

Why `reveal_type` Is Useful

Consider:

```
def find_user(user_id: int) -> User | None:
    ...

user = find_user(10)

reveal_type(user)
```

mypy can tell you:

```
User | None
```

Then you can narrow it:

```
if user is not None:
    reveal_type(user)
```

Now mypy can identify it as:

```
User
```

Type Narrowing and Control Flow

Static type checkers analyze control flow.

Example:

```
def process(value: int | str | None) -> None:

    if value is None:
        return
```

```
    if isinstance(value, int):
        print(value + 10)
        return

    print(value.upper())
```

The checker understands the different branches:

```
None
 ↓
return

int
 ↓
integer operation

str
 ↓
string operation
```

This is one reason good type annotations work best when code has clear control flow.

Invariance

Generic collections have important type relationships.

Consider:

```
list[int]
```

and:

```
list[object]
```

It may seem like:

```
list[int] → list[object]
```

should always be allowed.

But mutable lists are generally invariant.

Why?

Because if a `list[int]` were treated as a `list[object]`, code could potentially insert a string into it.

```
numbers = [1, 2, 3]
```

Then:

```
objects = numbers
objects.append("hello")
```

Now the original integer list contains a string.

Static type systems therefore use rules that protect mutable collections.

Covariance

Some read-only or producer-like types can safely be covariant.

For example, an object that only produces values can often be treated more flexibly than a mutable collection that accepts values.

This becomes important when designing generic APIs.

You do not need to memorize every variance rule immediately.

The practical lesson is:

> Mutable containers have stricter type relationships because they can be changed.

`Sequence` Instead of `list`

Suppose a function only reads items:

```
from collections.abc import Sequence

def total(numbers: Sequence[int]) -> int:
    return sum(numbers)
```

The function does not need to modify the collection.

Using `Sequence` communicates that requirement more accurately than:

```
def total(numbers: list[int]) -> int:
```

This can make APIs more flexible.

`Iterable` vs `Sequence`

An `Iterable` only needs to support iteration:

```
from collections.abc import Iterable

def print_items(items: Iterable[str]) -> None:
```

```
    for item in items:
        print(item)
```

A `Sequence` provides more:

```
iteration
+
indexing
+
ordered access
```

For example:

```
items[0]
```

works for a sequence.

Choose the narrowest interface your function actually needs.

Protocol Based Checking

Static type checking can work with behavior instead of inheritance.

Example:

```
from typing import Protocol


class HasName(Protocol):
    name: str
```

A class does not need to inherit from `HasName`.

If it provides:

```
class User:
    def __init__(self, name: str):
        self.name = name
```

then a function expecting `HasName` can use it.

This is called structural typing.

Structural vs Nominal Typing

Nominal typing

The relationship depends on declared inheritance or named types.

```
class Dog(Animal):
    ...
```

Structural typing

The relationship depends on whether the object has the required structure or behavior.

```
class Printable(Protocol):
    def print_data(self) -> str:
        ...
```

Protocols provide a way to express structural expectations.

Generic Functions

Generics allow a function to preserve type relationships.

```
from typing import TypeVar

T = TypeVar("T")


def first(items: list[T]) -> T:
    return items[0]
```

Then:

```
first([1, 2, 3])
```

has:

```
int
```

while:

```
first(["Hafsa", "Asim"])
```

has:

```
str
```

The same function works with multiple types while maintaining useful type information.

Type Checking and OOP

mypy can analyze classes, inheritance, method overriding, and attributes.

Example:

```
class Animal:
    def speak(self) -> str:
        return "Sound"


class Dog(Animal):
    def speak(self) -> str:
        return "Woof"
```

The method signatures are compatible.

But an incompatible override can be reported.

For example, changing the child method to an incompatible signature can violate the parent's contract.

Type Checking Method Overrides

Parent:

```
class Animal:
    def speak(self, volume: int) -> str:
        return "Sound"
```

Child:

```
class Dog(Animal):
    def speak(self, volume: str) -> str:
        return "Woof"
```

The child changes:

```
int
```

into:

```
str
```

This can violate substitutability and static type checking rules.

Type checkers can help catch these design problems.

`@override`

Modern Python versions provide `typing.override`.

Example:

```
from typing import override


class Animal:
    def speak(self) -> str:
        return "Sound"


class Dog(Animal):

    @override
    def speak(self) -> str:
        return "Woof"
```

This explicitly communicates that the method is intended to override a parent method.

It can help type checkers catch accidental or incompatible overrides.

Type Checking Untyped Functions

Consider:

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

There are no annotations.

mypy has much less information about the function.

Compare:

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

The second function provides a clear contract.

For a gradually typed project, adding annotations to important functions is often a good starting point.

`disallow_untyped_defs`

A project can configure mypy to require annotations for function definitions.

For example:

```
[mypy]
disallow_untyped_defs = True
```

This helps prevent new untyped functions from entering the codebase.

`check_untyped_defs`

You can also ask mypy to inspect the bodies of untyped functions:

```
[mypy]
check_untyped_defs = True
```

This provides some checking even when a function has not been fully annotated.

Strict Optional Checking

A major source of bugs is accidentally treating `None` as another value.

Consider:

```
def find_user(user_id: int) -> User | None:
    ...
```

Then:

```
user = find_user(1)

print(user.name)
```

This is unsafe because `user` might be `None`.

The correct pattern is:

```
if user is not None:
    print(user.name)
```

Static checking helps make these cases visible.

The `Optional` Mental Model

Do not think:

```
Optional[User]
```

means:

```
probably User
```

Think:

```
User OR None
```

Therefore, your code must handle both possibilities.

Modern syntax:

```
User | None
```

is often easier to read.

Type Checking External Data

Static type checking is strongest when your program already knows the types.

External data is different.

Examples:

- JSON
- API responses
- user input
- environment variables
- files
- database results

These values need parsing or validation before you can safely treat them as trusted typed data.

Example: User Input

```
age = input("Enter age: ")
```

The result is a string.

If you need an integer:

```
age = int(input("Enter age: "))
```

Type checking can understand the second result as an integer.

But invalid input such as:

```
twenty
```

can still cause a runtime `ValueError`.

Static type checking does not replace runtime error handling.

Example: Environment Variables

Environment variables are typically strings.

```
import os

port = os.getenv("PORT")
```

The result may be:

```
str | None
```

You should handle the possibility of `None` and convert the value when an integer is required.

```
port_value = os.getenv("PORT")

if port_value is None:
    raise RuntimeError("PORT is missing")

port = int(port_value)
```

Now the program has performed runtime validation/conversion.

Type Checking and Tests

Static checking and automated tests complement each other.

Type checker

Can catch:

```
wrong argument type
wrong return type
incompatible assignment
missing attributes
incorrect overrides
some unreachable or inconsistent code
```

Tests

Can catch:

```
wrong business logic
incorrect output
unexpected behavior
edge cases
integration failures
```

A robust project can use both.

---

mypy in a CI Pipeline

A professional project can run mypy automatically.

For example:

```
Developer pushes code
        ↓
CI starts
        ↓
Run tests
        ↓
Run mypy
        ↓
Check formatting/linting
        ↓
Build/deploy
```

If mypy finds a blocking error, the CI pipeline can fail.

This prevents new type problems from being merged.

Type Checking as a Quality Gate

A project can define a rule:

```
No new type errors
```

This allows a large legacy codebase to improve gradually.

Instead of requiring perfect typing immediately:

```
Old code → many issues
        ↓
Fix important areas
        ↓
Prevent new issues
        ↓
Improve coverage
        ↓
Stronger typing over time
```

Common mypy Workflow

A practical workflow is:

1. Write code

```
def calculate_total(
    price: float,
    quantity: int
) -> float:
    return price * quantity
```

2. Run mypy

```
mypy .
```

3. Read the error

Do not immediately suppress it.

4. Find the root cause

Maybe:

```
wrong argument
wrong return value
incorrect annotation
missing None handling
incorrect imported type
```

5. Fix the code or annotation

6. Run mypy again

7. Run tests

Type checking is not a replacement for tests.

Common mypy Errors and What They Mean

Argument type mismatch

Conceptually:

```
Argument 1 has incompatible type "str"; expected "int"
```

Meaning:

You passed the wrong type to a function.

Incompatible return value

Conceptually:

```
Incompatible return value type
```

Meaning:

The returned value does not match the declared return type.

Incompatible assignment

Conceptually:

```
Incompatible types in assignment
```

Meaning:

A value does not match the variable's expected type.

Item has no attribute

Example:

```
name: str = "Hafsa"

name.append("x")
```

A string does not have `append()`.

Possibly None

Example:

```
user: User | None
print(user.name)
```

The value may be `None`.

You need to narrow it first.

Common Mistakes

1. Thinking mypy changes Python's runtime behavior

It does not.

mypy analyzes code.

---

2. Assuming type hints validate external data

They do not.

Use runtime validation and parsing.

---

3. Using `Any` to silence everything

This removes useful static information.

---

4. Adding `type: ignore` everywhere

This hides problems instead of solving them.

---

5. Using `cast()` as validation

`cast()` does not check or convert a value at runtime.

---

6. Ignoring `None`

If the type says:

```
User | None
```

handle both possibilities.

---

7. Making every annotation extremely specific

Overly complicated types can make code harder to understand.

Prefer useful and maintainable types.

---

8. Running mypy only before deployment

Type checking is most useful when it becomes part of normal development and CI.

Best Practices

1. Start With Function Boundaries

Annotate:

```
def search(query: str) -> list[Resource]:
```

before worrying about every local variable.

---

2. Avoid `Any` When a Real Type Exists

Prefer:

```
list[Resource]
```

over:

```
Any
```

when the structure is known.

---

3. Use Type Narrowing

Write clear checks:

```
if user is None:
    return

print(user.name)
```

---

4. Keep Functions Small

Smaller functions make both human reasoning and static analysis easier.

---

5. Use Abstract Interfaces Where Appropriate

Use:

```
Iterable[str]
```

when you only need iteration.

Use:

```
Sequence[str]
```

when you need ordered/indexed access.

---

6. Run Type Checking Automatically

Use CI or pre-commit tooling so type checking does not depend on remembering a command manually.

---

7. Increase Strictness Gradually

Do not create an unmanageable wall of errors in a large existing project.

---

8. Fix Root Causes

When mypy reports an error, first ask:

> Is my code wrong, or is my annotation incomplete?

Do not automatically suppress the warning.

---

mypy vs Runtime Validation

Concern	mypy	Runtime Validation
Runs before program execution	Yes	No
Executes application logic	No	Yes
Checks annotations	Yes	Depends on tool
Validates user input	No	Yes
Validates API data	No	Yes
Detects wrong function arguments	Often	Only if runtime code checks
Converts values	No	Can
Helps IDEs/tools	Yes	Sometimes

Finds some bugs before execution	Yes	No

The two approaches solve different problems.

---

mypy vs pytest

`pytest` and `mypy` should not be treated as competing tools.

They answer different questions.

```
pytest
   ↓
Does the program behave correctly?

mypy
   ↓
Are values being used consistently with their declared types?
```

For a professional Python project, both can be valuable.

---

mypy vs Linters

A linter checks code quality and potential problems such as:

- unused variables
- style issues
- suspicious patterns
- imports
- complexity

A type checker focuses on type relationships.

Modern Python projects often use several tools together.

For example:

```
Formatter
   +
Linter
   +
Type Checker
```

```
   +
Tests
```

Each provides a different layer of feedback.

Mini Project: Typed Resource Service

Create a small service for haas.dev resources.

Step 1: Model

```
from dataclasses import dataclass


@dataclass
class Resource:
    title: str
    category: str
    pages: int
```

Step 2: Storage

```
resources: list[Resource] = []
```

Step 3: Add Function

```
def add_resource(resource: Resource) -> None:
    resources.append(resource)
```

Step 4: Search Function

```
def find_resource(
    title: str
) -> Resource | None:

    for resource in resources:
        if resource.title == title:
            return resource

    return None
```

Step 5: Introduce an Error

Try:

```
add_resource("Python")
```

mypy should identify that a string is being passed where a `Resource` is expected.

Step 6: Fix It

```
resource = Resource(
    title="Python Type Hints",
    category="Python",
    pages=20
)

add_resource(resource)
```

Step 7: Test `None`

```
resource = find_resource("Does Not Exist")

if resource is not None:
    print(resource.title)
```

This project demonstrates:

- annotations
- inference
- function contracts
- `None`
- type narrowing
- static checking
- classes
- collections

---

5 Minute Challenge

Create:

```
def calculate_average(scores: list[int]) -> float:
    ...
```

Then intentionally create three errors:

1. Pass a string instead of a list.
2. Return a string instead of a float.
3. Put strings inside the integer list.

Run:

```
mypy .
```

Study each error.

Then fix all three without using:

```
# type: ignore
```

Exercises

Exercise 1: Function Contract

Create:

```
def format_price(...)
```

Requirements:

```
Input:
price → float

Output:
str
```

Run mypy against the file.

Exercise 2: Optional Value

Create:

```
def find_username(
    user_id: int
) -> str | None:
    ...
```

Use proper type narrowing before calling string methods.

Exercise 3: Nested Data

Create:

```
users: list[dict[str, int | str]]
```

Write a function that safely retrieves a user's name.

Use type narrowing where necessary.

Exercise 4: Generic Function

Create:

```
T = TypeVar("T")

def first(items: list[T]) -> T:
    ...
```

Verify that:

```
first([1, 2, 3])
```

is inferred as `int`.

And:

```
first(["a", "b"])
```

is inferred as `str`.

Use `reveal_type()` to inspect the result.

Exercise 5: Protocol

Create a `Protocol` called:

```
Printable
```

It should require:

```
print_data() -> str
```

Create two unrelated classes that satisfy the protocol.

Pass both to the same function.

Interview Questions

1. What is static type checking?

Static type checking analyzes source code before execution to identify potential type inconsistencies.

2. What is mypy?

mypy is a static type checker for Python that uses type annotations and inference to analyze code.

3. Does mypy execute Python code?

No. Its primary role is static analysis.

4. Does Python enforce type hints at runtime?

Normally, no.

5. What is type inference?

It is the ability of a type checker to determine a value's likely type from its usage and assigned values.

6. What is type narrowing?

Type narrowing is reducing a broader type to a more specific type based on control-flow checks.
Example:

```
  if value is not None:
```

can narrow:

```
  str | None
```

to:

```
  str
```

7. Why is `Any` dangerous?

It disables much of the static type checking for values that use it.

8. What is `cast()`?

`cast()` tells the type checker to treat a value as a specified type. It does not perform runtime conversion or validation.

9. What is `reveal_type()`?

It helps developers inspect the type that mypy infers for an expression.

10. What is a type stub?

A `.pyi` file containing type information that a type checker can use for a module or library.

11. What is gradual typing?

Gradual typing allows typed and dynamically typed portions of a Python project to coexist.

12. Why use `Sequence` instead of `list`?

When a function only requires sequence behavior, `Sequence` can make the API more general and accurately describe its requirements.

13. What is a `Protocol`?

A Protocol describes the structure or behavior an object must provide for static type checking, without requiring explicit inheritance.

14. Does mypy replace tests?

No. Type checking and testing detect different classes of problems.

15. Why use mypy in CI?

It can prevent new type errors from being merged or deployed.

Static Type Checking Cheat Sheet

Install

```
pip install mypy
```

Check one file

```
mypy app.py
```

Check a project

```
mypy .
```

Strict checking

```
mypy --strict .
```

Inspect an inferred type

```
reveal_type(value)
```

Explicit variable type

```
age: int = 25
```

Function

```
def greet(name: str) -> str:
    return f"Hello {name}"
```

Optional

```
User | None
```

Union

```
int | str
```

Generic

```
from typing import TypeVar

T = TypeVar("T")
```

Runtime conversion

```
age = int("25")
```

Static cast

```
from typing import cast

age = cast(int, value)
```

Remember:

```
cast() ≠ conversion
```

Configuration

```
[tool.mypy]
python_version = "3.12"
```

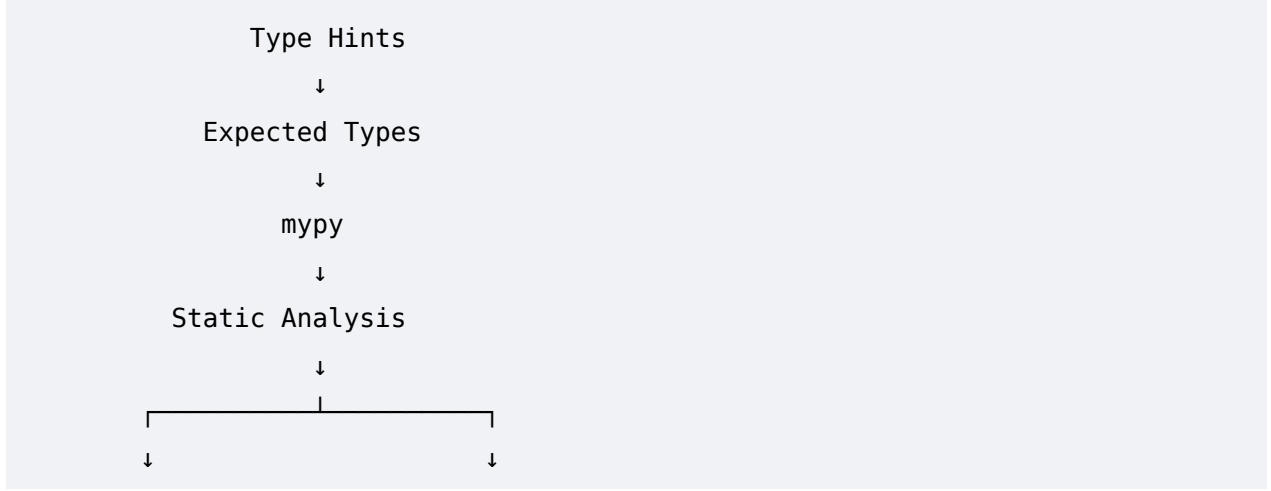
Ignore a specific line

```
value = legacy_function()  # type: ignore
```

Use sparingly.

Final Mental Model

Think of a Python application as a large system of contracts.



```
Compatible Code       Type Problems
                                ↓
                           Fix Before
                            Execution
```

But remember that static checking is only one layer:

```
Type Hints
    ↓
mypy
    ↓
Tests
    ↓
Runtime Validation
    ↓
Production Monitoring
```

Each layer catches different kinds of problems.

The goal is not to make every Python value perfectly annotated.

The goal is to make important contracts clear enough that both developers and tools can detect mistakes earlier.

Visit haas.dev for more resources and guides.

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

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