Chapter 5. Type Annotations

Annotating your Python code with type information is an optional step which can be very helpful during development and maintenance of a large project or a library. Static type checkers and lint tools help identify and locate data type mismatches in function arguments and return values. IDEs can use these *type annotations* (also called *type hints*) to improve autocompletion and to provide pop-up documentation. Third-party packages and frameworks can use type annotations to tailor runtime behavior, or to autogenerate code based on type annotations for methods and variables.

Type annotations and checking in Python continue to evolve, and touch on many complicated issues. This chapter covers some of the most common use cases for type annotations; you can find more comprehensive material in the resources listed at the end of the chapter.

TYPE ANNOTATION SUPPORT VARIES BY PYTHON VERSION

Python's features supporting type annotations have evolved from version to version, with some significant additions and deletions. The rest of this chapter will describe the type annotation support in the most recent versions of Python (3.10 and later), with notes to indicate features that might be present or absent in other versions.

History

Python is, fundamentally, a *dynamically typed* language. This lets you rapidly develop code by naming and using variables without having to declare them. Dynamic typing allows for flexible coding idioms, generic containers, and polymorphic data handling without requiring explicit

definition of interface types or class hierarchies. The downside is that the language offers no help during development in flagging variables of incompatible types being passed to or returned from functions. In place of the development-time compile step that some languages utilize to detect and report data type issues, Python relies on developers to maintain comprehensive unit tests, especially (though far from exclusively![1]) to uncover data type errors by re-creating the runtime environment in a series of test cases.

TYPE ANNOTATIONS ARE NOT ENFORCED

Type annotations are *not* enforced at runtime. Python does not perform any type validation or data conversion based on them; the executable Python code is still responsible for using variables and function arguments properly. However, type annotations must be syntactically correct. A late-imported or dynamically imported module containing an invalid type annotation raises a `SyntaxError` exception in your running Python program, just like any invalid Python statement.

Historically, the absence of any kind of type checking was often seen as a shortcoming of Python, with some programmers citing this as a reason for choosing other programming languages. However, the community wanted Python to maintain its runtime type freedom, so the logical approach was to add support for static type checks performed at development time by lint-like tools (described further in the following section) and IDEs. Some attempts were made at type checking based on parsing function signatures or docstrings. Guido van Rossum cited several cases on the **Python Developers mailing list** showing that type annotations could be helpful; for example, when maintaining large legacy codebases. With an annotation syntax, development tools could perform static type checks to highlight variable and function usages that conflict with the intended types.

The first official version of type annotations used specially formatted comments to indicate variable types and return codes, as defined in **PEP 484**, a provisional PEP for Python 3.5.[2] Using comments allowed for rapid implementation of, and experimentation with, the new typing syntax, without having to modify the Python compiler itself.[3] The third-party

package **mypy** gained broad acceptance performing static type checking using these comments. With the adoption of **PEP 526** in Python 3.6, type annotations were fully incorporated into the Python language itself, with a supporting `typing` module added to the standard library.

Type-Checking Utilities

As type annotations have become an established part of Python, type-checking utilities and IDE plug-ins have also become part of the Python ecosystem.

mypy

The standalone **mypy** utility continues to be a mainstay for static type checking, always up-to-date (give or take a Python version!) with evolving Python type annotation forms. `mypy` is also available as a plug-in for editors including Vim, Emacs, and SublimeText, and for the Atom, PyCharm, and VS Code IDEs. (PyCharm, VS Code, and Wing IDE also incorporate their own type-checking features separate from `mypy`.) The most common command for running `mypy` is simply **`mypy my_python_script.py`**.

You can find more detailed usage examples and command-line options in the **mypy online documentation**, as well as a **cheat sheet** that serves as a handy reference. Code examples later in this section will include example `mypy` error messages to illustrate the kinds of Python errors that can be caught using type checking.

Other Type Checkers

Other type checkers to consider using include:

MonkeyType
: Instagram's **MonkeyType** uses the `sys.setprofile` hook to detect types dynamically at runtime; like `pytype` (see below), it can also generate a *.pyi* (stub) file instead of, or in addition to, inserting type annotations in the Python code file itself.

`pydantic`

`pydantic` also works at runtime, but it does not generate stubs or insert type annotations; rather, its primary goal is to parse inputs and ensure that Python code receives clean data. As described in the **online docs**, it also allows you to extend its validation features for your own environment. See **"FastAPI"** for a simple example.

Pylance

Pylance is a type checking module primarily meant to embed Pyright (see below) into VS Code.

Pyre

Facebook's **Pyre** can also generate *.pyi* files. It currently does not run on Windows, unless you have the **Windows Subsystem for Linux (WSL)** installed.

Pyright

Pyright is Microsoft's static type checking tool, available as a command-line utility and a VS Code extension.

`pytype`

`pytype` from Google is a static type checker that focuses on *type inferencing* (and offers advice even in the absence of type hints) in addition to type annotations. Type inferencing offers a powerful capability for detecting type errors even in code without annotations. `pytype` can also generate *.pyi* files and merge stub files back into *.py* sources (the most recent versions of `mypy` are following suit on this). Currently, `pytype` does not run on Windows unless you first install **WSL**.

The emergence of type-checking applications from multiple major software organizations is a testimonial to the widespread interest in the Python developer community in using type annotations.

Type Annotation Syntax

A *type annotation* is specified in Python using the form:

```
identifier: type_specification
```

type_specification can be any Python expression, but usually involves one or more built-in types (for example, just mentioning a Python type is

a perfectly valid expression) and/or attributes imported from the `typing` module (discussed in the following section). The typical form is:

```
type_specifier[type_parameter, ...]
```

Here are some examples of type expressions used as type annotations for a variable:

```
import typing

# an int
count: int

# a list of ints, with a default value
counts: list[int] = []

# a dict with str keys, values are tuples containing 2 ints and a str
employee_data: dict[str, tuple[int, int, str]]

# a callable taking a single str or bytes argument and returning a bool
str_predicate_function: typing.Callable[[str | bytes], bool]

# a dict with str keys, whose values are functions that take and return
# an int
str_function_map: dict[str, typing.Callable[[int], int]] = {
    'square': lambda x: x * x,
    'cube': lambda x: x * x * x,
}
```

Note that **`lambda`**s do *not* accept type annotations.

TYPING SYNTAX CHANGES IN PYTHON 3.9 AND 3.10

One of the most significant changes in type annotations during the span of Python versions covered in this book was the support added in Python 3.9 for using built-in Python types, as shown in these examples.

-3.9 Prior to Python 3.9, these annotations required the use of type names imported from the `typing` module, such as `Dict`, `List`, `Tuple`, etc.

3.10+ Python 3.10 added support for using | to indicate alternative types, as a more readable, concise alternative to the `Union[`*`atype`*`,` *`btype`*`, ...]` notation. The | operator can also be used to replace `Optional[`*`atype`*`]` with *`atype`* | `None`.

For instance, the previous `str_predicate_function` definition would take one of the following forms, depending on your version of Python:

```
# prior to 3.10, specifying alternative types
# requires use of the Union type
from typing import Callable, Union
str_predicate_function: Callable[Union[str, bytes], bool]

# prior to 3.9, built-ins such as list, tuple, dict,
# set, etc. required types imported from the typing
# module
from typing import Dict, Tuple, Callable, Union
employee_data: Dict[str, Tuple[int, int, str]]
str_predicate_function: Callable[Union[str, bytes], bool]
```

To annotate a function with a return type, use the form:

```
def identifier(argument, ...) -> type_specification :
```

where each *argument* takes the form:

```
identifier[: type_specification[ = default_value]]
```

Here's an example of an annotated function:

```
def pad(a: list[str], min_len: int = 1, padstr: str = ' ') -> list[str]:
    """Given a list of strings and a minimum length, return a copy of
       the list extended with "padding" strings to be at least the
       minimum length.
    """
    return a + ([padstr] * (min_len - len(a)))
```

Note that when an annotated parameter has a default value, PEP 8 recommends using spaces around the equals sign.

FORWARD-REFERENCING TYPES THAT ARE NOT YET FULLY DEFINED

At times, a function or variable definition needs to reference a type that has not yet been defined. This is quite common in class methods, or methods that must define arguments or return values of the type of the current class. Those function signatures are parsed at compile time, and at that point the type is not yet defined. For example, this `classmethod` fails to compile:

```
class A:
    @classmethod
    def factory_method(cls) -> A:
        # ... method body goes here ...
```

Since class `A` has not yet been defined when Python compiles `factory_method`, the code raises `NameError`.

The problem can be resolved by enclosing the return type `A` in quotes:

```
class A:
    @classmethod
    def factory_method(cls) -> 'A':
        # ... method body goes here ...
```

A future version of Python may defer the evaluation of type annotations until runtime, making the enclosing quotes unnecessary (Python's Steering Committee is evaluating various possibilities). You can preview this behavior using `from __future__ import annotations`.

The typing Module

The `typing` module supports type hints. It contains definitions that are useful when creating type annotations, including:

- Classes and functions for defining types
- Classes and functions for modifying type expressions
- Abstract base classes (ABCs)

- Protocols
- Utilities and decorators
- Classes for defining custom types

Types

The initial implementations of the `typing` module included definitions of types corresponding to Python built-in containers and other types, as well as types from standard library modules. Many of these types have since been deprecated (see below), but some are still useful, since they do not correspond directly to any Python built-in type. **Table 5-1** lists the `typing` types still useful in Python 3.9 and later.

Table 5-1. Useful definitions in the `typing` module

Type	Description
`Any`	Matches any type.
`AnyStr`	Equivalent to `str | bytes`. `AnyStr` is meant to be used to annotate function arguments and return types where either string type is acceptable, but the types should not be mixed between multiple arguments, or arguments and return types.
`BinaryIO`	Matches streams with binary (`bytes`) content such as those returned from `open` with `mode='b'`, or `io.BytesIO`.
`Callable`	`Callable[[`*`argument_type`*`, ...], `*`return_type`*`]` Defines the type signature for a callable object. Takes a list of types corresponding to the arguments to the callable, and a type for the return value of the function. If the callable takes no arguments, indicate this with an

Type	Description
	empty list, `[]`. If the callable has no return value, use **`None`** for *`return_type`*.
`IO`	Equivalent to `BinaryIO	TextIO`.
`Literal [expression,...]`	**3.8+** Specifies a list of valid values that the variable may take.
`LiteralString`	**3.11+** Specifies a `str` that must be implemented as a literal quoted value. Used to guard against leaving code open to injection attacks.
`NoReturn`	Use as the return type for functions that "run forever," such as those that call `http.serve_forever` or `event_loop.run_forever` without returning. This is *not* intended for functions that simply return with no explicit value; for those use `->` **`None`**. More discussion of return types can be found in **"Adding Type Annotations to Existing Code (Gradual Typing)"**.
`Self`	**3.11+** Use as the return type for instance functions that **`return`** `self` (and in a few other cases, as exemplified in **PEP 673**).
`TextIO`	Matches streams with text (`str`) content, such as those returned from `open` with `mode='t'`, or `io.StringIO`.

-3.9 Prior to 3.9, the definitions in the `typing` module were used to create types representing built-in types, such as `List[int]` for a list of `int`s. From 3.9 onward, these names are deprecated, as their corresponding

built-in or standard library types now support the `[]` syntax: a list of `int`s is now simply typed using `list[int]`. **Table 5-2** lists the definitions from the `typing` module that were necessary prior to Python 3.9 for type annotations using built-in types.

Table 5-2. Python built-in types and their pre-3.9 definitions in the `typing` module

Built-in type	Pre-3.9 typing module equivalent
`dict`	`Dict`
`frozenset`	`FrozenSet`
`list`	`List`
`set`	`Set`
`str`	`Text`
`tuple`	`Tuple`
`type`	`Type`
`collections.ChainMap`	`ChainMap`
`collections.Counter`	`Counter`
`collections.defaultdict`	`DefaultDict`
`collections.deque`	`Deque`
`collections.OrderedDict`	`OrderedDict`
`re.Match`	`Match`
`re.Pattern`	`Pattern`

Type Expression Parameters

Some types defined in the `typing` module modify other type expressions. The types listed in **Table 5-3** provide additional typing information or constraints for the modified types in *`type_expression`*.

Table 5-3. Type expression parameters

Parameter	Usage and description
`Annotated`	`Annotated[`*`type_expression, expression, ...`*`]` **3.9+** Extends the *`type_expression`* with additional metadata. The extra metadata values for function *`fn`* can be retrieved at runtime using `get_type_hints(`*`fn`*`, include_extras=True)`.
`ClassVar`	`ClassVar[`*`type_expression`*`]` Indicates that the variable is a class variable, and should not be assigned as an instance variable.
`Final`	`Final[`*`type_expression`*`]` **3.8+** Indicates that the variable should not be written to or overridden in a subclass.
`Optional`	`Optional[`*`type_expression`*`]` Equivalent to *`type_expression`* `|` **`None`**. Often used for named arguments with a default value of **`None`**. (`Optional` does not automatically define **`None`** as the default value, so you must still follow it with `=`**`None`** in a function signature.) **3.10+** With the availability of the `|` operator for specifying alternative type attributes, there is a growing consensus to prefer *`type_expression`* `|` **`None`** over using `Optional[`*`type_expression`*`]`.

Abstract Base Classes

Just as for built-in types, the initial implementations of the `typing` module included definitions of types corresponding to abstract base classes in the `collections.abc` module. Many of these types have since been deprecated (see below), but two definitions have been retained as aliases to ABCs in `collections.abc` (see **Table 5-4**).

Table 5-4. Abstract base class aliases

Type	Method subclasses must implement
`Hashable`	`__hash__`
`Sized`	`__len__`

-3.9 Prior to Python 3.9, the following definitions in the `typing` module represented abstract base classes defined in the `collections.abc` module, such as `Sequence[int]` for a sequence of `int`s. From 3.9 onward, these names in the `typing` module are deprecated, as their corresponding types in `collections.abc` now support the `[]` syntax:

`AbstractSet`	`Container`	`Mapping`
`AsyncContextManager`	`ContextManager`	`MappingView`
`AsyncGenerator`	`Coroutine`	`MutableMapping`
`AsyncIterable`	`Generator`	`MutableSequence`
`AsyncIterator`	`ItemsView`	`MutableSet`
`Awaitable`	`Iterable`	`Reversible`
`ByteString`	`Iterator`	`Sequence`

Collection	KeysView	ValuesView

Protocols

The `typing` module defines several *protocols*, which are similar to what some other languages call "interfaces." Protocols are abstract base classes intended to concisely express constraints on a type, ensuring it contains certain methods. Each protocol currently defined in the `typing` module relates to a single special method, and its name starts with `Supports` followed by the name of the method (however, other libraries, such as those defined in **typeshed**, need not follow the same constraints). Protocols can be used as minimal abstract classes to determine a class's support for that protocol's capabilities: all that a class needs to do to comply with a protocol is to implement the protocol's special method(s).

Table 5-5 lists the protocols defined in the `typing` module.

Table 5-5. Protocols in the `typing` module and their required methods

Protocol	Has method
SupportsAbs	__abs__
SupportsBytes	__bytes__
SupportsComplex	__complex__
SupportsFloat	__float__
SupportsIndex 3.8+	__index__
SupportsInt	__int__
SupportsRound	__round__

A class does not have to explicitly inherit from a protocol in order to satisfy `issubclass(cls, protocol_type)`, or for its instances to satisfy `isinstance(obj, protocol_type)`. The class simply has to implement the method(s) defined in the protocol. Imagine, for example, a class implementing Roman numerals:

```
class RomanNumeral:
    """Class representing some Roman numerals and their int
       values.
    """
    int_values = {'I': 1, 'II': 2, 'III': 3, 'IV': 4, 'V': 5}

    def __init__(self, label: str):
        self.label = label

    def __int__(self) -> int:
        return RomanNumeral.int_values[self.label]
```

To create an instance of this class (to, say, represent a sequel in a movie title) and get its value, you could use the following code:

```
>>> movie_sequel = RomanNumeral('II')
>>> print(int(movie_sequel))
```

```
2
```

`RomanNumeral` satisfies `issubclass`, and `isinstance` checks with `SupportsInt` because it implements `__int__`, even though it does not inherit explicitly from the protocol class `SupportsInt`:[4]

```
>>> issubclass(RomanNumeral, typing.SupportsInt)
```

```
True
```

```
>>> isinstance(movie_sequel, typing.SupportsInt)
```

```
True
```

Utilities and Decorators

Table 5-6 lists commonly used functions and decorators defined in the `typing` module; it's followed by a few examples.

Table 5-6. Commonly used functions and decorators defined in the `typing` module

Function/decorator	Usage and description
`cast`	`cast(type, var)` Signals to the static type checker that *var* should be considered as type *type*. Returns *var*; at runtime there is no change, conversion, or validation of *var*. See the example after the table.
`final`	`@final` 3.8+ Used to decorate a method in a class definition, to warn if the method is overridden in a subclass. Can also be used as a class decorator, to warn if the class itself is being subclassed.
`get_args`	`get_args(custom_type)` Returns the arguments used to construct a custom type.

Function/decorator	Usage and description
`get_origin`	`get_origin(`*`custom_type`*`)` 3.8+ Returns the base type used to construct a custom type.
`get_type_hints`	`get_type_hints(`*`obj`*`)` Returns results as if accessing *`obj`*`.__annotations__`. Can be called with optional `globalns` and `localns` namespace arguments to resolve forward type references given as strings, and/or with optional Boolean `include_extras` argument to include any nontyping annotations added using `Annotations`.
`NewType`	`NewType(`*`type_name`*`, `*`type`*`)` Defines a custom type derived from *`type`*. *`type_name`* is a string that should match the local variable to which the `NewType` is being assigned. Useful for distinguishing different uses for common types, such as a `str` used for an employee name versus a `str` used for a department name. See **"NewType"** for more on this function.
`no_type_check`	`@no_type_check` Used to indicate that annotations are not intended to be used as type information. Can be applied to a class or function.
`no_type_check_decorator`	`@no_type_check_decorator` Used to add `no_type_check` behavior to another decorator.

Function/decorator	Usage and description
`overload`	`@overload` Used to allow defining multiple methods with the same name but differing types in their signatures. See the example after the table.
`runtime_checkable`	`@runtime_checkable` 3.8+ Used to add `isinstance` and `issubclass` support for custom protocol classes. See **"Using Type Annotations at Runtime"** for more on this decorator.
`TypeAlias`	*`name`*`: TypeAlias = `*`type_expression`* 3.10+ Used to distinguish the definition of a type alias from a simple assignment. Most useful in cases where *`type_expression`* is a simple class name or a string value referring to a class that is not yet defined, which might look like an assignment. `TypeAlias` may only be used at module scope. A common use is to make it easier to consistently reuse a lengthy type expression, e.g.: `Number: TypeAlias = int | float | Fraction`. See **"TypeAlias"** for more on this annotation.
`type_check_only`	`@type_check_only` Used to indicate that the class or function is only used at type-checking time and is not available at runtime.
`TYPE_CHECKING`	A special constant that static type checkers evaluate as **`True`** but that is set to **`False`** at runtime. Use this to skip imports of large, slow-to-import modules used solely to

Function/decorator	Usage and description
	support type checking (so that the import is not needed at runtime).
`TypeVar`	`TypeVar(`*`type_name`*`, *`*`types`*`)` Defines a type expression element for use in complex generic types using `Generic`. *`type_name`* is a string that should match the local variable to which the `TypeVar` is being assigned. If *`types`* are not given, then the associated `Generic` will accept any type. If *`types`* are given, then the `Generic` will only accept instances of any of the provided types or their subclasses. Also accepts the named Boolean arguments `covariant` and `contravariant` (both defaulting to `False`), and the argument `bound`. These are described in more detail in **"Generics and TypeVars"** and in the **typing module docs**.

Use `overload` at type-checking time to flag named arguments that must be used in particular combinations. In this case, `fn` must be called with either a `str` key and `int` value pair, or with a single `bool` value:

```
@typing.overload
def fn(*, key: str, value: int):
    ...

@typing.overload
def fn(*, strict: bool):
    ...

def fn(**kwargs):
    # implementation goes here, including handling of differing
    # named arguments
    pass
```

```
# valid calls
fn(key='abc', value=100)
fn(strict=True)

# invalid calls
fn(1)
fn('abc')
fn('abc', 100)
fn(key='abc')
fn(True)
fn(strict=True, value=100)
```

Note that the `overload` decorator is used purely for static type checking. To actually dispatch to different methods based on a parameter type at runtime, use `functools.singledispatch`.

Use the `cast` function to force a type checker to treat a variable as being of a particular type, within the scope of the cast:

```
def func(x: list[int] | list[str]):
    try:
        return sum(x)
    except TypeError:
        x = cast(list[str], x)
        return ','.join(x)
```

USE CAST WITH CAUTION

`cast` is a way of overriding any inferences or prior annotations that may be present at a particular place in your code. It may hide actual type errors in your code, rendering the type-checking pass incomplete or inaccurate. The `func` in the preceding example raises no `mypy` warnings itself, but fails at runtime if passed a list of mixed `int`s and `str`s.

Defining Custom Types

Just as Python's **`class`** syntax permits the creation of new runtime types and behavior, the `typing` module constructs discussed in this section enable the creation of specialized type expressions for advanced type checking.

The `typing` module includes three classes from which your classes can inherit to get type definitions and other default features, listed in **Table 5-7**.

Table 5-7. Base classes for defining custom types

`Generic`	`Generic[`*`type_var`*`, ...]` Defines a type-checking abstract base class for a class whose methods reference one or more `TypeVar`-defined types. Generics are described in more detail in the following subsection.
`NamedTuple`	`NamedTuple` A typed implementation of `collections.namedtuple`. See **"NamedTuple"** for further details and examples.
`TypedDict`	`TypedDict` **3.8+** Defines a type-checking `dict` that has specific keys and value types for each key. See **"TypedDict"** for details.

Generics and TypeVars

Generics are types that define a template for classes that can adapt the type annotations of their method signatures based on one or more type parameters. For instance, `dict` is a generic that takes two type parameters: the type for the dictionary keys and the type for the dictionary values. Here is how `dict` might be used to define a dictionary that maps color names to RGB triples:

```
color_lookup: dict[str, tuple[int, int, int]] = {}
```

The variable `color_lookup` will support statements like:

```
color_lookup['red'] = (255, 0, 0)
color_lookup['red'][2]
```

However, the following statements generate `mypy` errors, due to a mismatched key or value type:

```
color_lookup[0]
```

```
error: Invalid index type "int" for "dict[str, tuple[int, int, int]]";
expected type "str"
```

```
color_lookup['red'] = (255, 0, 0, 0)
```

```
error: Incompatible types in assignment (expression has type
"tuple[int, int, int, int]", target has type "tuple[int, int, int]")
```

Generic typing permits the definition of behavior in a class that is independent of the specific types of the objects that class works with. Generics are often used for defining container types, such as `dict`, `list`, `set`, etc. By defining a generic type, we avoid the necessity of exhaustively defining types for `DictOfStrInt`, `DictOfIntEmployee`, and so on. Instead, a generic `dict` is defined as `dict[`*`KT`*`, `*`VT`*`]`, where *`KT`* and *`VT`* are placeholders for the `dict`'s key type and value type, and the specific types for any particular `dict` can be defined when the `dict` is instantiated.

As an example, let's define a hypothetical generic class: an accumulator that can be updated with values, but which also supports an `undo` method. Since the accumulator is a generic container, we declare a `TypeVar` to represent the type of the contained objects:

```
import typing
T = typing.TypeVar('T')
```

The `Accumulator` class is defined as a subclass of `Generic`, with `T` as a type parameter. Here is the class declaration and its `__init__` method, which creates a contained list, initially empty, of objects of type `T`:

```
class Accumulator(typing.Generic[T]):
    def __init__(self):
        self._contents: list[T] = []
```

To add the `update` and `undo` methods, we define arguments that reference the contained objects as being of type `T`:

```
    def update(self, *args: T) -> None:
        self._contents.extend(args)

    def undo(self) -> None:
        # remove last value added
        if self._contents:
            self._contents.pop()
```

Lastly, we add `__len__` and `__iter__` methods so that `Accumulator` instances can be iterated over:

```
    def __len__(self) -> int:
        return len(self._contents)
```

```
    def __iter__(self) -> typing.Iterator[T]:
        return iter(self._contents)
```

Now this class can be used to write code using `Accumulator[int]` to collect a number of `int` values:

```
acc: Accumulator[int] = Accumulator()
acc.update(1, 2, 3)
print(sum(acc))  # prints 6
acc.undo()
print(sum(acc))  # prints 3
```

Because `acc` is an `Accumulator` containing `int`s, the following statements generate `mypy` error messages:

```
acc.update('A')
```

```
error: Argument 1 to "update" of "Accumulator" has incompatible type
"str"; expected "int"
```

```
print(''.join(acc))
```

```
error: Argument 1 to "join" of "str" has incompatible type
"Accumulator[int]"; expected "Iterable[str]"
```

Restricting TypeVar to specific types

Nowhere in our `Accumulator` class do we ever invoke methods directly on the contained `T` objects themselves. For this example, the `T TypeVar` is purely untyped, so type checkers like `mypy` cannot infer the presence of

any attributes or methods of the `T` objects. If the generic needs to access attributes of the `T` objects it contains, then `T` should be defined using a modified form of `TypeVar`.

Here are some examples of `TypeVar` definitions:

```
# T must be one of the types listed (int, float, complex, or str)
T = typing.TypeVar('T', int, float, complex, str)
# T must be the class MyClass or a subclass of the class MyClass
T = typing.TypeVar('T', bound=MyClass)
# T must implement __len__ to be a valid subclass of the Sized protocol
T = typing.TypeVar('T', bound=collections.abc.Sized)
```

These forms of `T` allow a generic defined on `T` to use methods from these types in `T`'s `TypeVar` definition.

NamedTuple

The `collections.namedtuple` function simplifies the definition of class-like tuple types that support named access to the tuple elements. `NamedTuple` provides a typed version of this feature, using a class with attributes-style syntax similar to `dataclasses` (covered in **"Data Classes"**). Here's a `NamedTuple` with four elements, with names, types, and optional default values:

```
class HouseListingTuple(typing.NamedTuple):
    address: str
    list_price: int
    square_footage: int = 0
    condition: str = 'Good'
```

`NamedTuple` classes generate a default constructor, accepting positional or named arguments for each named field:

```
listing1 = HouseListingTuple(
    address='123 Main',
    list_price=100_000,
    square_footage=2400,
    condition='Good',
)

print(listing1.address)  # prints: 123 Main
print(type(listing1))    # prints: <class 'HouseListingTuple'>
```

Attempting to create a tuple with too few elements raises a runtime error:

```
listing2 = HouseListingTuple(
    '123 Main',
)
# raises a runtime error: TypeError: HouseListingTuple.__new__()
# missing 1 required positional argument: 'list_price'
```

TypedDict

3.8+ Python `dict` variables are often difficult to decipher in legacy codebases, because `dicts` are used in two ways: as collections of key/value pairs (such as a mapping from user ID to username), and records mapping known field names to values. It is usually easy to see that a function argument is to be passed as a `dict`, but the actual keys and value types are dependent on the code that may call that function. Beyond simply defining that a `dict` may be a mapping of `str` values to `int` values, as in `dict[str, int]`, a `TypedDict` defines the expected keys and the types of each corresponding value. The following example defines a `TypedDict` version of the previous house listing type (note that `TypedDict` definitions do not accept default value definitions):

```
class HouseListingDict(typing.TypedDict):
    address: str
    list_price: int
```

```
    square_footage: int
    condition: str
```

`TypedDict` classes generate a default constructor, accepting named arguments for each defined key:

```
listing1 = HouseListingDict(
    address='123 Main',
    list_price=100_000,
    square_footage=2400,
    condition='Good',
)

print(listing1['address'])  # prints 123 Main
print(type(listing1))  # prints <class 'dict'>

listing2 = HouseListingDict(
    address='124 Main',
    list_price=110_000,
)
```

Unlike the `NamedTuple` example, `listing2` will not raise a runtime error, simply creating a `dict` with just the given keys. However, `mypy` will flag `listing2` as a type error with the message:

```
error: Missing keys ("square_footage", "condition") for TypedDict
"HouseListing"
```

To indicate to the type checker that some keys may be omitted (but to still validate those that are given), add `total=False` to the class declaration:

```
class HouseListing(typing.TypedDict, total=False):
    # ...
```

3.11+ Individual fields can also use the `Required` or `NotRequired` type annotations to explicitly mark them as required or optional:

```
class HouseListing(typing.TypedDict):
    address: typing.Required[str]
    list_price: int
    square_footage: typing.NotRequired[int]
    condition: str
```

`TypedDict` can be used to define a generic type, too:

```
T = typing.TypeVar('T')

class Node(typing.TypedDict, typing.Generic[T]):
    label: T
    neighbors: list[T]

n = Node(label='Acme', neighbors=['anvil', 'magnet', 'bird seed'])
```

DO NOT USE THE LEGACY TYPEDDICT(NAME, **FIELDS) FORMAT

To support backporting to older versions of Python, the initial release of `TypedDict` also let you use a syntax similar to that for `namedtuple`, such as:

```
HouseListing = TypedDict('HouseListing',
                         address=str,
                         list_price=int,
                         square_footage=int,
                         condition=str)
```

or:

```
HouseListing = TypedDict('HouseListing',
                         {'address': str,
                          'list_price': int,
                          'square_footage': int,
                          'condition': str})
```

These forms are deprecated in Python 3.11, and are planned to be removed in Python 3.13.

Note that `TypedDict` does not actually define a new type. Classes created by inheriting from `TypedDict` actually serve as `dict` factories, such that instances created from them *are* `dicts`. Reusing the previous code snippet defining the `Node` class, we can see this using the `type` built-in function:

```
n = Node(label='Acme', neighbors=['anvil', 'magnet', 'bird seed'])
print(type(n))           # prints: <class 'dict'>
print(type(n) is dict)   # prints: True
```

There is no special runtime conversion or initialization when using `TypedDict`; the benefits of `TypedDict` are those of static type checking and self documentation, which naturally accrue from using type annotations.

WHICH SHOULD YOU USE, NAMEDTUPLE OR TYPEDDICT?

The two data types appear similar in terms of their supported features, but there are significant differences that should help you determine which one to use.

`NamedTuple`s are immutable, so they can be used as dictionary keys or stored in sets, and are inherently safe to share across threads. As a `NamedTuple` object is a tuple, you can get its property values in order simply by iterating over it. However, to get the attribute names, you need to use the special `__annotations__` attribute.

Since classes created with `TypedDict` are actually `dict` factories, instances created from them are `dict`s, with all the behavior and attributes of `dict`s. They are mutable, so their values can be updated without creating a new container instance, and they support all the `dict` methods, such as `keys`, `values`, and `items`. They are also easily serialized using JSON or `pickle`. However, being mutable, they cannot be used as keys in another `dict`, nor can they be stored in a `set`.

`TypedDict`s are more lenient than `NamedTuple`s about missing keys. When a key is omitted when constructing a `TypedDict`, there is no error (though you will get a type-check warning from the static type checker). On the other hand, if an attribute is omitted when constructing a `NamedTuple`, this will raise a runtime `TypeError`.

In short, there is no across-the-board rule for when to use a `NamedTuple` versus a `TypedDict`. Consider these alternative behaviors and how they relate to your program and its use of these data objects when deciding between a `NamedTuple` and a `TypedDict`—and don't forget the other, often preferable, alternative of using a `dataclass` (covered in **"Data Classes"**) instead!

TypeAlias

3.10+ Defining a simple type alias can be misinterpreted as assigning a class to a variable. For instance, here we define a type for record identifiers in a database:

```
Identifier = int
```

To clarify that this statement is intended to define a custom type name for the purposes of type checking, use `TypeAlias`:

```
Identifier: TypeAlias = int
```

`TypeAlias` is also useful when defining an alias for a type that is not yet defined, and so referenced as a string value:

```
# Python will treat this like a standard str assignment
TBDType = 'ClassNotDefinedYet'

# indicates that this is actually a forward reference to a class
TBDType: TypeAlias = 'ClassNotDefinedYet'
```

`TypeAlias` types may only be defined at module scope. Custom types defined using `TypeAlias` are interchangeable with the target type. Contrast `TypeAlias` (which does not create a new type, just gives a new name for an existing one) with `NewType`, covered in the following section, which does create a new type.

NewType

`NewType` allows you to define application-specific subtypes, to avoid confusion that might result from using the same type for different variables. If your program uses `str` values for different types of data, for example, it is easy to accidentally interchange values. Suppose you have a program that models employees in departments. The following type declaration is not sufficiently descriptive—which is the key and which is the value?

```
employee_department_map: dict[str, str] = {}
```

Defining types for employee and department IDs makes this declaration clearer:

```
EmpId = typing.NewType('EmpId', str)
DeptId = typing.NewType('DeptId', str)
employee_department_map: dict[EmpId, DeptId] = {}
```

These type definitions will also allow type checkers to flag this incorrect usage:

```
def transfer_employee(empid: EmpId, to_dept: DeptId):
    # update department for employee
     employee_department_map[to_dept] = empid
```

Running `mypy` reports these errors for the line `employee_department_map[to_dept] = empid`:

```
error: Invalid index type "DeptId" for "Dict[EmpId, DeptId]"; expected
type "EmpId"
error: Incompatible types in assignment (expression has type "EmpId",
target has type "DeptId")
```

Using `NewType` often requires you to use `typing.cast` too; for example, to create an `EmpId`, you need to cast a `str` to the `EmpId` type.

You can also use `NewType` to indicate the desired implementation type for an application-specific type. For instance, the basic US postal zip code is five numeric digits. It is common to see this implemented using `int`, which becomes problematic with zip codes that have a leading `0`. To indicate that zip codes should be implemented using `str`, your code can define this type-checking type:

```
ZipCode = typing.NewType("ZipCode", str)
```

Annotating variables and function arguments using `ZipCode` will help flag incorrect uses of `int` for zip code values.

Using Type Annotations at Runtime

Function and class variable annotations can be introspected by accessing the function or class's `__annotations__` attribute (although a **better practice** is to instead call `inspect.get_annotations()`):

```
>>> def f(a:list[str], b) -> int:
...     pass
...
>>> f.__annotations__
```

```
{'a': list[str], 'return': <class 'int'>}
```

```
>>> class Customer:
...     name: str
...     reward_points: int = 0
...
>>> Customer.__annotations__
```

```
{'name': <class 'str'>, 'reward_points': <class 'int'>}
```

This feature is used by third-party packages such as `pydantic` and FastAPI to provide extra code generation and validation capabilities.

3.8+ To define your own custom protocol class that supports runtime checking with `issubclass` and `isinstance`, define that class as a subclass of `typing.Protocol`, with empty method definitions for the required protocol methods, and decorate the class with `@runtime_checkable` (covered

in Table 5-6). If you *don't* decorate it with `@runtime_checkable`, you're still defining a `Protocol` that's quite usable for static type checking, but it won't be runtime-checkable with `issubclass` and `isinstance`.

For example, we could define a protocol that indicates that a class implements the `update` and `undo` methods as follows (the Python `Ellipsis`, `...`, is a convenient syntax for indicating an empty method definition):

```
T = typing.TypeVar('T')

@typing.runtime_checkable
class SupportsUpdateUndo(typing.Protocol):
    def update(self, *args: T) -> None:
        ...
    def undo(self) -> None:
        ...
```

Without making any changes to the inheritance path of `Accumulator` (defined in "Generics and TypeVars"), it now satisfies runtime type checks with `SupportsUpdateUndo`:

```
>>> issubclass(Accumulator, SupportsUpdateUndo)
```

```
True
```

```
>>> isinstance(acc, SupportsUpdateUndo)
```

```
True
```

In addition, any other class that implements `update` and `undo` methods will now qualify as a `SupportsUpdateUndo` "subclass."

How to Add Type Annotations to Your Code

Having seen some of the features and capabilities provided by using type annotations, you may be wondering about the best way to get started. This section describes a few scenarios and approaches to adding type annotations.

Adding Type Annotations to New Code

When you start writing a short Python script, adding type annotations may seem like an unnecessary extra burden. As a spinoff of the **Two Pizza Rule**, we suggest the Two Function Rule: as soon as your script contains two functions or methods, go back and add type annotations to the method signatures, and any shared variables or types as necessary. Use `TypedDict` to annotate any `dict` structures that are used in place of classes, so that `dict` keys get clearly defined up front or get documented as you go; use `NamedTuples` (or `dataclasses`: some of this book's authors *strongly* prefer the latter option) to define the specific attributes needed for those data "bundles."

If you are beginning a major project with many modules and classes, then you should definitely use type annotations from the beginning. They can easily make you more productive, as they help avoid common naming and typing mistakes and ensure you get more fully supported autocompletion while working in your IDE. This is even more important on projects with multiple developers: having documented types helps tell everyone on the team the expectations for types and values to be used across the project. Capturing these types in the code itself makes them immediately accessible and visible during development, much more so than separate documentation or specifications.

If you are developing a library to be shared across projects, then you should also use type annotations from the very start, most likely paralleling the function signatures in your API design. Having type annotations in a library will make life easier for your client developers, as all modern

IDEs include type annotation plug-ins to support static type checking and function autocompletion and documentation. They will also help you when writing your unit tests, since you will benefit from the same rich IDE support.

For any of these projects, add a type-checking utility to your pre-commit hooks, so that you stay ahead of any type infractions that might creep into your new codebase. This way you can fix them as they occur, instead of waiting until you do a large commit and finding that you have made some fundamental typing errors in multiple places.

Adding Type Annotations to Existing Code (Gradual Typing)

Several companies that have run projects to apply type annotations to large existing codebases recommend an incremental approach, referred to as *gradual typing*. With gradual typing, you can work through your codebase in a stepwise manner, adding and validating type annotations a few classes or modules at a time.

Some utilities, like `mypy`, will let you add type annotations function by function. `mypy`, by default, skips functions without typed signatures, so you can methodically go through your codebase a few functions at a time. This incremental process allows you to focus your efforts on individual parts of the code, as opposed to adding type annotations everywhere and then trying to sort out an avalanche of type-checker errors.

Some recommended approaches are:

- Identify your most heavily used modules, and begin adding types to them, a method at a time. (These could be core application class modules, or widely shared utility modules.)
- Annotate a few methods at a time, so that type-checking issues get raised and resolved gradually.
- Use `pytype` or `pyre` inference to generate initial *.pyi* stub files (discussed in the following section). Then, steadily migrate types from

the *.pyi* files, either manually or using automation such as `pytype`'s `merge_pyi` utility.

- Begin using type checkers in a lenient default mode, so that most code is skipped and you can focus attention on specific files. As work progresses, shift to a stricter mode so that remaining items are made more prominent, and files that have been annotated do not regress by taking on new nonannotated code.

Using .pyi Stub Files

Sometimes you don't have access to Python type annotations. For example, you might be using a library that does not have type annotations, or using a module whose functions are implemented in C.

In these cases, you can use separate *.pyi* stub files containing just the related type annotations. Several of the type checkers mentioned at the beginning of this chapter can generate these stub files. You can download stub files for popular Python libraries, as well as the Python standard library itself, from the **`typeshed` repository**. You can maintain stub files from the Python source, or, using merging utilities available in some of the type checkers, integrate them back into the original Python source.

DO TYPE ANNOTATIONS GET IN THE WAY OF CODING?

Type annotations carry some stigma, especially for those who have worked with Python for many years and are used to taking full advantage of Python's adaptive nature. Flexible method signatures like that of the built-in function `max`, which can take a single argument containing a sequence of values or multiple arguments containing the values to be maximized, have been cited as being **especially challenging to type-annotate**. Is this the fault of the code? Of typing? Of Python itself? Each of these explanations is possible.

In general, typing fosters a degree of formalism and discipline that can be more confining than the historical Python philosophy of "coding by and for consenting adults." Moving forward, we may find that the flexibility of style in older Python code is not wholly conducive to long-term use, reuse, and maintenance by those who are not the original code authors. As a recent PyCon presenter **suggested**, "Ugly type annotations hint at ugly code." (However, it may sometimes be the case, like for `max`, that it's the typing system that's not expressive enough.)

You can take the level of typing difficulty as an indicator of your method design. If your methods require multiple `Union` definitions, or multiple overrides for the same method using different argument types, perhaps your design is too flexible across multiple calling styles. You may be overdoing the flexibility of your API because Python allows it, but that might not always be a good idea in the long run. After all, as the **Zen of Python** says, "There should be one—and preferably only one—obvious way to do it." Maybe that should include "only one obvious way" to call your API!

Summary

Python has steadily risen to prominence as a powerful language and programming ecosystem, supporting important enterprise applications. What was once a utility language for scripting and task automation has become a platform for significant and complex applications affecting millions of users, used in mission-critical and even extraterrestrial systems.[5] Adding type annotations is a significant step in developing and maintaining these systems.

The **online documentation** for type annotations provides up-to-date descriptions, examples, and **best practices** as the syntax and practices for annotating types continue to evolve. The authors also recommend ***Fluent Python*, 2nd edition**, by Luciano Ramalho (O'Reilly), especially Chapters 8 and 15, which deal specifically with Python type annotations.

1 Strong, extensive unit tests will also guard against many business logic problems that no amount of type checking would ever catch for you—so, type hints are not to be used *instead of* unit tests, but *in addition* to them.

2 The *syntax* for type annotation was introduced in Python 3.0, but only later were its *semantics* specified.

3 This approach was also compatible with Python 2.7 code, still in widespread use at the time.

4 And `SupportsInt` uses the `runtime_checkable` decorator.

5 NASA's Jet Propulsion Lab used Python for the Persistence Mars Rover and the Ingenuity Mars Helicopter; the team responsible for the discovery of gravitational waves used Python both to coordinate the instrumentation and to analyze the resulting hoard of data.