13
Type Hints

Introduction to Type Hints in Python

Python is a dynamically typed language, which means variables and function arguments do not require explicit type declarations. This flexibility allows for rapid development but can also lead to hard-to-debug errors, especially in large codebases. As projects grow, maintaining code clarity and preventing type-related issues becomes increasingly important.

To address these challenges, Python introduced **type hints** in PEP 484, allowing developers to specify expected data types for variables, function parameters, and return values. Type hints do not enforce type safety at runtime, but they provide valuable information to developers, improve code readability, and enable static type checkers like mypy to catch potential errors before execution. Modern IDEs also leverage type hints to offer better autocomplete suggestions, linting, and real-time error detection, making development smoother and more efficient.

Despite these advantages, Python remains as flexible as ever. Type hints are **optional** and can be used selectively. They enhance documentation without restricting Python's dynamic capabilities, allowing developers to strike a balance between strict type checking and the language's inherent expressiveness.

This chapter explores the various aspects of type hints, from basic annotations to more advanced features like generics, protocols, and structural subtyping. By the end, you will understand how to integrate type hints effectively into your codebase, improve maintainability, and leverage static analysis tools to catch potential issues early.

Contents of This Chapter

- **Introduction to Type Hints** – Understanding what type hints are, their benefits, and how Python remains dynamically typed.
- **Basic Type Annotations** – Applying type hints to variables, function parameters, and return values using built-in types.
- **Using the typing Module** – Exploring advanced type annotations such as Optional, Union, and type aliases.
- **Type Hints for Functions and Methods** – Annotating instance methods, class methods, and higher-order functions with Callable.
- **Type Hints for Custom Classes** – Using user-defined classes as type annotations and handling forward references.
- **Type Hints for Special Cases** – Leveraging Any, NoReturn, Literal, and Final for unique type hinting scenarios.
- **Generic Types and TypeVar** – Implementing reusable, type-safe functions and classes with generics.
- **Protocols and Structural Subtyping** – Defining behavior-based interfaces without explicit class inheritance.

- **Type Checking and Static Analysis** – Using mypy, pyright, and pylance to detect type errors before execution.
- **Working with Type Hints and Importing Untyped Modules** – Handling third-party libraries and modules that lack type hints.
- **Best Practices and Performance Considerations** – Balancing type safety and readability while avoiding excessive complexity.

By the end of this chapter, you will be equipped with the knowledge to use type hints effectively, improving both the robustness and clarity of your Python programs while retaining the language's dynamic nature.

Introduction to Type Hints

What Are Type Hints?

Type hints are a feature in Python that allow developers to specify the expected data types of variables, function parameters, and return values. Introduced in **PEP 484** (Python 3.5), type hints provide a way to indicate what types should be used in a program without enforcing strict type checking at runtime.

For example, a function with type hints:

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

Here, name: str specifies that name should be a string, and -> str indicates that the function returns a string. However, Python does **not** enforce these types, meaning the function will still execute even if an incorrect type is passed.

Benefits of Using Type Hints in Python

1. Improved Code Readability

Type hints make it clear what types are expected, reducing ambiguity and making code easier to understand for both the author and other developers.

2. Enhanced Developer Productivity

Modern IDEs and code editors can use type hints to provide **autocomplete suggestions, better linting, and real-time error detection**, reducing debugging time.

3. Static Type Checking

While Python does not enforce type hints at runtime, tools like **mypy**, **Pyright**, and **Pylance** can analyze code statically, catching type-related errors before execution.

4. Better Documentation

Explicit type annotations serve as **self-documenting code**, reducing the need for additional comments and external documentation to explain function signatures.

5. Easier Refactoring and Maintenance

When refactoring large codebases, type hints help ensure that function calls and assignments remain consistent, reducing the risk of introducing bugs.

How Python Remains Dynamically Typed Despite Type Hints

Python is inherently **dynamically typed**, meaning variable types are determined at runtime and can change. Type hints do **not** alter this behavior. Unlike statically typed languages (e.g., Java, C++), Python does not enforce type safety at runtime.

For example, even if a function has type hints, Python will not raise an error if an incorrect type is passed:

```
def add(x: int, y: int) -> int:
    return x + y


print(add(5, "hello"))  # No runtime error, but incorrect behavior
```

Despite this, **static type checkers** like mypy will detect such issues before execution:

```
$ mypy script.py
```

error: Argument 2 to "add" has incompatible type "str"; expected "int"

Thus, Python remains **dynamically typed**, with type hints serving as **optional guidance** rather than strict rules enforced by the interpreter. This allows flexibility while enabling better code quality through static analysis.

Basic Type Annotations

Type annotations in Python provide a way to indicate the expected data types for variables and function parameters. While Python does not enforce these annotations at runtime, they improve **code clarity, documentation, and static type checking**.

Annotating Variables

In Python, you can specify the type of a variable using a colon (:) followed by the expected type.

```
x: int = 10
y: float = 3.14
name: str = "Alice"
is_active: bool = True
```

Even though these variables have explicit type hints, Python still allows type changes at runtime:

```
x = "Hello"  # No runtime error, but static type checkers will flag this as an issue.
```

Function Parameter and Return Type Hints

Functions can have type hints for **parameters** and **return values**, making their expected usage clear.

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

- a: int and b: int indicate that both parameters should be integers.
- -> int specifies that the function should return an integer.

Examples with Different Return Types

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


def is_even(n: int) -> bool:
    return n % 2 == 0


def get_pi() -> float:
    return 3.14159
```

- The greet() function takes a string and returns a string.
- The is_even() function takes an integer and returns a boolean.
- The get_pi() function returns a floating-point number.

Type Hints for Built-in Types

Python supports type hints for all built-in data types, including collections like **lists, tuples, dictionaries, and sets**.

Type	Example Usage
int	age: int = 25
float	price: float = 19.99
str	name: str = "Alice"
bool	is_valid: bool = True
list	numbers: list[int] = [1, 2, 3]
tuple	coordinates: tuple[float, float] = (10.5, 20.8)
dict	user: dict[str, int] = {"Alice": 30, "Bob": 25}
set	unique_numbers: set[int] = {1, 2, 3, 4}

Examples of Type Hints for Collections

```
numbers: list[int] = [1, 2, 3, 4, 5]
```

```
grades: dict[str, float] = {"Alice": 90.5, "Bob": 88.0}

colors: set[str] = {"red", "blue", "green"}

point: tuple[int, int] = (10, 20)
```

In older versions of Python (before 3.9), you had to import these types from the typing module:

```
from typing import List, Dict, Tuple, Set


numbers: List[int] = [1, 2, 3]

grades: Dict[str, float] = {"Alice": 90.5, "Bob": 88.0}

point: Tuple[int, int] = (10, 20)
```

Starting from **Python 3.9**, built-in generics like list[int] and dict[str, int] are preferred over List[int] and Dict[str, int].

Summary

- Type hints can be used for **variables** (x: int = 10) and **function signatures** (def add(a: int, b: int) -> int).
- Python supports type hints for primitive types (int, float, str, bool) and collections (list, tuple, dict, set).
- Type hints **do not enforce types at runtime** but help with **static analysis** and **code readability**.
- In **Python 3.9+**, built-in type hints like list[int] and dict[str, int] are preferred.

Using type hints makes Python code **more readable, maintainable, and reliable**, helping developers catch potential errors before execution.

Using the typing Module

The typing module in Python provides advanced type hints, allowing developers to specify more complex types, including optional values and multiple possible types. This helps improve code clarity and ensures better static type checking.

Handling None Values with Optional

In Python, some function parameters or variables may accept None as a valid value. To indicate this, we use Optional from the typing module.

Syntax:

```
from typing import Optional


def get_username(user_id: int) -> Optional[str]:
    if user_id == 1:
        return "Alice"
```

```
    return None
```

How It Works:

- Optional[str] means that the function may return either a str **or** None.
- This is equivalent to writing Union[str, None] but is more readable.
- If None is a valid return type or input, always specify Optional.

Example Usage:

```
username = get_username(2)
if username is not None:
    print(f"User: {username}")
else:
    print("User not found")
```

Without Optional, a static type checker like mypy might raise warnings when handling None.

Using Union for Multiple Possible Types

The Union type hint allows a variable or function parameter to accept multiple data types.

Syntax:

```
from typing import Union

def format_value(value: Union[int, str]) -> str:
    return f"Formatted: {value}"
```

How It Works:

- Union[int, str] means value can be either an int or a str.
- This is useful when a function needs to handle multiple data types in a controlled way.

Starting with Python 3.10, you can simplify your type hints by using the new union syntax (|):

```
def format_value (value: int | str) -> None:
    return f"Formatted: {value}"
```

This is equivalent to Union[int, str] but is shorter and clearer.

Example Usage:

```
print(format_value(100))  # Formatted: 100
print(format_value("Hello"))  # Formatted: Hello
```

Using Union helps ensure **type safety** while allowing flexibility.

When to Use Optional vs. Union

Scenario	Use

A value can be None or another type	Optional[T] (e.g., Optional[str])
A value can be multiple different types (excluding None)	Union[T1, T2] (e.g., Union[int, float])
A value can be multiple types including None	Optional[T] (equivalent to Union[T, None])

Summary

- Optional[T] is used when a value **can be None or type T** (e.g., Optional[str]).
- Union[T1, T2] allows specifying **multiple valid types** (e.g., Union[int, str]).

Type Hints for Methods

Type hints improve function and method documentation by specifying the expected argument and return types. This section explores how to apply type hints to **function parameters, return values, class methods**, and **functions passed as arguments** using Callable.

Hinting Methods Inside a Class (self and cls Annotations)

Instance Methods (self)

When defining instance methods, use self as the first parameter and apply type hints to parameters and return values as usual.

```
class Car:
    def __init__(self, brand: str, speed: int = 0):
        self.brand: str = brand
        self.speed: int = speed

    def accelerate(self, increase: int) -> None:
        self.speed += increase
```

- self.brand: str and self.speed: int – Type hints for instance attributes.
- increase: int – The accelerate method expects an integer parameter.
- -> None – The method does **not** return a value.

Class Methods (cls)

Class methods operate at the class level and use cls instead of self. Type hints apply in the same way.

```
from typing import Type

class Car:
```

```
    count: int = 0

    def __init__(self, brand: str):
        self.brand = brand
        Car.count += 1

    @classmethod
    def total_cars(cls: Type["Car"]) -> int:
        return cls.count
```

cls: Type["Car"] – Type hint for cls, referring to the class itself.

-> int – The method returns an integer.

Using Type["Car"] ensures that subclasses can return their own class type.

Using Callable for Function Arguments

Sometimes, a function needs to accept another function as a parameter. In such cases, we use Callable from the typing module.

Syntax of Callable

```
from typing import Callable


Callable[[ArgType1, ArgType2, ...], ReturnType]
```

Example: Passing a Function as an Argument

```
from typing import Callable


def apply_operation(x: int, y: int, operation: Callable[[int, int], int]) -> int:
    return operation(x, y)


def multiply(a: int, b: int) -> int:
    return a * b


result = apply_operation(5, 3, multiply)  # Calls multiply(5, 3)
print(result)  # Output: 15
```

- operation: Callable[[int, int], int] – The function must take two int arguments and return an int.

- The multiply function matches this signature and is passed as an argument.

Using Callable ensures that only functions with the correct signature can be passed.

Summary

- **Function type hints** specify expected parameter types and return types (def add(a: int, b: int) -> int:).
- **Instance methods** use self, and attributes can have type annotations (self.brand: str).
- **Class methods** use cls: Type["ClassName"] to refer to the class type.
- **Callable** allows type hinting for function arguments, ensuring type-safe higher-order functions.

These type hints improve **readability, maintainability, and static type checking**, helping developers catch potential errors before runtime.

Type Hints for Custom Classes

Type hints are not limited to built-in types like int, str, and list. You can also use **user-defined classes** as type annotations. This allows for better clarity and static type checking when working with object-oriented programming (OOP) in Python.

Hinting with User-Defined Classes

When defining a custom class, you can use it as a type hint for function parameters and return values.

Example: Type Hinting with a Custom Class

```
class Car:
    def __init__(self, brand: str, speed: int):
        self.brand: str = brand
        self.speed: int = speed

    def accelerate(self, increase: int) -> None:
        self.speed += increase
```

- brand: str and speed: int define the attributes with type hints.
- The accelerate method takes an increase parameter of type int and returns None.

Using Classes as Type Hints in Function Parameters and Return Values

Once a class is defined, it can be used as a type annotation in function signatures.

Example: Using a Class as a Type Hint

```
def display_car_info(car: Car) -> str:
    return f"{car.brand} is moving at {car.speed} km/h."
```

```
my_car = Car("Toyota", 100)
print(display_car_info(my_car))  # Output: Toyota is moving at 100 km/h.
```

- car: Car specifies that the function expects a Car instance as an argument.
- -> str means that the function returns a string.

You can also return a custom class from a function.

Example: Returning an Instance of a Custom Class

```
def create_car(brand: str, speed: int) -> Car:
    return Car(brand, speed)

new_car = create_car("Honda", 120)
print(display_car_info(new_car))  # Output: Honda is moving at 120 km/h.
```

- -> Car means the function returns an instance of the Car class.

Forward References for Self-Referencing Types

Sometimes, a class may reference itself, such as in tree structures or linked lists. Since Python processes code line by line, using a class inside its own definition can lead to a **NameError**. To avoid this, **forward references** are used.

From **Python 3.7+,** you can enclose the class name in **quotes ("ClassName")** to refer to it before it is fully defined.

Example: Using Forward References

```
class Node:
    def __init__(self, value: int, next_node: "Node" = None):
        self.value: int = value
        self.next_node: "Node" = next_node  # Forward reference
```

- "Node" is enclosed in quotes because the class is not yet fully defined when it is used as a type hint.

From **Python 3.10+,** forward references are automatically resolved, so quotes are no longer necessary.

For **Python 3.7+**, you can also use:

```
from __future__ import annotations  # Enables automatic forward reference resolution

class Node:
```

```
    def __init__(self, value: int, next_node: Node = None):
        self.value: int = value
        self.next_node: Node = next_node  # No need for quotes
```

- from __future__ import annotations allows referencing Node without quotes, even before its definition.

Summary

- **User-defined classes** can be used as type hints for parameters and return values.
- **Function arguments** can be annotated with a class name to indicate they expect an instance of that class.
- **Forward references** ("ClassName") are used when a class references itself.
- **Python 3.7+** allows enabling forward references without quotes using from __future__ import annotations.

Using type hints for custom classes improves **code clarity, maintainability, and static type checking**.

Type Hints for Special Cases

Python's type hints support **special cases** where standard annotations are not enough. The typing module provides additional type hints to handle cases like **disabling type checking, defining functions that never return, restricting specific values, and preventing subclassing or reassignment**.

Any for Disabling Type Checking

The Any type allows **bypassing type checking** for a variable, function parameter, or return type. It is useful when working with **dynamically typed data**, **third-party libraries without type hints**, or cases where strict type checking is unnecessary.

Example: Using Any to Allow Any Type

```
from typing import Any


def process_data(data: Any) -> None:
    print(f"Processing: {data}")


value: Any = 42
value = "Hello"  # Allowed, since Any disables type checking
```

- data: Any means that process_data can accept any type.
- value: Any can hold different types at different points.

Even when using mypy, Any suppresses type errors, making it useful for flexibility.

NoReturn for Functions That Never Return

The NoReturn type is used for **functions that do not return a value and never complete execution** (e.g., functions that raise exceptions or cause program termination).

Example: Using NoReturn for Functions That Exit the Program

```
from typing import NoReturn


def fatal_error(message: str) -> NoReturn:
    raise RuntimeError(f"Fatal Error: {message}")
```

- The function **never reaches a return statement** because it always raises an exception.
- NoReturn helps type checkers understand that the function will **not return a valid result**.

Literal for Restricting Specific Values

The Literal type allows restricting a variable or function parameter to **specific allowed values**. This is useful for enforcing strict input values.

Example: Using Literal for Strict Input Validation

```
from typing import Literal


def set_mode(mode: Literal["auto", "manual"]) -> str:
    return f"Mode set to {mode}"


mode1 = set_mode("auto")   # ✅ Valid
mode2 = set_mode("manual") # ✅ Valid
# mode3 = set_mode("fast")  # ❌ Error: "fast" is not a valid
Literal value
```

- mode: Literal["auto", "manual"] means the function **only** accepts "auto" or "manual".
- Any other value will cause a **static type checking error** (e.g., with mypy).

Using Literal helps prevent **unexpected input values** and ensures consistency.

Final to Prevent Subclassing or Attribute Reassignment

The Final type is used to:

1. **Prevent attributes from being reassigned** in subclasses.
2. **Prevent a class from being subclassed** by marking it as Final.

Example: Preventing Attribute Reassignment

```
from typing import Final
```

```
class Settings:
    API_KEY: Final[str] = "12345"

settings = Settings()
```

settings.API_KEY = "67890" # ❌ Error: Cannot reassign a Final attribute

- API_KEY: Final[str] ensures the value **cannot be changed after initialization**.
- Trying to modify it will **raise a type checking error** in static analysis tools.

Example: Preventing a Class from Being Subclassed

```
from typing import final

@final
class Base:
    pass
```

class Derived(Base): # ❌ Error: Cannot subclass a @final class

pass

- Marking a class with @final prevents any other class from **extending** it.
- This is useful for enforcing **strict inheritance rules** in APIs or frameworks.

Summary

Special Type	Purpose
Any	Disables type checking, allowing dynamic types.
NoReturn	Used for functions that **never return** (e.g., raise exceptions, exit programs).
Literal	Restricts a value to a predefined set of choices (e.g., "yes", "no").
Final	Prevents reassignment of attributes or subclassing of classes.

These **special type hints** help enforce stricter type safety and maintainability in Python programs, while still allowing flexibility where needed.

Generic Types and TypeVar

Generics allow us to write **reusable, type-safe** functions and classes that can work with multiple types while maintaining **type constraints**. The typing module provides the TypeVar and Generic classes to implement generics in Python.

Creating Reusable, Type-Safe Functions with TypeVar

When defining functions, we often want to allow **flexibility** in the input types while ensuring **type safety**. Instead of using Any, which disables type checking, we can use TypeVar to define a placeholder type.

Example: Using TypeVar in a Function

```
from typing import TypeVar

T = TypeVar("T")  # A generic type placeholder

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

print(get_first_element([1, 2, 3]))    # Output: 1 (int)
print(get_first_element(["a", "b"]))   # Output: "a" (str)
```

- T = TypeVar("T") declares a generic type variable.
- get_first_element(items: list[T]) -> T ensures that the **input and return type remain the same**.
- If list[int] is passed, T becomes int; if list[str] is passed, T becomes str.

Using TypeVar ensures **type consistency** across function parameters and return values.

Generics in Function Arguments and Class Definitions

Just like functions, we can define **generic classes** where the type of attributes or methods is determined dynamically.

Example: Creating a Generic Class

```
from typing import Generic

T = TypeVar("T")  # Define a generic type variable

class Box(Generic[T]):  # A class that works with any type T
   def __init__(self, value: T):
      self.value: T = value

   def get_value(self) -> T:
```

```
        return self.value


int_box = Box(100)      # Box[int]
str_box = Box("Hello")   # Box[str]


print(int_box.get_value())  # Output: 100 (int)
print(str_box.get_value())  # Output: Hello (str)
```

- Box[T] is a **generic class** where T represents the type stored inside the box.
- self.value: T ensures that the stored value is of type T.
- get_value() returns T, ensuring that the return type matches the input type.

Generic Base Class for Custom Generic Types

When creating more complex class hierarchies, we may need to define **generic base classes**. This allows child classes to **inherit** type behavior from a generic parent class.

Example: Creating a Generic Base Class

```
from typing import Generic, TypeVar

T = TypeVar("T")

class Storage(Generic[T]):  # Generic base class
    def __init__(self, item: T):
        self.item = item

    def get_item(self) -> T:
        return self.item

class NumberStorage(Storage[int]):  # A subclass with a specific type
    def double(self) -> int:
        return self.item * 2

num_storage = NumberStorage(10)
print(num_storage.get_item())  # Output: 10
```

```
print(num_storage.double())     # Output: 20
```

- Storage[T] is a **generic base class** that can store any type T.
- NumberStorage(Storage[int]) is a subclass that **restricts T to int**.
- The double() method ensures that self.item remains an integer.

Using **generic base classes** allows us to **enforce type safety** while **reusing code** in different subclasses.

Summary

Concept	Usage
TypeVar	Defines a **placeholder type** for generic functions.
Generic Functions	Enable **reusable** and **type-safe** operations on different types.
Generic Classes	Allow attributes and methods to work **dynamically with any type**.
Generic Base Classes	Create **type-safe** and **extensible** hierarchies for multiple subclasses.

Generics improve **code reusability, flexibility, and type safety**, making Python programs more **scalable and maintainable**.

Protocols and Structural Subtyping

Python's **Protocols** and **structural subtyping** allow us to define behavior without relying on **class inheritance**. This is useful when we want to specify that an object must implement certain methods, regardless of its class hierarchy. This approach aligns with **duck typing**, where an object's behavior is more important than its explicit type.

What Are Protocols?

A **protocol** in Python is similar to an **interface** in other programming languages. It defines a set of methods and attributes that a class must implement but **does not require explicit inheritance**.

Python's typing module provides the **Protocol** class, which enables **structural subtyping**—meaning an object is considered valid **as long as it implements the required methods, regardless of its class**.

This contrasts with **nominal typing** (traditional inheritance), where a class must explicitly inherit from another to be recognized as compatible.

Using Protocol to Define Behavior Without Inheritance

Instead of forcing a class to inherit from a base class, we can define a **protocol** that specifies the required methods.

Example: Defining a Protocol for a FileWriter

```
from typing import Protocol


```

```
class FileWriter(Protocol):  # Defines expected behavior
    def write(self, data: str) -> None:
        ...

class TextFile:
    def write(self, data: str) -> None:
        print(f"Writing to text file: {data}")

class Logger:
    def write(self, data: str) -> None:
        print(f"Logging data: {data}")

def save_data(writer: FileWriter, data: str) -> None:
    writer.write(data)

text_file = TextFile()
logger = Logger()

save_data(text_file, "Hello, world!")  # Writing to text file: Hello, world!
save_data(logger, "Error log")         # Logging data: Error log
```

Key Takeaways:

- FileWriter is a **protocol** that expects a write() method.
- **Neither TextFile nor Logger inherit from FileWriter**, yet they are considered valid because they implement write().
- The function save_data(writer: FileWriter, data: str) -> None can accept any object that matches the protocol, regardless of its class.

This is an example of **structural subtyping**, where the compatibility is determined **by behavior, not inheritance**.

Duck Typing with Type Hints

Python naturally follows **duck typing**, meaning "If it **quacks** like a duck and **walks** like a duck, it must be a duck."

With **protocols and type hints**, we can formally define duck-typed behavior while ensuring **static type safety**.

Example: Using Duck Typing for a Messaging System

```
class Messenger(Protocol):
    def send_message(self, message: str) -> None:
        ...

class Email:
    def send_message(self, message: str) -> None:
        print(f"Sending email: {message}")

class SMS:
    def send_message(self, message: str) -> None:
        print(f"Sending SMS: {message}")

def notify(messenger: Messenger, message: str) -> None:
    messenger.send_message(message)

email = Email()
sms = SMS()

notify(email, "Hello via email!")  # Sending email: Hello via email!
notify(sms, "Hello via SMS!")      # Sending SMS: Hello via SMS!
```

How Duck Typing Works Here:

- Messenger is a **protocol** that requires a send_message() method.
- Email and SMS **do not inherit from Messenger**, yet they work because they implement the required method.
- The notify() function can accept **any object** that conforms to the Messenger protocol.

Summary

Concept	Explanation
Protocols	Define required behavior without enforcing class inheritance.
Structural Subtyping	Objects are type-compatible if they implement the required methods, even if they don't inherit from a specific class.

Duck Typing	Objects are considered valid based on behavior rather than explicit type declarations.
Flexible Type Checking	Protocol enables static type checking while keeping the flexibility of duck typing.

Using **protocols and structural subtyping** makes Python code **more flexible, maintainable, and type-safe**, especially when designing **interfaces, APIs, and loosely coupled systems**.

Type Checking and Static Analysis

Python is dynamically typed, meaning it does not enforce type hints at runtime. However, **static analysis tools** like mypy, pyright, and pylance allow developers to catch **type errors** before execution. This improves code quality, maintainability, and reduces bugs in large projects.

Running mypy for Static Type Checking

mypy is a popular **static type checker** for Python. It analyzes code without executing it and ensures that type hints are correctly followed.

Installing mypy

To install mypy, use:

```
pip install mypy
```

Running mypy on a Python File

```
mypy script.py
```

If the file contains type inconsistencies, mypy will report errors.

Example: Detecting Type Errors

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


print(add(5, "hello"))  # ❌ Incorrect: Passing a string instead of an int
```

Running mypy script.py produces:

```
error: Argument 2 to "add" has incompatible type "str"; expected "int"
```

This helps catch bugs **before running the program**.

Ignoring Specific Lines

If needed, you can disable type checking for a specific line:

```
result = add(5, "hello")  # type: ignore
```

Use # type: ignore cautiously, as it bypasses type checking.

Using pyright or pylance for Type Checking in IDEs

1. pyright for Command Line and IDEs

pyright is a **fast static type checker** developed by Microsoft. It can be used in the **command line** or integrated into **VS Code**.

Installing pyright

```
pip install pyright
```

Running pyright on a File

```
pyright script.py
```

- It provides **real-time** feedback with detailed error messages.
- Unlike mypy, pyright supports **TypeScript-like gradual typing** for improved performance.

2. pylance for VS Code

- **pylance** is the official **VS Code extension** for Python, built on pyright.
- It provides **inline error highlighting**, **autocomplete**, and **type inference**.
- To enable it, install the **Python extension** in VS Code and select pylance as the language server.

Common Type Hinting Errors and How to Fix Them

Error Type	Example	Fix
Type Mismatch	def add(a: int, b: int) -> int: add(5, "hello")	Ensure arguments match the expected types.
Missing Type Hints	def greet(name):	Use explicit type hints: def greet(name: str) -> str:
None Return Type Not Handled	def get_user() -> str: return None	Use Optional[str] if None is a possible return value.
Incorrect Union Usage	Union[int, str] = 42	Assign a value inside a function, not directly to Union.
Using a Forward Reference Without from __future__ import annotations	def parent(child: Parent) -> None:	Use "Parent" as a forward reference or enable from __future__ import annotations.

Summary

- **mypy** is a powerful static type checker for Python.
- **pyright** and **pylance** provide **fast, real-time** type checking in IDEs like VS Code.
- **Common errors** include type mismatches, missing hints, and incorrect Union or Optional usage.

- **Static analysis tools improve code quality**, reduce runtime errors, and enhance developer productivity.

Working with Type Hints and Importing Modules Without Type Hints

When working with **type hints in Python**, you may encounter third-party libraries or built-in modules that **do not provide type hints**. Since Python is dynamically typed, older or external modules may not have explicit type annotations, which can cause challenges for static type checkers like mypy. This guide explains **how to handle such cases** effectively.

Using Any for Unknown or Unannotated Types

The simplest way to work with untyped modules is to use Any from the typing module. This disables strict type checking for the variable or return type.

Example: Importing an Untyped Module

```
from typing import Any
import some_untyped_module  # Hypothetical third-party module


def process_data(data: Any) -> None:
    result = some_untyped_module.process(data)  # No type checking
    print(result)
```

- data: Any means the function can accept any type.
- The return value of some_untyped_module.process(data) is not type-checked.
- This allows flexibility but removes the benefits of static type checking.

Creating Manual Type Hints for an Untyped Module

If you know the expected return types and parameter types of the functions in an untyped module, you can manually declare them using **stub type hints**.

Example: Adding Type Hints to an Unannotated Function

```
import some_untyped_module


def fetch_data(url: str) -> dict[str, Any]:  # Assume it returns a dictionary
    return some_untyped_module.get_data(url)
```

- Even though get_data() is untyped, we assume it returns dict[str, Any].
- This allows type checkers to verify **subsequent operations** on the returned data.

Using cast() to Force a Type on an Untyped Return Value

The cast() function from typing allows you to **explicitly define a type** for a value returned from an untyped function.

Example: Using cast() for Type Assurance

```
from typing import cast, Dict, Any

import some_untyped_module


def get_config() -> Dict[str, Any]:

   raw_config = some_untyped_module.load_config()

   return cast(Dict[str, Any], raw_config)
```

- cast(Dict[str, Any], raw_config) tells type checkers that raw_config should be treated as a dict[str, Any].
- This provides **type safety downstream** while allowing flexibility with untyped code.

Using # type: ignore to Suppress Type Checking

If a module lacks type hints and you want to **disable type errors**, you can use # type: ignore on the import or function call.

Example: Ignoring Type Checking for an Import

```
import some_untyped_module  # type: ignore
```

- Prevents mypy or static type checkers from raising errors about missing type hints.

Example: Ignoring Type Checking for a Function Call

```
def fetch_data():

    return some_untyped_module.get_data()  # type: ignore
```

- This avoids **false positives** when the function works correctly but lacks type hints.

Writing a Stub File (.pyi) for an Untyped Module

For **more structured type hinting**, you can create a **stub file (.pyi)** that defines the expected types for an untyped module.

Steps to Create a Stub File

1. Create a new file **some_untyped_module.pyi** (same name as the module, with a .pyi extension).
2. Define the module's functions and types.
3. Save the file in the same directory as the Python script or inside a typings/ directory.

Example Stub File (some_untyped_module.pyi)

```
# some_untyped_module.pyi (Type Stub File)
```

```
from typing import Dict, Any


def get_data(url: str) -> Dict[str, Any]: ...
def load_config() -> Dict[str, Any]: ...
```

- Now, when mypy or an IDE checks the module, it will recognize the expected types.

Using Third-Party Type Stubs (types- Packages)

Some popular third-party libraries provide **external type hints** through packages prefixed with types-.

Example: Installing Type Stubs for Requests

```
pip install types-requests
```

- This installs type hints for the requests module, allowing static type checking.

After installation, **IDE autocomplete and type checking tools** will recognize the types.

Summary

Approach	Use Case
Any	When the return type is unknown or dynamic.
Manual Type Hinting	When you know the expected types of function parameters and return values.
cast()	When you want to force a known type on an untyped return value.
# type: ignore	When you want to suppress type errors for specific lines.
Stub Files (.pyi)	When defining explicit types for an untyped module.
Installing Type Stubs	When external type hints are available via types- packages.

By applying these strategies, you can effectively **integrate type hints with untyped modules** while maintaining **code clarity and static analysis benefits**.

Best Practices and Performance Considerations

Type hints improve **code clarity, maintainability, and error detection**, but they should be used wisely. Overusing them can reduce readability and introduce unnecessary complexity. This section discusses when to use type hints, how to balance type safety with code readability, and best practices for keeping type annotations simple.

When to Use Type Hints and When to Avoid Them

Type hints are beneficial in **most cases**, but there are scenarios where they may not be necessary.

When to Use Type Hints

✅ **For function and method signatures** – Clearly defining expected inputs and return types improves documentation and reduces bugs.
✅ **For large codebases** – Helps multiple developers understand function behavior and catch type-related errors early.
✅ **For public APIs and libraries** – Ensures API consumers know the expected types and can use tools like mypy for validation.
✅ **For complex data structures** – Improves readability when working with nested dictionaries, lists, or tuples.

When to Avoid Type Hints

❌ **For simple, short functions** – If a function's behavior is obvious, adding type hints may be unnecessary.
❌ **For quick scripts and prototypes** – When working on small scripts or exploratory code, type hints may slow down development.
❌ **For dynamically typed data** – If a function genuinely needs to handle multiple types flexibly, using Any or omitting type hints might be more practical.

Example of Overuse (Unnecessary in Simple Code):

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

Since name is obviously a string, the type hints add little value in a **small, self-contained function**.

Balancing Type Safety and Code Readability

Type hints should **enhance** code readability, not make it more complex. Excessive type annotations can make code harder to read and maintain.

Best Practices for Readability

✓ **Use meaningful variable and function names** – Sometimes, a good name is enough to convey type information.
✓ **Use Optional instead of Union[T, None]** – Optional[str] is more readable than Union[str, None].
✓ **Keep type hints simple** – Avoid excessive nesting of complex types.

Example: Good vs. Bad Type Hinting for Readability

✅ Readable

```
from typing import Optional


def find_user(user_id: int) -> Optional[str]:
    return "Alice" if user_id == 1 else None
```

❌ Complex and unreadable

```
from typing import Union
```

```
def find_user(user_id: Union[int, float, str]) -> Union[str, None]:
    return "Alice" if user_id == 1 else None
```

- The first version is **clear and easy to understand**.
- The second version is **too complex** and makes it harder to determine valid input types.

Avoiding Excessive Type Complexity

Some data structures and functions require complex type hints, but **too much complexity reduces clarity**.

Tips to Avoid Complexity

✓ **Use type aliases for complex types** – Improves readability by breaking down long type hints.
✓ **Prefer TypedDict for structured dictionaries** – Makes dictionary keys and values easier to manage.
✓ **Break down large functions** – If type hints get too long, it may indicate that the function needs refactoring.

Example: Using a Type Alias to Simplify a Complex Type

```
from typing import Dict, List, TypeAlias


UserData: TypeAlias = Dict[str, List[int]]  # Type alias for
readability


def process_scores(data: UserData) -> None:
    for user, scores in data.items():
        print(f"{user}: {sum(scores) / len(scores) if scores else
0}")
```

- Instead of using Dict[str, List[int]] repeatedly, we define UserData, making the function signature **easier to read**.

Summary

Best Practice	Why It Matters
Use type hints for function signatures	Improves documentation and maintainability.
Avoid type hints in trivial cases	Keeps code concise and readable.
Keep type hints simple	Reduces complexity and makes code easier to understand.
Use type aliases for complex types	Improves readability when working with long or nested types.

Leverage Optional and TypedDict	Provides clarity for optional values and structured dictionaries.

Type hints **should aid readability, not complicate it**. Striking the right balance ensures Python remains **both expressive and type-safe** while maintaining its dynamic flexibility.

Summary

Type hints in Python provide a way to annotate variables, function parameters, and return values with expected data types. Introduced in PEP 484 with Python 3.5, they improve code readability, enhance documentation, and enable static type checking without enforcing type safety at runtime. By making expected types explicit, type hints help developers catch potential errors early through tools like mypy, pyright, and pylance. Despite their presence, Python remains dynamically typed, meaning type hints do not affect execution or runtime behavior.

Using type hints in function signatures allows developers to specify expected input and output types, making it easier to understand the intended use of functions. Built-in types such as integers, strings, lists, tuples, dictionaries, and sets can all be annotated explicitly. For more flexibility, the typing module provides features like Optional for handling None values and Union for specifying multiple possible types. When working with object-oriented programming, user-defined classes can serve as type hints for function parameters and return values.

Python's typing module introduces advanced type hinting features such as TypeVar, which enables generic programming by defining reusable and type-safe functions and classes. Protocols and structural subtyping provide an alternative to inheritance, allowing objects to be recognized based on behavior rather than their explicit class hierarchy. Special type hints such as Literal restrict values to predefined choices, while Final prevents attribute reassignment or subclassing.

Static type checking tools like mypy help enforce type correctness before runtime. When working with untyped modules or third-party libraries, strategies such as using Any, type casting, stub files, or external type stubs ensure compatibility with static analysis. While type hints improve maintainability and reduce errors, excessive complexity should be avoided to maintain readability. Overusing type hints in trivial cases can clutter code, whereas properly applied annotations make large-scale projects more structured and easier to maintain.

Python's type hints strike a balance between flexibility and type safety. They enhance documentation, enable better tooling support, and simplify debugging while preserving Python's dynamic nature. By following best practices and using type hints selectively, developers can improve code clarity without compromising readability.

Exercises

For each exercise, implement the required code, add a few lines in the main function to test it, and use mypy to perform static type checking.

1. Basic Type Annotations

Define a function square that takes an integer as input and returns an integer representing its square. Add appropriate type hints.

2. Type Hints for Built-in Collections

Create a function sum_list that takes a list of integers and returns the sum of the numbers in the list. Use type hints to specify the function signature.

3. Type Hints for Optional Values

Write a function get_username that takes an optional integer user_id. If user_id is None, return "Guest", otherwise return "User" + str(user_id). Use Optional from the typing module for type hinting.

4. Using Union for Multiple Types

Define a function format_value that accepts either an integer or a string and returns a formatted string. Use Union from the typing module.

5. Annotating Methods in a Class

Create a class Car with attributes brand (string) and speed (integer). Add a method accelerate that increases the speed and a method brake that sets the speed to zero. Use type hints for attributes and methods.

6. Working with Callable

Write a function apply_operation that takes two integers and a function as arguments. The function argument should accept two integers and return an integer. Use Callable for type hinting.

7. Using Type Hints for Custom Classes

Create a class Rectangle with attributes width and height. Define a function calculate_area that takes a Rectangle instance as input and returns its area. Use type hints for parameters and return values.

8. Implementing Generics with TypeVar

Define a generic function get_first_element that takes a list of any type and returns the first element. Use TypeVar to ensure type consistency.

9. Defining a Protocol for Structural Subtyping

Create a protocol Writer that requires a method write(self, data: str) -> None. Implement two classes, FileWriter and ConsoleWriter, that conform to this protocol. Define a function save_data that accepts a Writer instance and writes data.

10. Handling Untyped Modules with Type Hints

Imagine you are working with a third-party module that lacks type hints. Define a function fetch_data that calls an untyped function external_api_call(). Use cast from typing to enforce that the return value is a dictionary of strings to integers.