Type Hints in Python

What Are Type Hints?

Type hints are a way to tell Python, developers, and development tools what type of data a variable, parameter, or function is expected to contain.

For example:

```
name: str = "Hafsa"
age: int = 25
price: float = 499.99
is_active: bool = True
```

The type hint comes after the variable name:

```
variable_name: type
```

Type hints make code easier to understand because the expected data type is visible directly in the code.

Without a type hint:

```
name = "Hafsa"
```

With a type hint:

```
name: str = "Hafsa"
```

The second version immediately tells another developer that `name` is expected to be a string.

Why Do Type Hints Matter?

Python is dynamically typed.

That means Python does not require you to declare the type of a variable before using it.

This is valid:

```
value = 10
value = "hello"
value = [1, 2, 3]
```

Python allows the variable to refer to different types of values during its lifetime.

This flexibility is useful, but in larger applications it can make code harder to understand.

Consider:

```
def calculate_total(price, quantity):
    return price * quantity
```

What should `price` be?

What should `quantity` be?

What does the function return?

The function itself does not tell us.

With type hints:

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

Now the expected types are clear:

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

Type hints improve:

- readability
- editor support
- autocomplete
- static analysis
- documentation
- refactoring
- team collaboration
- maintenance
- bug detection

Type Hints Do Not Turn Python Into a Statically Typed Language

This is one of the most important things to understand.

Type hints usually do **not** enforce types at runtime.

For example:

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

Python will not automatically reject this:

```
greet(123)
```

The type hint says:

```
name should be a str
```

But Python itself generally does not enforce that annotation.

Tools such as:

- mypy
- Pyright
- IDEs
- linters

can inspect your code and report type-related problems.

Think of type hints as **information for humans and tools**, rather than automatic runtime restrictions.

Basic Variable Type Hints

The simplest syntax is:

```
variable: type = value
```

Examples:

```
name: str = "Hafsa"
age: int = 25
height: float = 5.2
is_developer: bool = True
```

You can also annotate a variable without assigning a value immediately:

```
username: str
```

Later:

```
username = "Hafsa"
```

This is useful when a variable is initialized later.

Common Built In Types

Some of the most common annotations are:

```
name: str
age: int
price: float
active: bool
data: None
```

Example:

```
product_name: str = "Python Course"
student_count: int = 100
rating: float = 4.8
is_published: bool = True
```

Type Hints for Functions

Function parameters can have type hints.

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

Here:

```
name: str
```

means the function expects `name` to be a string.

You can also specify the return type.

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

The syntax is:

```
def function(parameter: Type) -> ReturnType:
    ...
```

Multiple Parameters

You can annotate every parameter.

```
def calculate_area(length: float, width: float) -> float:
    return length * width
```

This tells us:

```
length → float
width  → float
result → float
```

Another example:

```
def create_user(name: str, age: int, active: bool) -> dict:
    return {
        "name": name,
        "age": age,
        "active": active
    }
```

Type Hints With Default Arguments

Type hints work with default values.

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

Another example:

```
def calculate_discount(price: float, discount: float = 0.10) -> float:
    return price * (1 - discount)
```

The type hint describes the expected type.

The default value provides what should happen when an argument is omitted.

Return Type Hints

Return annotations are written after `->`.

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

The return type is:

```
-> int
```

For a function returning a string:

```
def get_name() -> str:
    return "Hafsa"
```

For a function returning a boolean:

```
def is_valid() -> bool:
    return True
```

For a function that intentionally returns nothing:

```
def save_user(name: str) -> None:
    print(f"Saving {name}")
```

`None` means the function does not return a meaningful value.

Type Hints for Lists

A list can contain a specific type of value.

Modern Python allows:

```
names: list[str] = ["Hafsa", "Asim"]
```

This means:

```
names is expected to be a list of strings
```

For integers:

```
scores: list[int] = [90, 85, 95]
```

For floats:

```
prices: list[float] = [10.5, 20.0, 15.75]
```

Function example:

```
def calculate_average(scores: list[int]) -> float:
    return sum(scores) / len(scores)
```

Type Hints for Dictionaries

You can describe both dictionary key and value types.

```
user: dict[str, str] = {
    "name": "Hafsa",
    "city": "Gujranwala"
}
```

This means:

```
key   → str
value → str
```

Another example:

```
scores: dict[str, int] = {
    "Hafsa": 95,
    "Asim": 88
}
```

Here:

```
key   → str
value → int
```

Function example:

```
def get_scores() -> dict[str, int]:
    return {
        "Hafsa": 95,
        "Asim": 88
    }
```

Type Hints for Tuples

Tuples can also be annotated.

```
user: tuple[str, int] = ("Hafsa", 25)
```

This means:

```
first value  → str
second value → int
```

Another example:

```
point: tuple[float, float] = (10.5, 20.2)
```

Both values are expected to be floats.

Type Hints for Sets

Use:

```
tags: set[str] = {"python", "web", "backend"}
```

This means the set contains strings.

For integers:

```
numbers: set[int] = {1, 2, 3, 4}
```

Type Hints for Nested Data

Real applications often contain nested structures.

For example:

```
users: list[dict[str, str]] = [
    {
        "name": "Hafsa",
        "role": "developer"
    },
    {
        "name": "Asim",
        "role": "designer"
    }
]
```

The annotation says:

```
users
  ↓
list
  ↓
dictionary
  ↓
string keys
string values
```

Another example:

```
products: dict[str, list[str]] = {
    "Python": ["Functions", "OOP", "Testing"],
    "JavaScript": ["Arrays", "Objects"]
}
```

The `Any` Type

Sometimes you genuinely do not know what type a value will contain.

Python provides `Any`.

```
from typing import Any

value: Any = 10
value = "hello"
value = [1, 2, 3]
```

`Any` essentially tells type checkers:

> Do not enforce a specific type here.

For example:

```
def process_data(data: Any) -> Any:
    return data
```

However, avoid using `Any` everywhere.

This:

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

provides very little useful type information.

Prefer specific types when you know what the data should be.

`Optional` and `None`

Sometimes a value can either have a type or be `None`.

Modern Python can express this using `|`.

```
name: str | None = None
```

This means:

```
name can be a string
OR
name can be None
```

Function example:

```
def find_user(user_id: int) -> str | None:
    if user_id == 1:
        return "Hafsa"

    return None
```

Older Python code often uses:

```
from typing import Optional

def find_user(user_id: int) -> Optional[str]:
    ...
```

Modern Python generally prefers:

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

Union Types

Sometimes a value can have more than one possible type.

Modern syntax:

```
value: int | str
```

This means:

```
value can be int OR str
```

Example:

```
def format_id(user_id: int | str) -> str:
    return str(user_id)
```

Both are valid:

```
format_id(101)
format_id("USR-101")
```

Older syntax:

```
from typing import Union

def format_id(user_id: Union[int, str]) -> str:
    return str(user_id)
```

Modern Python generally prefers:

```
int | str
```

Type Hints With `None`

A common mistake is confusing:

```
None
```

with:

```
str | None
```

They mean different things.

```
def save() -> None:
    ...
```

means the function returns nothing useful.

But:

```
def get_name() -> str | None:
    ...
```

means the function returns either:

```
str
```

or:

```
None
```

Type Aliases

If a type becomes complicated, you can give it a meaningful name.

```
User = dict[str, str]
```

Then:

```
def get_user() -> User:
    return {
        "name": "Hafsa",
        "role": "developer"
    }
```

For more complex aliases:

```
User = dict[str, str]
Users = list[User]
```

Now:

```
def get_users() -> Users:
    return [
        {
            "name": "Hafsa",
            "role": "developer"
        }
    ]
```

This can make complicated annotations easier to read.

`TypeAlias`

For explicit type aliases, modern Python supports:

```
from typing import TypeAlias

User: TypeAlias = dict[str, str]
```

This makes it clear that `User` is intended to represent a type alias.

`Literal`

Sometimes you do not want to allow every string.

You want specific values.

For example:

```
from typing import Literal

def set_status(status: Literal["draft", "published"]) -> None:
    print(status)
```

Valid:

```
set_status("draft")
set_status("published")
```

Invalid according to the type checker:

```
set_status("deleted")
```

`Literal` is useful when a parameter should accept one of a fixed set of values.

`Final`

Sometimes you want to communicate that a value should not be reassigned.

```
from typing import Final

MAX_USERS: Final = 100
```

A type checker can warn if you later try:

```
MAX_USERS = 200
```

`Final` communicates an intended constant.

Remember that type hints are not runtime enforcement.

`Callable`

Sometimes a function receives another function.

For example:

```
def apply_operation(
    value: int,
    operation: Callable[[int], int]
) -> int:
    return operation(value)
```

Here:

```
Callable[[int], int]
```

means:

```
function receives an int
function returns an int
```

Example:

```
def double(number: int) -> int:
    return number * 2

result = apply_operation(5, double)

print(result)
```

Output:

```
10
```

`Callable` is useful for callbacks, higher-order functions, event handlers, and dependency injection.

Type Hints for Classes

You can use classes as types.

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

A function can accept a `User` object:

```
def greet_user(user: User) -> str:
    return f"Hello {user.name}"
```

Now the function clearly expects a `User` object.

Type Hints for Class Attributes

You can annotate attributes:

```
class User:
    name: str
    age: int

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
```

This tells tools what type those attributes are expected to contain.

Type Hints for Methods

Methods can have parameter and return annotations.

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

Type hints work naturally with OOP.

Type Hints and Inheritance

Suppose we have:

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

```
        return "Sound"


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

A function can accept the parent type:

```
def make_animal_speak(animal: Animal) -> str:
    return animal.speak()
```

A `Dog` can be passed because it is an `Animal`.

```
dog = Dog()

print(make_animal_speak(dog))
```

This works well with polymorphism.

Type Hints for Iterables

Sometimes you do not care whether the caller provides a list, tuple, set, or another iterable.

Instead of:

```
def print_names(names: list[str]) -> None:
    ...
```

you can use:

```
from collections.abc import Iterable

def print_names(names: Iterable[str]) -> None:
    for name in names:
        print(name)
```

Now all of these can work:

```
print_names(["Hafsa", "Asim"])
print_names(("Hafsa", "Asim"))
print_names({"Hafsa", "Asim"})
```

This is often more flexible than requiring a specific collection type.

`Sequence`

If your function needs ordered, indexable data, you can use:

```
from collections.abc import Sequence

def first_item(items: Sequence[str]) -> str:
    return items[0]
```

A list or tuple can satisfy this expectation.

`Mapping`

If your function only needs dictionary-like access, you can use:

```
from collections.abc import Mapping

def get_username(user: Mapping[str, str]) -> str:
    return user["name"]
```

This is more general than requiring exactly:

```
dict[str, str]
```

Type Hints for Generators

Generators can also have type hints.

```
from collections.abc import Iterator

def count_numbers(limit: int) -> Iterator[int]:
    for number in range(limit):
        yield number
```

The function produces integers one at a time.

Type Hints for Async Functions

Type hints also work with asynchronous code.

```
import asyncio

async def get_data() -> str:
    await asyncio.sleep(1)
    return "Data loaded"
```

The return annotation describes the value produced by the coroutine when awaited.

Type Hints for `async` Iterators

For asynchronous iteration:

```
from collections.abc import AsyncIterator

async def generate_numbers() -> AsyncIterator[int]:
    for number in range(5):
        yield number
```

This becomes useful in asynchronous applications, APIs, streaming systems, and data processing.

Forward References

Sometimes a type refers to a class before that class has been defined.

For example:

```
class Node:
    def __init__(self, next_node: "Node | None" = None):
        self.next_node = next_node
```

The quotes allow the type to be represented as a forward reference.

Modern Python code may also use:

```
from __future__ import annotations
```

Then:

```
from __future__ import annotations

class Node:
    def __init__(self, next_node: Node | None = None):
        self.next_node = next_node
```

This postpones evaluation of annotations.

`Self`

When a method returns an instance of its own class, modern Python supports `Self`.

```
from typing import Self

class Builder:
    def set_name(self, name: str) -> Self:
```

```
        self.name = name
        return self
```

This is useful for fluent APIs and method chaining.

Example:

```
builder = Builder()

builder.set_name("Python")
```

Type Hints for Dictionaries With Different Value Types

Suppose a dictionary can contain different kinds of values:

```
user = {
    "name": "Hafsa",
    "age": 25,
    "active": True
}
```

A simple:

```
dict[str, Any]
```

would work:

```
from typing import Any

user: dict[str, Any]
```

But this loses useful structure.

For fixed dictionary structures, `TypedDict` can provide better type information.

`TypedDict`

```
from typing import TypedDict

class User(TypedDict):
    name: str
    age: int
    active: bool
```

Now:

```
user: User = {
    "name": "Hafsa",
    "age": 25,
    "active": True
}
```

The type checker understands the expected keys and their values.

This is especially useful when working with JSON-like data.

Optional Keys in `TypedDict`

A field can be optional.

```
from typing import NotRequired, TypedDict

class User(TypedDict):
    name: str
    age: int
    bio: NotRequired[str]
```

Now `bio` does not have to exist.

`Protocol`

Sometimes you care about what an object can do rather than which class it belongs to.

Python's `Protocol` supports structural typing.

```
from typing import Protocol

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

Any class with a compatible `print_data()` method can satisfy the protocol for static type checking.

Example:

```
class Report:
    def print_data(self) -> str:
        return "Report"


class Invoice:
```

```
    def print_data(self) -> str:
        return "Invoice"
```

A function can accept:

```
def print_document(document: Printable) -> str:
    return document.print_data()
```

This focuses on behavior rather than inheritance.

Type Hints and Duck Typing

Python traditionally follows duck typing:

> If an object behaves like the required object, use it.

Type hints can describe that expected behavior through tools such as `Protocol`.

This allows flexible designs without requiring unnecessary inheritance.

Type Inference

Python development tools can often determine types without explicit annotations.

For example:

```
name = "Hafsa"
```

A type checker can infer:

```
name → str
```

Likewise:

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

can be inferred as:

```
list[int]
```

Therefore, you do not need to annotate every variable.

Prefer useful annotations over annotating every single line.

When Should You Add Type Hints?

Type hints are especially useful for:

Function boundaries

```
def create_user(name: str, age: int) -> User:
    ...
```

Public APIs

```
def get_resources() -> list[Resource]:
    ...
```

Complex data

```
resources: dict[str, list[str]]
```

Important class attributes

```
class Resource:
    title: str
    url: str
```

Large projects

Type hints make collaboration and maintenance easier.

When You Do Not Need Excessive Type Hints

Avoid turning simple code into unnecessary noise.

Instead of:

```
name: str = "Hafsa"
```

everywhere, sometimes this is perfectly readable:

```
name = "Hafsa"
```

Modern tools can infer simple types.

A useful principle is:

> Annotate where the type adds information that would otherwise be unclear.

Type Hints and Static Type Checkers

Type hints become much more powerful when combined with static analysis tools.

Common tools include:

- mypy
- Pyright
- Pylance

- IDE type checking

These tools inspect code without running the program.

For example:

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

Then:

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

A type checker can identify that `"10"` is a string while `20` is an integer.

Python itself may not stop the code simply because the annotation exists.

Mypy Example

Install mypy:

```
pip install mypy
```

Create:

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


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

Run:

```
mypy app.py
```

A type checker can report the incompatible argument.

This lets you find certain bugs before running the application.

Pyright Example

Pyright is another popular static type checker.

It can analyze Python code and report type inconsistencies.

VS Code users may encounter Pyright functionality through Pylance.

The important idea is:

```
Python code
     ↓
```

```
Type hints
     ↓
Static type checker
     ↓
Potential problems detected
```

Type Hints Are Not Validation

This is a critical distinction.

Suppose you write:

```
def register_age(age: int) -> None:
    print(age)
```

This does not automatically validate external input.

If you receive:

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

then:

```
age
```

is a string.

You still need to convert or validate it:

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

Type hints describe expectations.

Validation checks actual data.

These are different responsibilities.

Type Hints vs Runtime Validation

Consider an API receiving JSON.

The incoming data might be:

```
{
    "age": "twenty"
}
```

Your annotation:

```
def create_user(age: int) -> None:
    ...
```

does not magically convert or validate `"twenty"`.

For external data, you may need:

- validation
- parsing
- schema validation
- libraries such as Pydantic

Type hints and runtime validation can work together.

Type Hints and Pydantic

Libraries such as Pydantic use Python type annotations to build runtime data validation.

For example:

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int
```

Then external data can be parsed and validated against the model.

This is particularly useful for:

- APIs
- configuration
- request data
- database boundaries
- external JSON

Type hints alone do not provide this runtime validation.

Type Hints and JSON

Suppose an API returns:

```
{
    "title": "Python",
    "difficulty": "beginner"
}
```

You could represent this with:

```
class Resource(TypedDict):
    title: str
    difficulty: str
```

Then:

```
resource: Resource
```

The structure becomes explicit.

For more complex applications, a model such as a Pydantic model may be more appropriate when runtime validation is required.

Type Hints in Real Projects

Imagine a haas.dev resource service.

Without type hints:

```
def get_resource(id):
    ...
```

The reader has to guess:

- Is `id` a number?
- Is it a string?
- What does the function return?

With type hints:

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

Now the contract is immediately visible:

```
Input:
resource_id → str

Output:
Resource OR None
```

Another example:

```
def search_resources(
    query: str,
    limit: int = 10
```

```
) -> list[Resource]:
    ...
```

The function's expected interface is clear before reading its implementation.

Type Hints as Contracts

A useful mental model is:

> Type hints describe the expected contract between parts of your program.

For example:

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

The contract is:

```
Input:
price    → float
quantity → int

Output:
          → float
```

This makes functions easier to use correctly.

Type Hints Improve IDE Support

Editors can use type information to provide:

- autocomplete
- parameter information
- navigation
- warnings
- refactoring support
- documentation hints

For example:

```
names: list[str] = ["Hafsa", "Asim"]

names.
```

The editor can understand that `names` is a list and suggest list methods.

Type hints therefore improve the development experience even before a type checker is run.

Type Hints and Refactoring

Imagine a function originally returns:

```
list[str]
```

Later you change it to:

```
list[Resource]
```

Type-aware tools can help identify parts of the project that may need updating.

This is especially valuable as projects grow.

Type Hints and Documentation

Compare:

```
def search(query, limit=10):
    ...
```

with:

```
def search(query: str, limit: int = 10) -> list[Resource]:
    ...
```

The second function documents its basic interface directly in the code.

You often need fewer comments to explain obvious type expectations.

Common Type Hint Mistakes

Mistake 1: Thinking Type Hints Enforce Types

This:

```
age: int = "twenty"
```

does not automatically cause Python to raise a type error at runtime.

Use static type checking if you want those issues detected during development.

Mistake 2: Using `Any` Everywhere

Avoid:

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

when you know the actual structure.

Prefer:

```
def process(data: list[str]) -> list[str]:
    ...
```

Mistake 3: Annotating Everything

You do not need:

```
x: int = 10
y: int = 20
z: int = x + y
```

in every small piece of code.

Type inference often handles simple local variables.

Mistake 4: Confusing `list` With `list[str]`

This:

```
items: list
```

only says that `items` is a list.

This:

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

also tells us what the list contains.

When possible, provide useful element types.

Mistake 5: Using the Wrong Return Type

If a function sometimes returns `None`:

```
def find_user(id: int) -> str:
    ...
```

may be incorrect if it can actually return:

```
None
```

Use:

```
def find_user(id: int) -> str | None:
    ...
```

Mistake 6: Treating Type Hints as Validation

This:

```
age: int
```

does not validate user input.

External data still needs validation.

Best Practices

1. Type function boundaries

Prioritize:

```
def calculate(price: float, quantity: int) -> float:
    ...
```

over annotating every local variable.

2. Use specific types

Prefer:

```
list[str]
```

over:

```
list
```

when the element type matters.

3. Avoid unnecessary `Any`

Use `Any` only when flexibility is genuinely required.

4. Use modern syntax

Prefer:

```
str | None
```

over:

```
Optional[str]
```

when your supported Python version allows it.

5. Use abstract collection types when appropriate

Prefer:

```
Sequence[str]
```

when the function only needs sequence behavior rather than specifically requiring a list.

6. Keep annotations readable

If an annotation becomes extremely complicated, introduce a type alias or model.

7. Use a type checker on larger projects

Type hints become much more useful when paired with tools such as mypy or Pyright.

8. Keep runtime validation separate

Use validation tools when dealing with untrusted or external data.

Type Hints vs Comments

Without type hints:

```
# name should be a string
# age should be an integer
def create_user(name, age):
    ...
```

With type hints:

```
def create_user(name: str, age: int) -> User:
    ...
```

Type hints are structured information that development tools can understand.

Comments generally cannot provide the same level of tooling support.

Type Hints vs Docstrings

Docstrings explain what a function does:

```
def calculate_total(price: float, quantity: int) -> float:
    """Calculate the total price."""
    return price * quantity
```

The type hints explain the data contract.

The docstring explains the behavior.

They complement each other.

Type Hints and `__annotations__`

Python stores annotations on functions and classes.

Example:

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

You can inspect:

```
print(greet.__annotations__)
```

You may see:

```
{'name': <class 'str'>, 'return': <class 'str'>}
```

This demonstrates that annotations are available as Python metadata.

However, do not assume that annotations automatically perform validation.

`typing.get_type_hints()`

Python provides a helper for retrieving resolved type hints.

```
from typing import get_type_hints

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

print(get_type_hints(greet))
```

This can be useful for frameworks and advanced tooling.

Type Hints and Generics

Suppose you want a reusable function that works with many types while preserving the relationship between input and output.

A generic can express this.

```
from typing import TypeVar

T = TypeVar("T")

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

Now:

```
numbers = first_item([1, 2, 3])
names = first_item(["Hafsa", "Asim"])
```

The type relationship is preserved:

```
list[int]    → int
list[str]    → str
```

Modern Generic Syntax

Modern Python versions also support type parameter syntax.

For example:

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

This is useful when targeting Python versions that support the newer generic syntax.

Always consider the Python version your project supports before using newer syntax.

A Complete Example

Let's build a small resource system.

```
from dataclasses import dataclass


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


def create_resource(
    title: str,
    category: str,
    pages: int
) -> Resource:
    return Resource(
        title=title,
        category=category,
        pages=pages
```

```
    )


def get_resources() -> list[Resource]:
    return [
        create_resource(
            "Python Type Hints",
            "Python",
            20
        ),
        create_resource(
            "Python Testing",
            "Python",
            25
        )
    ]


def find_resource(
    resources: list[Resource],
    title: str
) -> Resource | None:

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

    return None
```

Now the structure is easy to understand.

```
Resource
   ↓
title → str
category → str
pages → int

create_resource()
   ↓
Resource

get_resources()
   ↓
```

```
  list[Resource]

  find_resource()
     ↓
  Resource | None
```

Type hints make the relationships between different parts of the program explicit.

A Practical Workflow for Adding Type Hints

Do not try to annotate an entire large project randomly.

Use this workflow:

Step 1: Start With Public Functions

Find important functions:

```
  def search_resources(...):
```

Add annotations.

Step 2: Annotate Return Values

```
  def search_resources(...) -> list[Resource]:
```

Step 3: Annotate Parameters

```
  def search_resources(query: str, limit: int) -> list[Resource]:
```

Step 4: Annotate Important Attributes

```
  class Resource:
      title: str
      category: str
```

Step 5: Handle Optional Values

If something can be missing:

```
  Resource | None
```

Step 6: Run a Type Checker

Use tools such as:

```
  mypy .
```

or configure Pyright/Pylance.

Step 7: Fix Real Problems

Do not simply silence every warning.

Understand why the types disagree.

Type Hints in Team Development

Imagine three developers working on the same application.

Developer A writes:

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

Developer B immediately knows:

```
user_id must be an integer
result may be User
result may be None
```

Developer C can rely on the same contract while building another part of the application.

This reduces misunderstandings between modules.

Type Hints and API Design

Type hints are particularly useful when designing service boundaries.

For example:

```
def create_resource(
    title: str,
    category: str,
    tags: list[str]
) -> Resource:
    ...
```

The function communicates its interface clearly.

A caller does not have to read the entire implementation just to understand the expected data.

Type Hints and Testing

Type hints and tests solve different problems.

Tests answer:

> Does the program behave correctly?

Type checkers answer:

> Are the values being used according to their declared types?

For example:

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

A test can verify:

```
assert add(2, 3) == 5
```

A type checker can identify:

```
add("2", 3)
```

These tools complement each other.

Type Hints and Dataclasses

Dataclasses work especially well with type hints.

```
from dataclasses import dataclass


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

The annotations define the expected fields.

This makes dataclasses concise and readable.

Type Hints and IDEs

Modern Python IDEs can use type hints to provide:

- autocomplete
- type warnings
- parameter hints
- navigation
- refactoring assistance
- documentation
- error highlighting

For many developers, this is one of the immediate benefits of using type hints.

A Simple Mental Model

Think of type hints as **labels on the roads of your program**.

Without them:

```
function
   ↓
??? → ???
   ↓
???
```

With them:

```
str + int
   ↓
function
   ↓
Resource | None
```

They do not necessarily stop the car from going the wrong way.

They make the intended route visible to developers and tools.

---

5 Minute Challenge

Create a function:

```
def calculate_average(...)
```

Requirements:

1. Accept a list of integers.
2. Return a float.
3. Add type hints.
4. Test it with:

```
[10, 20, 30]
```

Expected result:

```
20.0
```

Then intentionally call it with incorrect data and see what your type checker reports.

Exercises

Exercise 1: User Function

Create:

```
def create_user(...):
    ...
```

It should accept:

```
name → str
age → int
```

and return a dictionary containing the user information.

Add complete type hints.

Exercise 2: Search Function

Create:

```
def search_products(...)
```

Requirements:

```
query → str
products → list[str]
return → list[str]
```

Return products containing the search query.

Exercise 3: Optional Return

Create:

```
def find_product(...)
```

It should return:

```
str | None
```

when a product may or may not exist.

Exercise 4: Nested Data

Create a type hint for:

```
[
    {
        "name": "Python",
        "level": "beginner"
    }
]
```

Use an appropriate type representation.

Exercise 5: Callback

Create a function that accepts:

```
a number
a function that receives an integer and returns an integer
```

Use `Callable`.

Mini Project: Typed Resource Manager

Build a small resource manager for haas.dev.

Create a `Resource` model containing:

```
title
category
pages
```

Add type hints to:

```
Resource
add_resource()
get_resources()
find_resource()
remove_resource()
```

Requirements:

- use `list[Resource]`
- use `Resource | None` where appropriate
- type every function parameter
- type every return value
- use a static type checker
- intentionally introduce a type mismatch

- observe the type checker warning
- fix the problem

Example:

```
from dataclasses import dataclass


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


resources: list[Resource] = []


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


def get_resources() -> list[Resource]:
    return resources


def find_resource(title: str) -> Resource | None:
    for resource in resources:
        if resource.title == title:
            return resource

    return None
```

This small project demonstrates how type hints connect multiple parts of an application.

Common Questions

Are type hints required in Python?

No.

Python code can work without type hints.

They are optional, but increasingly useful in professional and larger codebases.

Do type hints make Python statically typed?

Not by themselves.

Python remains dynamically typed.

Type checkers can perform static analysis using the annotations.

Do type hints improve performance?

Usually, type hints are primarily for readability and tooling.

They are not a general runtime performance optimization.

Can Python ignore type hints?

At runtime, Python generally does not enforce ordinary annotations automatically.

A program can still receive values that do not match the annotation.

Should every variable have a type hint?

No.

Focus on important interfaces, complex data, public APIs, and places where the expected type is unclear.

Should I use `Any` when I am unsure?

Not automatically.

If the structure is known, represent it accurately.

Use `Any` when unrestricted typing is genuinely appropriate.

Are type hints the same as validation?

No.

Type hints communicate expected types.

Validation checks actual runtime data.

Interview Questions

1. What are type hints in Python?

Type hints are annotations that communicate the expected types of variables, parameters, attributes, and return values.

2. Does Python enforce type hints at runtime?

Normally, no.

Static type checkers and development tools can use them to detect inconsistencies.

3. What does this mean?

```
def get_user(id: int) -> User | None:
```

The function expects an integer ID and may return either a `User` or `None`.

4. What is `Any`?

`Any` tells static type checkers to allow values without enforcing a specific type.

5. What is the difference between `list` and `list[str]`?

`list` only specifies that the value is a list.

`list[str]` specifies that it is expected to contain strings.

6. What is `Callable`?

`Callable` describes a callable object such as a function, including its expected arguments and return type.

7. What is `TypedDict`?

`TypedDict` describes the expected keys and value types of dictionary-like data.

8. What is a `Protocol`?

A `Protocol` defines expected behavior for structural typing without requiring explicit inheritance.

9. What is `TypeVar` used for?

`TypeVar` allows generic functions and classes to preserve relationships between input and output types.

10. How are type hints related to static type checking?

Type hints provide information that tools such as mypy and Pyright can analyze without executing the program.

Type Hints Cheat Sheet

Variables

```
name: str = "Hafsa"
age: int = 25
price: float = 99.9
active: bool = True
```

Function

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

List

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

Dictionary

```
scores: dict[str, int]
```

Tuple

```
point: tuple[float, float]
```

Set

```
tags: set[str]
```

Optional Value

```
name: str | None
```

Union

```
value: int | str
```

Any

```
from typing import Any

data: Any
```

Callable

```
from collections.abc import Callable

operation: Callable[[int], int]
```

Iterable

```
from collections.abc import Iterable

items: Iterable[str]
```

Sequence

```
from collections.abc import Sequence

items: Sequence[str]
```

Typed Dictionary

```
from typing import TypedDict

class User(TypedDict):
    name: str
    age: int
```

Literal

```
from typing import Literal

status: Literal["draft", "published"]
```

Final

```
from typing import Final

MAX_USERS: Final = 100
```

Generic

```
from typing import TypeVar

T = TypeVar("T")
```

Self

```
from typing import Self

def build(self) -> Self:
    return self
```

Type Alias

```
User = dict[str, str]
```

Static Checking

```
pip install mypy
```

Then:

```
mypy .
```

Key Takeaways

1. Type hints describe expected data types.
2. They improve readability and maintainability.
3. They help IDEs and static type checkers understand your code.
4. Python does not normally enforce ordinary type hints at runtime.
5. Use annotations especially at function and module boundaries.
6. `list[str]`, `dict[str, int]`, and similar syntax describe container contents.
7. `str | None` represents a value that may be a string or `None`.
8. `Any` should be used carefully.
9. `TypedDict` is useful for structured dictionary data.
10. `Protocol` describes behavior rather than requiring inheritance.
11. `Callable` describes functions passed as values.
12. Generic types allow reusable type-safe designs.
13. Type hints are not a replacement for runtime validation.
14. Type checkers such as mypy and Pyright make type hints significantly more useful.
15. Good type hints make large Python applications easier to understand, refactor, and maintain.
