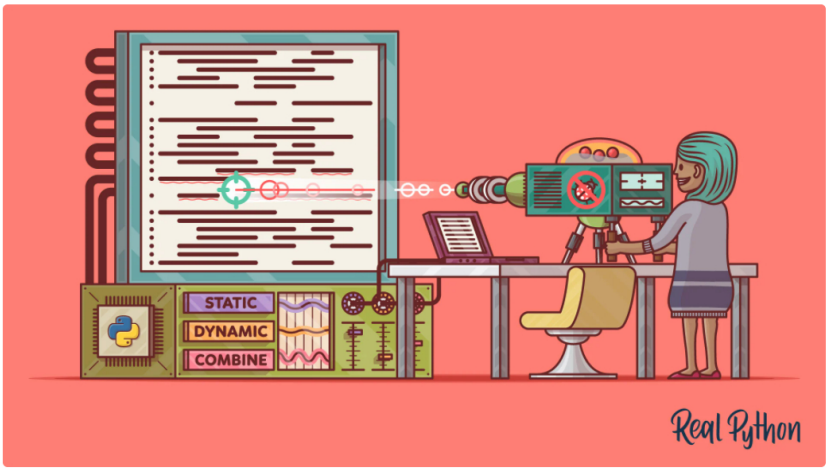


Python Type Checking (Guide)

by Geir Arne Hjelle Jan 07, 2019 38 Comments best-practices intermediate

Mark as Completed

Tweet Share Email

Table of Contents

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

- Type Systems
- Hello Types
- Pros and Cons
- Annotations
- Playing With Python Types, Part 1
- Type Theory
- Playing With Python Types, Part 2
- Static Type Checking
- Conclusion

Mark as Completed

Tweet Share Email

Recommended Video Course

Python Type Checking

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Python Type Checking**

In this guide, you will get a look into Python type checking. Traditionally, types have been handled by the Python interpreter in a flexible but implicit way. Recent versions of Python allow you to specify explicit type hints that can be used by different tools to help you develop your code more efficiently.

In this tutorial, you'll learn about the following:

- Type annotations and type hints
- Adding static types to code, both your code and the code of others
- Running a static type checker
- Enforcing types at runtime

This is a comprehensive guide that will cover a lot of ground. If you want to just get a quick glimpse of how type hints work in Python, and see whether type checking is something you would include in your code, you don't need to read all of it. The two sections Hello Types and Pros and Cons will give you a taste of how type checking works and recommendations about when it'll be useful.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Type Systems

All programming languages include some kind of type system that formalizes which categories of objects it can work with and how those categories are treated. For instance, a type system can define a numerical type, with 42 as one example of an object of numerical type.

Dynamic Typing

Python is a dynamically typed language. This means that the Python interpreter does type checking only as code runs, and that the type of a variable is allowed to change over its lifetime. The following dummy examples demonstrate that Python has dynamic typing:

Python >>>

```
>>> if False:
...     1 + "two"  # This line never runs, so no TypeError is raised
... else:
...     1 + 2
...
3
```

```
>>> 1 + "two"  # Now this is type checked, and a TypeError is raised
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the first example, the branch `1 + "two"` never runs so it's never type checked. The second example shows that when `1 + "two"` is evaluated it raises a `TypeError` since you can't add an integer and a string in Python.

Next, let's see if variables can change type:

Python >>>

```
>>> thing = "Hello"
>>> type(thing)
<class 'str'>

>>> thing = 28.1
>>> type(thing)
<class 'float'>
```

`type()` returns the type of an object. These examples confirm that the type of `thing` is allowed to change, and Python correctly infers the type as it changes.

Static Typing

The opposite of dynamic typing is static typing. Static type checks are performed without running the program. In most statically typed languages, for instance C and Java, this is done as your program is compiled.

With static typing, variables generally are not allowed to change types, although mechanisms for casting a variable to a different type may exist.

Let's look at a quick example from a statically typed language. Consider the following Java snippet:

Java

```
String thing;
thing = "Hello";
```

The first line declares that the variable name `thing` is bound to the `String` type at compile time. The name can never be rebound to another type. In the second line, `thing` is assigned a value. It can never be assigned a value that is not a `String` object. For instance, if you were to later say `thing = 28.1f` the compiler would raise an error because of incompatible types.

Python will always remain a dynamically typed language. However, PEP 484 introduced type hints, which make it possible to also do static type checking of Python code.

Unlike how types work in most other statically typed languages, type hints by themselves don't cause Python to enforce types. As the name says, type hints just suggest types. There are other tools, which you'll see later, that perform static type checking using type hints.

Duck Typing

Another term that is often used when talking about Python is duck typing. This moniker comes from the phrase "if it walks like a duck and it quacks like a duck, then it must be a duck" (or any of its variations).

Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines. Using duck typing you do not check types at all. Instead you check for the presence of a given method or attribute.

As an example, you can call `len()` on any Python object that defines a `.__len__()` method:

Python >>>

```
>>> class TheHobbit:
...     def __len__(self):
...         return 95022
...
>>> the_hobbit = TheHobbit()
```

```
>>> len(the_hobbit)
95022
```

Note that the call to `len()` gives the return value of the `.__len__()` method. In fact, the implementation of `len()` is essentially equivalent to the following:

Python

```
def len(obj):
    return obj.__len__()
```

In order to call `len(obj)`, the only real constraint on `obj` is that it must define a `.__len__()` method. Otherwise, the object can be of types as different as `str`, `list`, `dict`, or `TheHobbit`.

Duck typing is somewhat supported when doing static type checking of Python code, using structural subtyping. You'll learn more about duck typing later.

Hello Types

In this section you'll see how to add type hints to a function. The following function turns a text string into a headline by adding proper capitalization and a decorative line:

Python

```
def headline(text, align=True):
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "o")
```

By default the function returns the headline left aligned with an underline. By setting the `align` flag to `False` you can alternatively have the headline be centered with a surrounding line of `o`:

Python >>>

```
>>> print(headline("python type checking"))
Python Type Checking
--------------------

>>> print(headline("python type checking", align=False))
oooooooooooooo Python Type Checking oooooooooooooo
```

It's time for our first type hints! To add information about types to the function, you simply annotate its arguments and return value as follows:

Python

```
def headline(text: str, align: bool = True) -> str:
    ...
```

The `text: str` syntax says that the `text` argument should be of type `str`. Similarly, the optional `align` argument should have type `bool` with the default value `True`. Finally, the `-> str` notation specifies that `headline()` will return a string.

In terms of style, PEP 8 recommends the following:

- Use normal rules for colons, that is, no space before and one space after a colon: `text: str`.
- Use spaces around the `=` sign when combining an argument annotation with a default value: `align: bool = True`.
- Use spaces around the `->` arrow: `def headline(...) -> str`.

Adding type hints like this has no runtime effect: they are only hints and are not enforced on

their own. For instance, if we use a wrong type for the (admittedly badly named) `align` argument, the code still runs without any problems or warnings:

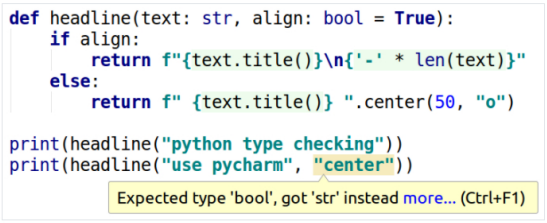
Python >>>

```
>>> print(headline("python type checking", align="left"))
Python Type Checking
--------------------
```

> **Note:** The reason this seemingly works is that the string `"left"` compares as truthy. Using `align="center"` would not have the desired effect as `"center"` is also truthy.

To catch this kind of error you can use a static type checker. That is, a tool that checks the types of your code without actually running it in the traditional sense.

You might already have such a type checker built into your editor. For instance PyCharm immediately gives you a warning:



The most common tool for doing type checking is Mypy though. You'll get a short introduction to Mypy in a moment, while you can learn much more about how it works later.

If you don't already have Mypy on your system, you can install it using `pip`:

Shell

```
$ pip install mypy
```

Put the following code in a file called `headlines.py`:

Python

```
1  # headlines.py
2
3  def headline(text: str, align: bool = True) -> str:
4      if align:
5          return f"{text.title()}\n{'-' * len(text)}"
6      else:
7          return f" {text.title()} ".center(50, "o")
8
9  print(headline("python type checking"))
10 print(headline("use mypy", align="center"))
```

This is essentially the same code you saw earlier: the definition of `headline()` and two examples that are using it.

Now run Mypy on this code:

Shell

```
$ mypy headlines.py
headlines.py:10: error: Argument "align" to "headline" has incompatible
                        type "str"; expected "bool"
```

Based on the type hints, Mypy is able to tell us that we are using the wrong type on line 10.

To fix the issue in the code you should change the value of the `align` argument you are passing in. You might also rename the `align` flag to something less confusing:

Python

```
1  # headlines.py
2
3  def headline(text: str, centered: bool = False) -> str:
4      if not centered:
```

```
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "o")

print(headline("python type checking"))
print(headline("use mypy", centered=True))
```

Here you've changed `align` to `centered`, and correctly used a Boolean value for `centered` when calling `headline()`. The code now passes Mypy:

Shell

```
$ mypy headlines.py
Success: no issues found in 1 source file
```

The success message confirms there were no type errors detected. Older versions of Mypy used to indicate this by showing no output at all. Furthermore, when you run the code you see the expected output:

Shell

```
$ python headlines.py
Python Type Checking
--------------------
oooooooooooooooooooo Use Mypy oooooooooooooooooooo
```

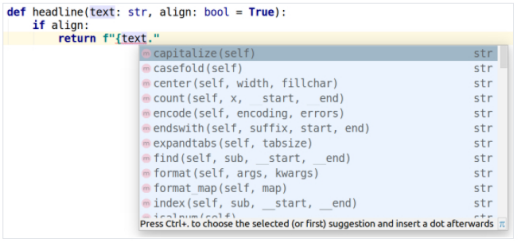
The first headline is aligned to the left, while the second one is centered.

Pros and Cons

The previous section gave you a little taste of what type checking in Python looks like. You also saw an example of one of the advantages of adding types to your code: type hints help **catch certain errors**. Other advantages include:

- Type hints help **document your code**. Traditionally, you would use docstrings if you wanted to document the expected types of a function's arguments. This works, but as there is no standard for docstrings (despite PEP 257 they can't be easily used for automatic checks.

- Type hints **improve IDEs and linters**. They make it much easier to statically reason about your code. This in turn allows IDEs to offer better code completion and similar features. With the type annotation, PyCharm knows that `text` is a string, and can give specific suggestions based on this:



- Type hints help you **build and maintain a cleaner architecture**. The act of writing type hints forces you to think about the types in your program. While the dynamic nature of Python is one of its great assets, being conscious about relying on duck typing, overloaded methods, or multiple return types is a good thing.

Of course, static type checking is not all peaches and cream. There are also some downsides you should consider:

- Type hints **take developer time and effort to add**. Even though it probably pays off in spending less time debugging, you will spend more time entering code.

- Type hints **work best in modern Pythons**. Annotations were introduced in Python 3.0, and it's possible to use type comments in Python 2.7. Still, improvements like variable annotations and postponed evaluation of type hints mean that you'll have a better experience doing type checks using Python 3.6 or even Python 3.7.

- Type hints **introduce a slight penalty in start-up time**. If you need to use the `typing` module the import time may be significant, especially in short scripts.

Measuring Import Time Show/Hide

So, should you use static type checking in your own code? Well, it's not an all-or-nothing question. Luckily, Python supports the concept of gradual typing. This means that you can gradually introduce types into your code. Code without type hints will be ignored by the static type checker. Therefore, you can start adding types to critical components, and continue as long as it adds value to you.

Looking at the lists above of pros and cons you'll notice that adding types will have no effect on your running program or the users of your program. Type checking is meant to make your life as a developer better and more convenient.

A few rules of thumb on whether to add types to your project are:

- If you are just beginning to learn Python, you can safely wait with type hints until you have more experience.

- Type hints add little value in short throw-away scripts.

- In libraries that will be used by others, especially ones published on PyPI, type hints add a lot of value. Other code using your libraries need these type hints to be properly type checked itself. For examples of projects using type hints see `cursive_re`, `black`, our own Real Python Reader, and Mypy itself.

- In bigger projects, type hints help you understand how types flow through your code, and are highly recommended. Even more so in projects where you cooperate with others.

In his excellent article The State of Type Hints in Python Bernát Gábor recommends that "**type hints should be used whenever unit tests are worth writing**." Indeed, type hints play a similar role as tests in your code: they help you as a developer write better code.

Hopefully you now have an idea about how type checking works in Python and whether it's something you would like to employ in your own projects.

In the rest of this guide, we'll go into more detail about the Python type system, including how you run static type checkers (with particular focus on Mypy), how you type check code that uses libraries without type hints, and how you use annotations at runtime.

Annotations

Annotations were introduced in Python 3.0, originally without any specific purpose. They were simply a way to associate arbitrary expressions to function arguments and return values.

Years later, PEP 484 defined how to add type hints to your Python code, based off work that Jukka Lehtosalo had done on his Ph.D. project—Mypy. The main way to add type hints is using annotations. As type checking is becoming more and more common, this also means that annotations should mainly be reserved for type hints.

The next sections explain how annotations work in the context of type hints.

Function Annotations

For functions, you can annotate arguments and the return value. This is done as follows:

```
Python
```

```
def func(arg: arg_type, optarg: arg_type = default) -> return_type:
    ...
```

For arguments the syntax is `argument: annotation`, while the return type is annotated using `-> annotation`. Note that the annotation must be a valid Python expression.

The following simple example adds annotations to a function that calculates the circumference of a circle:

Python

```
import math

def circumference(radius: float) -> float:
    return 2 * math.pi * radius
```

When running the code, you can also inspect the annotations. They are stored in a special `.__annotations__` attribute on the function:

Python >>>

```
>>> circumference(1.23)
7.728317927830891

>>> circumference.__annotations__
{'radius': <class 'float'>, 'return': <class 'float'>}
```

Sometimes you might be confused by how Mypy is interpreting your type hints. For those cases there are special Mypy expressions: `reveal_type()` and `reveal_locals()`. You can add these to your code before running Mypy, and Mypy will dutifully report which types it has inferred. As an example, save the following code to `reveal.py`:

Python

```
1 # reveal.py
2
3 import math
4 reveal_type(math.pi)
5
6 radius = 1
7 circumference = 2 * math.pi * radius
8 reveal_locals()
```

Next, run this code through Mypy:

Shell

```
$ mypy reveal.py
reveal.py:4: error: Revealed type is 'builtins.float'

reveal.py:8: error: Revealed local types are:
reveal.py:8: error: circumference: builtins.float
reveal.py:8: error: radius: builtins.int
```

Even without any annotations Mypy has correctly inferred the types of the built-in `math.pi`, as well as our local variables `radius` and `circumference`.

> **Note:** The reveal expressions are only meant as a tool helping you add types and debug your type hints. If you try to run the `reveal.py` file as a Python script it will crash with a `NameError` since `reveal_type()` is not a function known to the Python interpreter.

If Mypy says that "Name 'reveal_locals' is not defined" you might need to update your Mypy installation. The `reveal_locals()` expression is available in Mypy version 0.610 and later.

Variable Annotations

In the definition of `circumference()` in the previous section, you only annotated the arguments and the return value. You did not add any annotations inside the function body. More often than not, this is enough.

However, sometimes the type checker needs help in figuring out the types of variables as well. Variable annotations were defined in PEP 526 and introduced in Python 3.6. The syntax is the same as for function argument annotations:

```
Python
pi: float = 3.142

def circumference(radius: float) -> float:
    return 2 * pi * radius
```

The variable `pi` has been annotated with the `float` type hint.

> **Note:** Static type checkers are more than able to figure out that `3.142` is a float, so in this example the annotation of `pi` is not necessary. As you learn more about the Python type system, you'll see more relevant examples of variable annotations.

Annotations of variables are stored in the module level `__annotations__` dictionary:

```
Python                                                    >>>
>>> circumference(1)
6.284

>>> __annotations__
{'pi': <class 'float'>}
```

You're allowed to annotate a variable without giving it a value. This adds the annotation to the `__annotations__` dictionary, while the variable remains undefined:

```
Python                                                    >>>
>>> nothing: str
>>> nothing
NameError: name 'nothing' is not defined

>>> __annotations__
{'nothing': <class 'str'>}
```

Since no value was assigned to `nothing`, the name `nothing` is not yet defined.

Type Comments

As mentioned, annotations were introduced in Python 3, and they've not been backported to Python 2. This means that if you're writing code that needs to support legacy Python, you can't use annotations.

Instead, you can use type comments. These are specially formatted comments that can be used to add type hints compatible with older code. To add type comments to a function you do something like this:

```
Python
import math

def circumference(radius):
    # type: (float) -> float
    return 2 * math.pi * radius
```

The type comments are just comments, so they can be used in any version of Python.

Type comments are handled directly by the type checker, so these types are not available in the `__annotations__` dictionary:

```
Python                                                    >>>
```

```
>>> circumference.__annotations__
{}
```

A type comment must start with the `type:` literal, and be on the same or the following line as the function definition. If you want to annotate a function with several arguments, you write each type separated by comma:

Python

```python
def headline(text, width=80, fill_char="-"):
    # type: (str, int, str) -> str
    return f" {text.title()} ".center(width, fill_char)

print(headline("type comments work", width=40))
```

You are also allowed to write each argument on a separate line with its own annotation:

Python

```python
# headlines.py

def headline(
    text,           # type: str
    width=80,       # type: int
    fill_char="-",  # type: str
):                  # type: (...) -> str
    return f" {text.title()} ".center(width, fill_char)

print(headline("type comments work", width=40))
```

Run the example through Python and Mypy:

Shell

```
$  python headlines.py
---------- Type Comments Work ----------

$ mypy headlines.py
Success: no issues found in 1 source file
```

If you have errors, for instance if you happened to call `headline()` with `width="full"` on line 10, Mypy will tell you:

Shell

```
$ mypy headline.py
headline.py:10: error: Argument "width" to "headline" has incompatible
                       type "str"; expected "int"
```

You can also add type comments to variables. This is done similarly to how you add type comments to arguments:

Python

```python
pi = 3.142  # type: float
```

In this example, `pi` will be type checked as a float variable.

So, Type Annotations or Type Comments?

Should you use annotations or type comments when adding type hints to your own code? In short: **Use annotations if you can, use type comments if you must.**

Annotations provide a cleaner syntax keeping type information closer to your code. They are also the officially recommended way of writing type hints, and will be further developed and properly maintained in the future.

Type comments are more verbose and might conflict with other kinds of comments in your code like linter directives. However, they can be used in code bases that don't support annotations.

There is also hidden option number three: stub files. You will learn about these later, when we discuss adding types to third party libraries.

Stub files will work in any version of Python, at the expense of having to maintain a second set of files. In general, you only want to use stub files if you can't change the original source code.

Playing With Python Types, Part 1

Up until now you've only used basic types like `str`, `float`, and `bool` in your type hints. The Python type system is quite powerful, and supports many kinds of more complex types. This is necessary as it needs to be able to reasonably model Python's dynamic duck typing nature.

In this section you will learn more about this type system, while implementing a simple card game. You will see how to specify:

- The type of sequences and mappings like tuples, lists and dictionaries
- Type aliases that make code easier to read
- That functions and methods do not return anything
- Objects that may be of any type

After a short detour into some type theory you will then see even more ways to specify types in Python. You can find the code examples from this section here.

Example: A Deck of Cards

The following example shows an implementation of a regular (French) deck of cards:

Python

```python
# game.py

import random

SUITS = "♠ ♡ ♢ ♣".split()
RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()

def create_deck(shuffle=False):
    """Create a new deck of 52 cards"""
    deck = [(s, r) for r in RANKS for s in SUITS]
    if shuffle:
        random.shuffle(deck)
    return deck

def deal_hands(deck):
    """Deal the cards in the deck into four hands"""
    return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])

def play():
    """Play a 4-player card game"""
    deck = create_deck(shuffle=True)
    names = "P1 P2 P3 P4".split()
    hands = {n: h for n, h in zip(names, deal_hands(deck))}

    for name, cards in hands.items():
        card_str = " ".join(f"{s}{r}" for (s, r) in cards)
        print(f"{name}: {card_str}")

if __name__ == "__main__":
    play()
```

Each card is represented as a tuple of strings denoting the suit and rank. The deck is represented as a list of cards. `create_deck()` creates a regular deck of 52 playing cards, and optionally shuffles the cards. `deal_hands()` deals the deck of cards to four players.

Finally, `play()` plays the game. As of now, it only prepares for a card game by constructing a

Finally, `play()` plays the game. As of now, it only prepares for a card game by constructing a shuffled deck and dealing cards to each player. The following is a typical output:

Shell

```
$ python game.py
P4: ♣9 ♢9 ♡2 ♢7 ♡7 ♣A ♠6 ♡K ♡5 ♢6 ♢3 ♣3 ♣Q
P1: ♡A ♠2 ♠10 ♢J ♣10 ♣4 ♠5 ♡Q ♢5 ♣6 ♠A ♣5 ♢4
P2: ♢2 ♠7 ♡8 ♢K ♠3 ♡3 ♣K ♠J ♢A ♣7 ♡6 ♡10 ♠K
P3: ♣2 ♣8 ♠8 ♣J ♢Q ♡9 ♡J ♠4 ♢8 ♢10 ♠9 ♡4 ♠Q
```

You will see how to extend this example into a more interesting game as we move along.

Sequences and Mappings

Let's add type hints to our card game. In other words, let's annotate the functions `create_deck()`, `deal_hands()`, and `play()`. The first challenge is that you need to annotate composite types like the list used to represent the deck of cards and the tuples used to represent the cards themselves.

With simple types like `str`, `float`, and `bool`, adding type hints is as easy as using the type itself:

Python >>>

```
>>> name: str = "Guido"
>>> pi: float = 3.142
>>> centered: bool = False
```

With composite types, you are allowed to do the same:

Python >>>

```
>>> names: list = ["Guido", "Jukka", "Ivan"]
>>> version: tuple = (3, 7, 1)
>>> options: dict = {"centered": False, "capitalize": True}
```

However, this does not really tell the full story. What will be the types of `names[2]`, `version[0]`, and `options["centered"]`? In this concrete case you can see that they are `str`, `int`, and `bool`, respectively. However, the type hints themselves give no information about this.

Instead, you should use the special types defined in the `typing` module. These types add syntax for specifying the types of elements of composite types. You can write the following:

Python >>>

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

>>> names: List[str] = ["Guido", "Jukka", "Ivan"]
>>> version: Tuple[int, int, int] = (3, 7, 1)
>>> options: Dict[str, bool] = {"centered": False, "capitalize": True}
```

Note that each of these types start with a capital letter and that they all use square brackets to define item types:

- **`names`** is a list of strings
- **`version`** is a 3-tuple consisting of three integers
- **`options`** is a dictionary mapping strings to Boolean values

The `typing` module contains many more composite types, including `Counter`, `Deque`, `FrozenSet`, `NamedTuple`, and `Set`. In addition, the module includes other kinds of types that you'll see in later sections.

Let's return to the card game. A card is represented by a tuple of two strings. You can write this as `Tuple[str, str]`, so the type of the deck of cards becomes `List[Tuple[str, str]]`. Therefore you can annotate `create_deck()` as follows:

Python

```
8 def create_deck(shuffle: bool = False) -> List[Tuple[str, str]]:
```

```
    """Create a new deck of 52 cards"""
    deck = [(s, r) for r in RANKS for s in SUITS]
    if shuffle:
        random.shuffle(deck)
    return deck
```

In addition to the return value, you've also added the `bool` type to the optional `shuffle` argument.

> **Note:** Tuples and lists are annotated differently.
>
> A tuple is an immutable sequence, and typically consists of a fixed number of possibly differently typed elements. For example, we represent a card as a tuple of suit and rank. In general, you write `Tuple[t_1, t_2, ..., t_n]` for an n-tuple.
>
> A list is a mutable sequence and usually consists of an unknown number of elements of the same type, for instance a list of cards. No matter how many elements are in the list there is only one type in the annotation: `List[t]`.

In many cases your functions will expect some kind of sequence, and not really care whether it is a list or a tuple. In these cases you should use `typing.Sequence` when annotating the function argument:

Python

```
from typing import List, Sequence

def square(elems: Sequence[float]) -> List[float]:
    return [x**2 for x in elems]
```

Using `Sequence` is an example of using duck typing. A `Sequence` is anything that supports `len()` and `.__getitem__()`, independent of its actual type.

Type Aliases

The type hints might become quite oblique when working with nested types like the deck of cards. You may need to stare at `List[Tuple[str, str]]` a bit before figuring out that it matches our representation of a deck of cards.

Now consider how you would annotate `deal_hands()`:

Python

```
def deal_hands(
    deck: List[Tuple[str, str]]
) -> Tuple[
    List[Tuple[str, str]],
    List[Tuple[str, str]],
    List[Tuple[str, str]],
    List[Tuple[str, str]],
]:
    """Deal the cards in the deck into four hands"""
    return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])
```

That's just terrible!

Recall that type annotations are regular Python expressions. That means that you can define your own type aliases by assigning them to new variables. You can for instance create `Card` and `Deck` type aliases:

Python

```
from typing import List, Tuple

Card = Tuple[str, str]
Deck = List[Card]
```

```
Deck = List[Card]
```

`Card` can now be used in type hints or in the definition of new type aliases, like `Deck` in the example above.

Using these aliases, the annotations of `deal_hands()` become much more readable:

Python

```
15 def deal_hands(deck: Deck) -> Tuple[Deck, Deck, Deck, Deck]:
16     """Deal the cards in the deck into four hands"""
17     return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])
```

Type aliases are great for making your code and its intent clearer. At the same time, these aliases can be inspected to see what they represent:

Python >>>

```
>>> from typing import List, Tuple
>>> Card = Tuple[str, str]
>>> Deck = List[Card]

>>> Deck
typing.List[typing.Tuple[str, str]]
```

Note that when printing `Deck`, it shows that it's an alias for a list of 2-tuples of strings.

Functions Without Return Values

You may know that functions without an explicit return still return `None`:

Python >>>

```
>>> def play(player_name):
...     print(f"{player_name} plays")
...

>>> ret_val = play("Jacob")
Jacob plays

>>> print(ret_val)
None
```

While such functions technically return something, that return value is not useful. You should add type hints saying as much by using `None` also as the return type:

Python

```
1 # play.py
2
3 def play(player_name: str) -> None:
4     print(f"{player_name} plays")
5
6 ret_val = play("Filip")
```

The annotations help catch the kinds of subtle bugs where you are trying to use a meaningless return value. Mypy will give you a helpful warning:

Shell

```
$ mypy play.py
play.py:6: error: "play" does not return a value
```

Note that being explicit about a function not returning anything is different from not adding a type hint about the return value:

Python

```
# play.py

def play(player_name: str):
    print(f"{player_name} plays")

ret_val = play("Henrik")
```

In this latter case Mypy has no information about the return value so it will not generate any warning:

Shell

```
$ mypy play.py
Success: no issues found in 1 source file
```

As a more exotic case, note that you can also annotate functions that are never expected to return normally. This is done using `NoReturn`:

Python

```
from typing import NoReturn

def black_hole() -> NoReturn:
    raise Exception("There is no going back ...")
```

Since `black_hole()` always raises an exception, it will never return properly.

Example: Play Some Cards

Let's return to our card game example. In this second version of the game, we deal a hand of cards to each player as before. Then a start player is chosen and the players take turns playing their cards. There are not really any rules in the game though, so the players will just play random cards:

Python

```
# game.py

import random
from typing import List, Tuple

SUITS = "♠ ♡ ♢ ♣".split()
RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()

Card = Tuple[str, str]
Deck = List[Card]

def create_deck(shuffle: bool = False) -> Deck:
    """Create a new deck of 52 cards"""
    deck = [(s, r) for r in RANKS for s in SUITS]
    if shuffle:
        random.shuffle(deck)
    return deck

def deal_hands(deck: Deck) -> Tuple[Deck, Deck, Deck, Deck]:
    """Deal the cards in the deck into four hands"""
    return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])

def choose(items):
    """Choose and return a random item"""
    return random.choice(items)

def player_order(names, start=None):
    """Rotate player order so that start goes first"""
    if start is None:
        start = choose(names)
    start_idx = names.index(start)
    return names[start_idx:] + names[:start_idx]

def play() -> None:
    """Play a 4-player card game"""
    deck = create_deck(shuffle=True)
    names = "P1 P2 P3 P4".split()
    hands = {n: h for n, h in zip(names, deal_hands(deck))}
```

```
    start_player = choose(names)
    turn_order = player_order(names, start=start_player)

    # Randomly play cards from each player's hand until empty
    while hands[start_player]:
        for name in turn_order:
            card = choose(hands[name])
            hands[name].remove(card)
            print(f"{name}: {card[0] + card[1]:<3}  ", end="")
        print()

if __name__ == "__main__":
    play()
```

Note that in addition to changing `play()`, we have added two new functions that need type hints: `choose()` and `player_order()`. Before discussing how we'll add type hints to them, here is an example output from running the game:

Shell

```
$ python game.py
P3: ♢10  P4: ♣4   P1: ♡8   P2: ♡Q
P3: ♣8   P4: ♣6   P1: ♠5   P2: ♡K
P3: ♢9   P4: ♡J   P1: ♣A   P2: ♡A
P3: ♠Q   P4: ♣3   P1: ♠7   P2: ♠A
P3: ♡4   P4: ♡6   P1: ♣2   P2: ♠K
P3: ♣K   P4: ♣7   P1: ♡7   P2: ♠2
P3: ♣10  P4: ♠4   P1: ♢5   P2: ♡3
P3: ♣Q   P4: ♢K   P1: ♣J   P2: ♡9
P3: ♢2   P4: ♢4   P1: ♠9   P2: ♠10
P3: ♢A   P4: ♡5   P1: ♠J   P2: ♢Q
P3: ♠8   P4: ♢7   P1: ♢3   P2: ♢J
P3: ♣3   P4: ♡10  P1: ♣9   P2: ♡2
P3: ♢6   P4: ♣6   P1: ♣5   P2: ♢8
```

In this example, player `P3` was randomly chosen as the starting player. In turn, each player plays a card: first `P3`, then `P4`, then `P1`, and finally `P2`. The players keep playing cards as long as they have any left in their hand.

The `Any` Type

`choose()` works for both lists of names and lists of cards (and any other sequence for that matter). One way to add type hints for this would be the following:

Python

```
import random
from typing import Any, Sequence

def choose(items: Sequence[Any]) -> Any:
    return random.choice(items)
```

This means more or less what it says: `items` is a sequence that can contain items of any type and `choose()` will return one such item of any type. Unfortunately, this is not that useful. Consider the following example:

Python

```
# choose.py

import random
from typing import Any, Sequence

def choose(items: Sequence[Any]) -> Any:
    return random.choice(items)

names = ["Guido", "Jukka", "Ivan"]
reveal_type(names)

name = choose(names)
reveal_type(name)
```

While Mypy will correctly infer that `names` is a list of strings, that information is lost after the call to `choose()` because of the use of the `Any` type:

call to `choose()` because of the use of the `Any` type:

```
Shell
$ mypy choose.py
choose.py:10: error: Revealed type is 'builtins.list[builtins.str*]'
choose.py:13: error: Revealed type is 'Any'
```

You'll see a better way shortly. First though, let's have a more theoretical look at the Python type system, and the special role `Any` plays.

Type Theory

This tutorial is mainly a practical guide and we will only scratch the surface of the theory underpinning Python type hints. For more details PEP 483 is a good starting point. If you want to get back to the practical examples, feel free to skip to the next section.

Subtypes

One important concept is that of **subtypes**. Formally, we say that a type `T` is a subtype of `U` if the following two conditions hold:

- Every value from `T` is also in the set of values of `U` type.
- Every function from `U` type is also in the set of functions of `T` type.

These two conditions guarantees that even if type `T` is different from `U`, variables of type `T` can always pretend to be `U`.

For a concrete example, consider `T = bool` and `U = int`. The `bool` type takes only two values. Usually these are denoted `True` and `False`, but these names are just aliases for the integer values `1` and `0`, respectively:

```
Python >>>
>>> int(False)
0

>>> int(True)
1

>>> True + True
2

>>> issubclass(bool, int)
True
```

Since 0 and 1 are both integers, the first condition holds. Above you can see that Booleans can be added together, but they can also do anything else integers can. This is the second condition above. In other words, `bool` is a subtype of `int`.

The importance of subtypes is that a subtype can always pretend to be its supertype. For instance, the following code type checks as correct:

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

print(double(True))  # Passing in bool instead of int
```

Subtypes are somewhat related to subclasses. In fact all subclasses corresponds to subtypes, and `bool` is a subtype of `int` because `bool` is a subclass of `int`. However, there are also subtypes that do not correspond to subclasses. For instance `int` is a subtype of `float`, but `int` is not a subclass of `float`.

Covariant, Contravariant, and Invariant

What happens when you use subtypes inside composite types? For instance, is `Tuple[bool]` a subtype of `Tuple[int]`? The answer depends on the composite type, and whether that type is covariant, contravariant, or invariant. This gets technical fast, so let's just give a few examples:

- `Tuple` is covariant. This means that it preserves the type hierarchy of its item types: `Tuple[bool]` is a subtype of `Tuple[int]` because `bool` is a subtype of `int`.
- `List` is invariant. Invariant types give no guarantee about subtypes. While all values of `List[bool]` are values of `List[int]`, you can append an `int` to `List[int]` and not to `List[bool]`. In other words, the second condition for subtypes does not hold, and `List[bool]` is not a subtype of `List[int]`.
- `Callable` is contravariant in its arguments. This means that it reverses the type hierarchy. You will see how `Callable` works later, but for now think of `Callable[[T], ...]` as a function with its only argument being of type `T`. An example of a `Callable[[int], ...]` is the `double()` function defined above. Being contravariant means that if a function operating on a `bool` is expected, then a function operating on an `int` would be acceptable.

In general, you don't need to keep these expression straight. However, you should be aware that subtypes and composite types may not be simple and intuitive.

Gradual Typing and Consistent Types

Earlier we mentioned that Python supports gradual typing, where you can gradually add type hints to your Python code. Gradual typing is essentially made possible by the `Any` type.

Somehow `Any` sits both at the top and at the bottom of the type hierarchy of subtypes. Any type behaves as if it is a subtype of `Any`, and `Any` behaves as if it is a subtype of any other type. Looking at the definition of subtypes above this is not really possible. Instead we talk about **consistent types**.

The type `T` is consistent with the type `U` if `T` is a subtype of `U` or either `T` or `U` is `Any`.

The type checker only complains about inconsistent types. The takeaway is therefore that you will never see type errors arising from the `Any` type.

This means that you can use `Any` to explicitly fall back to dynamic typing, describe types that are too complex to describe in the Python type system, or describe items in composite types. For instance, a dictionary with string keys that can take any type as its values can be annotated `Dict[str, Any]`.

Do remember, though, if you use `Any` the static type checker will effectively not do any type any checking.

Playing With Python Types, Part 2

Let's return to our practical examples. Recall that you were trying to annotate the general `choose()` function:

Python

```
import random
from typing import Any, Sequence

def choose(items: Sequence[Any]) -> Any:
    return random.choice(items)
```

The problem with using `Any` is that you are needlessly losing type information. You know that if you pass a list of strings to `choose()`, it will return a string. Below you'll see how to express this using type variables, as well as how to work with:

- Duck types and protocols
- Arguments with `None` as default value

- Class methods
- The type of your own classes
- Variable number of arguments

Type Variables

A type variable is a special variable that can take on any type, depending on the situation.

Let's create a type variable that will effectively encapsulate the behavior of `choose()`:

Python

```python
# choose.py

import random
from typing import Sequence, TypeVar

Choosable = TypeVar("Choosable")

def choose(items: Sequence[Choosable]) -> Choosable:
    return random.choice(items)

names = ["Guido", "Jukka", "Ivan"]
reveal_type(names)

name = choose(names)
reveal_type(name)
```

A type variable must be defined using `TypeVar` from the `typing` module. When used, a type variable ranges over all possible types and takes the most specific type possible. In the example, `name` is now a `str`:

Shell

```
$ mypy choose.py
choose.py:12: error: Revealed type is 'builtins.list[builtins.str*]'
choose.py:15: error: Revealed type is 'builtins.str*'
```

Consider a few other examples:

Python

```python
# choose_examples.py

from choose import choose

reveal_type(choose(["Guido", "Jukka", "Ivan"]))
reveal_type(choose([1, 2, 3]))
reveal_type(choose([True, 42, 3.14]))
reveal_type(choose(["Python", 3, 7]))
```

The first two examples should have type `str` and `int`, but what about the last two? The individual list items have different types, and in that case the `Choosable` type variable does its best to accommodate:

Shell

```
$ mypy choose_examples.py
choose_examples.py:5: error: Revealed type is 'builtins.str*'
choose_examples.py:6: error: Revealed type is 'builtins.int*'
choose_examples.py:7: error: Revealed type is 'builtins.float*'
choose_examples.py:8: error: Revealed type is 'builtins.object*'
```

As you've already seen `bool` is a subtype of `int`, which again is a subtype of `float`. So in the third example the return value of `choose()` is guaranteed to be something that can be thought of as a `float`. In the last example, there is no subtype relationship between `str` and `int`, so the best that can be said about the return value is that it is an object.

Note that none of these examples raised a type error. Is there a way to tell the type checker that `choose()` should accept both strings and numbers, but not both at the same time?

You can constrain type variables by listing the acceptable types:

Python

```python
# choose.py

import random
from typing import Sequence, TypeVar

Choosable = TypeVar("Choosable", str, float)

def choose(items: Sequence[Choosable]) -> Choosable:
    return random.choice(items)

reveal_type(choose(["Guido", "Jukka", "Ivan"]))
reveal_type(choose([1, 2, 3]))
reveal_type(choose([True, 42, 3.14]))
reveal_type(choose(["Python", 3, 7]))
```

Now `Choosable` can only be either `str` or `float`, and Mypy will note that the last example is an error:

Shell

```
$ mypy choose.py
choose.py:11: error: Revealed type is 'builtins.str*'
choose.py:12: error: Revealed type is 'builtins.float*'
choose.py:13: error: Revealed type is 'builtins.float*'
choose.py:14: error: Revealed type is 'builtins.object*'
choose.py:14: error: Value of type variable "Choosable" of "choose"
                     cannot be "object"
```

Also note that in the second example the type is considered `float` even though the input list only contains `int` objects. This is because `Choosable` was restricted to strings and floats and `int` is a subtype of `float`.

In our card game we want to restrict `choose()` to be used for `str` and `Card`:

Python

```python
Choosable = TypeVar("Choosable", str, Card)

def choose(items: Sequence[Choosable]) -> Choosable:
    ...
```

We briefly mentioned that `Sequence` represents both lists and tuples. As we noted, a `Sequence` can be thought of as a duck type, since it can be any object with `.__len__()` and `.__getitem__()` implemented.

Duck Types and Protocols

Recall the following example from the introduction:

Python

```python
def len(obj):
    return obj.__len__()
```

`len()` can return the length of any object that has implemented the `.__len__()` method. How can we add type hints to `len()`, and in particular the `obj` argument?

The answer hides behind the academic sounding term structural subtyping. One way to categorize type systems is by whether they are **nominal** or **structural**:

- In a **nominal** system, comparisons between types are based on names and declarations. The Python type system is mostly nominal, where an `int` can be used in place of a `float` because of their subtype relationship.

- In a **structural** system, comparisons between types are based on structure. You could define a structural type `Sized` that includes all instances that define `.__len__()`, irrespective of their nominal type.

There is ongoing work to bring a full-fledged structural type system to Python through PEP 544 which aims at adding a concept called protocols. Most of PEP 544 is already implemented in Mypy though.

A protocol specifies one or more methods that must be implemented. For example, all classes defining `.__len__()` fulfill the `typing.Sized` protocol. We can therefore annotate `len()` as follows:

Python

```
from typing import Sized

def len(obj: Sized) -> int:
    return obj.__len__()
```

Other examples of protocols defined in the `typing` module include `Container`, `Iterable`, `Awaitable`, and `ContextManager`.

You can also define your own protocols. This is done by inheriting from `Protocol` and defining the function signatures (with empty function bodies) that the protocol expects. The following example shows how `len()` and `Sized` could have been implemented:

Python

```
from typing_extensions import Protocol

class Sized(Protocol):
    def __len__(self) -> int: ...

def len(obj: Sized) -> int:
    return obj.__len__()
```

At the time of writing the support for self-defined protocols is still experimental and only available through the `typing_extensions` module. This module must be explicitly installed from PyPI by doing `pip install typing-extensions`.

The `Optional` Type

A common pattern in Python is to use `None` as a default value for an argument. This is usually done either to avoid problems with mutable default values or to have a sentinel value flagging special behavior.

In the card example, the `player_order()` function uses `None` as a sentinel value for `start` saying that if no start player is given it should be chosen randomly:

Python

```
27 def player_order(names, start=None):
28     """Rotate player order so that start goes first"""
29     if start is None:
30         start = choose(names)
31     start_idx = names.index(start)
32     return names[start_idx:] + names[:start_idx]
```

The challenge this creates for type hinting is that in general `start` should be a string. However, it may also take the special non-string value `None`.

In order to annotate such arguments you can use the `Optional` type:

Python

```
from typing import Sequence, Optional

def player_order(
    names: Sequence[str], start: Optional[str] = None
) -> Sequence[str]:
    ...
```

The `Optional` type simply says that a variable either has the type specified or is `None`. An equivalent way of specifying the same would be using the `Union` type: `Union[None, str]`

Note that when using either `Optional` or `Union` you must take care that the variable has the correct type when you operate on it. This is done in the example by testing whether `start is None`. Not doing so would cause both static type errors as well as possible runtime errors:

Python

```python
# player_order.py

from typing import Sequence, Optional

def player_order(
    names: Sequence[str], start: Optional[str] = None
) -> Sequence[str]:
    start_idx = names.index(start)
    return names[start_idx:] + names[:start_idx]
```

Mypy tells you that you have not taken care of the case where `start` is `None`:

Shell

```
$ mypy player_order.py
player_order.py:8: error: Argument 1 to "index" of "list" has incompatible
                          type "Optional[str]"; expected "str"
```

Note: The use of `None` for optional arguments is so common that Mypy handles it automatically. Mypy assumes that a default argument of `None` indicates an optional argument even if the type hint does not explicitly say so. You could have used the following:

Python

```python
def player_order(names: Sequence[str], start: str = None) -> Sequence[str]:
    ...
```

If you don't want Mypy to make this assumption you can turn it off with the `--no-implicit-optional` command line option.

Example: The Object(ive) of the Game

Let's rewrite the card game to be more object-oriented. This will allow us to discuss how to properly annotate classes and methods.

A more or less direct translation of our card game into code that uses classes for `Card`, `Deck`, `Player`, and `Game` looks something like the following:

Python

```python
# game.py

import random
import sys

class Card:
    SUITS = "♠ ♡ ♢ ♣".split()
    RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __repr__(self):
        return f"{self.suit}{self.rank}"

```

```
class Deck:
    def __init__(self, cards):
        self.cards = cards

    @classmethod
    def create(cls, shuffle=False):
        """Create a new deck of 52 cards"""
        cards = [Card(s, r) for r in Card.RANKS for s in Card.SUITS]
        if shuffle:
            random.shuffle(cards)
        return cls(cards)

    def deal(self, num_hands):
        """Deal the cards in the deck into a number of hands"""
        cls = self.__class__
        return tuple(cls(self.cards[i::num_hands]) for i in range(num_hands))

class Player:
    def __init__(self, name, hand):
        self.name = name
        self.hand = hand

    def play_card(self):
        """Play a card from the player's hand"""
        card = random.choice(self.hand.cards)
        self.hand.cards.remove(card)
        print(f"{self.name}: {card!r:<3}  ", end="")
        return card

class Game:
    def __init__(self, *names):
        """Set up the deck and deal cards to 4 players"""
        deck = Deck.create(shuffle=True)
        self.names = (list(names) + "P1 P2 P3 P4".split())[:4]
        self.hands = {
            n: Player(n, h) for n, h in zip(self.names, deck.deal(4))
        }

    def play(self):
        """Play a card game"""
        start_player = random.choice(self.names)
        turn_order = self.player_order(start=start_player)

        # Play cards from each player's hand until empty
        while self.hands[start_player].hand.cards:
            for name in turn_order:
                self.hands[name].play_card()
            print()

    def player_order(self, start=None):
        """Rotate player order so that start goes first"""
        if start is None:
            start = random.choice(self.names)
        start_idx = self.names.index(start)
        return self.names[start_idx:] + self.names[:start_idx]

if __name__ == "__main__":
    # Read player names from command line
    player_names = sys.argv[1:]
    game = Game(*player_names)
    game.play()
```

Now let's add types to this code.

Type Hints for Methods

First of all type hints for methods work much the same as type hints for functions. The only difference is that the `self` argument need not be annotated, as it always will be a class instance. The types of the `Card` class are easy to add:

Python

```
class Card:
    SUITS = "♠ ♡ ♢ ♣".split()
    RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()

```

```
    def __init__(self, suit: str, rank: str) -> None:
        self.suit = suit
        self.rank = rank

    def __repr__(self) -> str:
        return f"{self.suit}{self.rank}"
```

Note that the `.__init__()` method always should have `None` as its return type.

Classes as Types

There is a correspondence between classes and types. For example, all instances of the `Card` class together form the `Card` type. To use classes as types you simply use the name of the class.

For example, a `Deck` essentially consists of a list of `Card` objects. You can annotate this as follows:

Python

```
class Deck:
    def __init__(self, cards: List[Card]) -> None:
        self.cards = cards
```

Mypy is able to connect your use of `Card` in the annotation with the definition of the `Card` class.

This doesn't work as cleanly though when you need to refer to the class currently being defined. For example, the `Deck.create()` class method returns an object with type `Deck`. However, you can't simply add `-> Deck` as the `Deck` class is not yet fully defined.

Instead, you are allowed to use string literals in annotations. These strings will only be evaluated by the type checker later, and can therefore contain self and forward references. The `.create()` method should use such string literals for its types:

Python

```
class Deck:
    @classmethod
    def create(cls, shuffle: bool = False) -> "Deck":
        """Create a new deck of 52 cards"""
        cards = [Card(s, r) for r in Card.RANKS for s in Card.SUITS]
        if shuffle:
            random.shuffle(cards)
        return cls(cards)
```

Note that the `Player` class also will reference the `Deck` class. This is however no problem, since `Deck` is defined before `Player`:

Python

```
class Player:
    def __init__(self, name: str, hand: Deck) -> None:
        self.name = name
        self.hand = hand
```

Usually annotations are not used at runtime. This has given wings to the idea of postponing the evaluation of annotations. Instead of evaluating annotations as Python expressions and storing their value, the proposal is to store the string representation of the annotation and only evaluate it when needed.

Such functionality is planned to become standard in the still mythical Python 4.0. However, in Python 3.7 and later, forward references are available through a `__future__` import:

Python

```
from __future__ import annotations

class Deck:
    @classmethod
    def create(cls, shuffle: bool = False) -> Deck:
        ...
```

With the `__future__` import you can use `Deck` instead of `"Deck"` even before `Deck` is defined.

Returning `self` or `cls`

As noted, you should typically not annotate the `self` or `cls` arguments. Partly, this is not necessary as `self` points to an instance of the class, so it will have the type of the class. In the `Card` example, `self` has the implicit type `Card`. Also, adding this type explicitly would be cumbersome since the class is not defined yet. You would have to use the string literal syntax, `self: "Card"`.

There is one case where you might want to annotate `self` or `cls`, though. Consider what happens if you have a superclass that other classes inherit from, and which has methods that return `self` or `cls`:

Python

```python
# dogs.py

from datetime import date

class Animal:
    def __init__(self, name: str, birthday: date) -> None:
        self.name = name
        self.birthday = birthday

    @classmethod
    def newborn(cls, name: str) -> "Animal":
        return cls(name, date.today())

    def twin(self, name: str) -> "Animal":
        cls = self.__class__
        return cls(name, self.birthday)

class Dog(Animal):
    def bark(self) -> None:
        print(f"{self.name} says woof!")

fido = Dog.newborn("Fido")
pluto = fido.twin("Pluto")
fido.bark()
pluto.bark()
```

While the code runs without problems, Mypy will flag a problem:

Shell

```
$ mypy dogs.py
dogs.py:24: error: "Animal" has no attribute "bark"
dogs.py:25: error: "Animal" has no attribute "bark"
```

The issue is that even though the inherited `Dog.newborn()` and `Dog.twin()` methods will return a `Dog` the annotation says that they return an `Animal`.

In cases like this you want to be more careful to make sure the annotation is correct. The return type should match the type of `self` or the instance type of `cls`. This can be done using type variables that keep track of what is actually passed to `self` and `cls`:

Python

```python
# dogs.py

from datetime import date
from typing import Type, TypeVar

TAnimal = TypeVar("TAnimal", bound="Animal")

class Animal:
    def __init__(self, name: str, birthday: date) -> None:
        self.name = name
        self.birthday = birthday

    @classmethod
```

```
    def newborn(cls: Type[TAnimal], name: str) -> TAnimal:
        return cls(name, date.today())

    def twin(self: TAnimal, name: str) -> TAnimal:
        cls = self.__class__
        return cls(name, self.birthday)

class Dog(Animal):
    def bark(self) -> None:
        print(f"{self.name} says woof!")

fido = Dog.newborn("Fido")
pluto = fido.twin("Pluto")
fido.bark()
pluto.bark()
```

There are a few things to note in this example:

- The type variable `TAnimal` is used to denote that return values might be instances of subclasses of `Animal`.
- We specify that `Animal` is an upper bound for `TAnimal`. Specifying `bound` means that `TAnimal` will only be `Animal` or one of its subclasses. This is needed to properly restrict the types that are allowed.
- The `typing.Type[]` construct is the typing equivalent of `type()`. You need it to note that the class method expects a class and returns an instance of that class.

Annotating `*args` and `**kwargs`

In the object oriented version of the game, we added the option to name the players on the command line. This is done by listing player names after the name of the program:

Shell

```
$ python game.py GeirArne Dan Joanna
Dan: ♢A   Joanna: ♡9   P1: ♣A   GeirArne: ♣2
Dan: ♡A   Joanna: ♡6   P1: ♠4   GeirArne: ♢8
Dan: ♢K   Joanna: ♢Q   P1: ♣K   GeirArne: ♠5
Dan: ♡2   Joanna: ♡J   P1: ♠7   GeirArne: ♡K
Dan: ♢10  Joanna: ♣3   P1: ♢4   GeirArne: ♠8
Dan: ♣6   Joanna: ♡Q   P1: ♣Q   GeirArne: ♢J
Dan: ♢2   Joanna: ♡4   P1: ♣8   GeirArne: ♡7
Dan: ♡10  Joanna: ♢3   P1: ♡3   GeirArne: ♠2
Dan: ♠K   Joanna: ♣5   P1: ♣7   GeirArne: ♠J
Dan: ♠6   Joanna: ♢9   P1: ♣J   GeirArne: ♣10
Dan: ♠3   Joanna: ♡5   P1: ♣9   GeirArne: ♠Q
Dan: ♠A   Joanna: ♠9   P1: ♠10  GeirArne: ♡8
Dan: ♢6   Joanna: ♢5   P1: ♢7   GeirArne: ♣4
```

This is implemented by unpacking and passing in `sys.argv` to `Game()` when it's instantiated. The `.__init__()` method uses `*names` to pack the given names into a tuple.

Regarding type annotations: even though `names` will be a tuple of strings, you should only annotate the type of each name. In other words, you should use `str` and not `Tuple[str]`:

Python

```
46 class Game:
47     def __init__(self, *names: str) -> None:
48         """Set up the deck and deal cards to 4 players"""
49         deck = Deck.create(shuffle=True)
50         self.names = (list(names) + "P1 P2 P3 P4".split())[:4]
51         self.hands = {
52             n: Player(n, h) for n, h in zip(self.names, deck.deal(4))
53         }
```

Similarly, if you have a function or method accepting `**kwargs`, then you should only annotate the type of each possible keyword argument.

Callables

Functions are first-class objects in Python. This means that you can use functions as

Functions are first class objects in Python. This means that you can use functions as arguments to other functions. That also means that you need to be able to add type hints representing functions.

Functions, as well as lambdas, methods and classes, are represented by `typing.Callable`. The types of the arguments and the return value are usually also represented. For instance, `Callable[[A1, A2, A3], Rt]` represents a function with three arguments with types `A1`, `A2`, and `A3`, respectively. The return type of the function is `Rt`.

In the following example, the function `do_twice()` calls a given function twice and prints the return values:

Python

```
1  # do_twice.py
2
3  from typing import Callable
4
5  def do_twice(func: Callable[[str], str], argument: str) -> None:
6      print(func(argument))
7      print(func(argument))
8
9  def create_greeting(name: str) -> str:
10     return f"Hello {name}"
11
12 do_twice(create_greeting, "Jekyll")
```

Note the annotation of the `func` argument to `do_twice()` on line 5. It says that `func` should be a callable with one string argument, that also returns a string. One example of such a callable is `create_greeting()` defined on line 9.

Most callable types can be annotated in a similar manner. However, if you need more flexibility, check out callback protocols and extended callable types.

Example: Hearts

Let's end with a full example of the game of Hearts. You might already know this game from other computer simulations. Here is a quick recap of the rules:

- Four players play with a hand of 13 cards each.
- The player holding the ♣2 starts the first round, and must play ♣2.
- Players take turns playing cards, following the leading suit if possible.
- The player playing the highest card in the leading suit wins the trick, and becomes start player in the next turn.
- A player can not lead with a ♡ until a ♡ has already been played in an earlier trick.
- After all cards are played, players get points if they take certain cards:
  - 13 points for the ♠Q
  - 1 point for each ♡
- A game lasts several rounds, until one player has 100 points or more. The player with the least points wins.

More details can be found found online.

There are not many new typing concepts in this example that you have not already seen. We'll therefore not go through this code in detail, but leave it as an example of annotated code.

Source Code for the Hearts Card Game Show/Hide

Here are a few points to note in the code:

- For type relationships that are hard to express using `Union` or type variables, you can use the `@overload` decorator. See `Deck.__getitem__()` for an example and the

documentation for more information.

- Subclasses correspond to subtypes, so that a `HumanPlayer` can be used wherever a `Player` is expected.
- When a subclass reimplements a method from a superclass, the type annotations must match. See `HumanPlayer.play_card()` for an example.

When starting the game, you control the first player. Enter numbers to choose which cards to play. The following is an example of game play, with the highlighted lines showing where the player made a choice:

Shell

```
$ python hearts.py GeirArne Aldren Joanna Brad

Starting new round:
Brad -> ♣2
  0: ♣5  1: ♣Q  2: ♣K  (Rest: ♢6 ♡10 ♡6 ♠J ♡3 ♡9 ♢10 ♠7 ♠K ♠4)
  GeirArne, choose card: 2
GeirArne => ♣K
Aldren -> ♣10
Joanna -> ♣9
GeirArne wins the trick

  0: ♠4  1: ♠5  2: ♢6  3: ♠7  4: ♢10  5: ♠J  6: ♣Q  7: ♠K  (Rest: ♡10 ♡6 ♡3 ♡9)
  GeirArne, choose card: 0
GeirArne => ♠4
Aldren -> ♠5
Joanna -> ♠3
Brad -> ♠2
Aldren wins the trick

...

Joanna -> ♡J
Brad -> ♡2
  0: ♡6  1: ♡9  (Rest: )
  GeirArne, choose card: 1
GeirArne => ♡9
Aldren -> ♡A
Aldren wins the trick

Aldren -> ♣A
Joanna -> ♡Q
Brad -> ♣J
  0: ♡6  (Rest: )
  GeirArne, choose card: 0
GeirArne => ♡6
Aldren wins the trick

Scores:
Brad            14  14
Aldren          10  10
GeirArne         1   1
Joanna           1   1
```

Static Type Checking

So far you have seen how to add type hints to your code. In this section you'll learn more about how to actually perform static type checking of Python code.

The Mypy Project

Mypy was started by Jukka Lehtosalo during his Ph.D. studies at Cambridge around 2012. Mypy was originally envisioned as a Python variant with seamless dynamic and static typing. See Jukka's slides from PyCon Finland 2012 for examples of the original vision of Mypy.

Most of those original ideas still play a big part in the Mypy project. In fact, the slogan "Seamless dynamic and static typing" is still prominently visible on Mypy's home page and describes the motivation for using type hints in Python well.

The [illegible] version [illegible] that Mypy is [illegible] of Python's first

The biggest change since 2012 is that Mypy is no longer a *variant* of Python. In its first versions Mypy was a stand-alone language that was compatible with Python except for its type declarations. Following a suggestion by Guido van Rossum, Mypy was rewritten to use annotations instead. Today Mypy is a static type checker for *regular* Python code.

Running Mypy

Before running Mypy for the first time, you must install the program. This is most easily done using `pip`:

Shell

```
$ pip install mypy
```

With Mypy installed, you can run it as a regular command line program:

Shell

```
$ mypy my_program.py
```

Running Mypy on your `my_program.py` Python file will check it for type errors without actually executing the code.

There are many available options when type checking your code. As Mypy is still under very active development, command line options are liable to change between versions. You should refer to Mypy's help to see which settings are default on your version:

Shell

```
$ mypy --help
usage: mypy [-h] [-v] [-V] [more options; see below]
            [-m MODULE] [-p PACKAGE] [-c PROGRAM_TEXT] [files ...]

Mypy is a program that will type check your Python code.

[... The rest of the help hidden for brevity ...]
```

Additionally, the Mypy command line documentation online has a lot of information.

Let's look at some of the most common options. First of all, if you are using third-party packages without type hints, you may want to silence Mypy's warnings about these. This can be done with the `--ignore-missing-imports` option.

The following example uses Numpy to calculate and print the cosine of several numbers:

Python

```
1  # cosine.py
2
3  import numpy as np
4
5  def print_cosine(x: np.ndarray) -> None:
6      with np.printoptions(precision=3, suppress=True):
7          print(np.cos(x))
8
9  x = np.linspace(0, 2 * np.pi, 9)
10 print_cosine(x)
```

Note that `np.printoptions()` is only available in version 1.15 and later of Numpy. Running this example prints some numbers to the console:

Shell

```
$ python cosine.py
[ 1.     0.707  0.    -0.707 -1.    -0.707 -0.     0.707  1.   ]
```

The actual output of this example is not important. However, you should note that the argument `x` is annotated with `np.ndarray` on line 5, as we want to print the cosine of a full array of numbers.

You can run Mypy on this file as usual:

Shell

```
$ mypy cosine.py
cosine.py:3: error: No library stub file for module 'numpy'
cosine.py:3: note: (Stub files are from https://github.com/python/typeshed)
```

These warnings may not immediately make much sense to you, but you'll learn about stubs and typeshed soon. You can essentially read the warnings as Mypy saying that the Numpy package does not contain type hints.

In most cases, missing type hints in third-party packages is not something you want to be bothered with so you can silence these messages:

Shell

```
$ mypy --ignore-missing-imports cosine.py
Success: no issues found in 1 source file
```

If you use the `--ignore-missing-import` command line option, Mypy will not try to follow or warn about any missing imports. This might be a bit heavy-handed though, as it also ignores actual mistakes, like misspelling the name of a package.

Two less intrusive ways of handling third-party packages are using type comments or configuration files.

In a simple example as the one above, you can silence the `numpy` warning by adding a type comment to the line containing the import:

Python

```
3 import numpy as np  # type: ignore
```

The literal `# type: ignore` tells Mypy to ignore the import of Numpy.

If you have several files, it might be easier to keep track of which imports to ignore in a configuration file. Mypy reads a file called `mypy.ini` in the current directory if it is present. This configuration file must contain a section called `[mypy]` and may contain module specific sections of the form `[mypy-module]`.

The following configuration file will ignore that Numpy is missing type hints:

Config File

```
# mypy.ini

[mypy]

[mypy-numpy]
ignore_missing_imports = True
```

There are many options that can be specified in the configuration file. It is also possible to specify a global configuration file. See the documentation for more information.

Adding Stubs

Type hints are available for all the packages in the Python standard library. However, if you are using third-party packages you've already seen that the situation can be different.

The following example uses the Parse package to do simple text parsing. To follow along you should first install Parse:

Shell

```
$ pip install parse
```

Parse can be used to recognize simple patterns. Here is a small program that tries its best to figure out your name:

Python

```
1 # parse_name.py
```

```
3  import parse
4
5  def parse_name(text: str) -> str:
6      patterns = (
7          "my name is {name}",
8          "i'm {name}",
9          "i am {name}",
10         "call me {name}",
11         "{name}",
12     )
13     for pattern in patterns:
14         result = parse.parse(pattern, text)
15         if result:
16             return result["name"]
17     return ""
18
19 answer = input("What is your name? ")
20 name = parse_name(answer)
21 print(f"Hi {name}, nice to meet you!")
```

The main flow is defined in the last three lines: ask for your name, parse the answer, and print a greeting. The `parse` package is called on line 14 in order to try to find a name based on one of the patterns listed on lines 7-11.

The program can be used as follows:

Shell

```
$ python parse_name.py
What is your name? I am Geir Arne
Hi Geir Arne, nice to meet you!
```

Note that even though I answer `I am Geir Arne`, the program figures out that `I am` is not part of my name.

Let's add a small bug to the program, and see if Mypy is able to help us detect it. Change line 16 from `return result["name"]` to `return result`. This will return a `parse.Result` object instead of the string containing the name.

Next run Mypy on the program:

Shell

```
$ mypy parse_name.py
parse_name.py:3: error: Cannot find module named 'parse'
parse_name.py:3: note: (Perhaps setting MYPYPATH or using the
                       "--ignore-missing-imports" flag would help)
```

Mypy prints a similar error to the one you saw in the previous section: It doesn't know about the `parse` package. You could try to ignore the import:

Shell

```
$ mypy parse_name.py --ignore-missing-imports
Success: no issues found in 1 source file
```

Unfortunately, ignoring the import means that Mypy has no way of discovering the bug in our program. A better solution would be to add type hints to the Parse package itself. As Parse is open source you can actually add types to the source code and send a pull request.

Alternatively, you can add the types in a stub file. A stub file is a text file that contains the signatures of methods and functions, but not their implementations. Their main function is to add type hints to code that you for some reason can't change. To show how this works, we will add some stubs for the Parse package.

First of all, you should put all your stub files inside one common directory, and set the `MYPYPATH` environment variable to point to this directory. On Mac and Linux you can set `MYPYPATH` as follows:

Shell

```
$ export MYPYPATH=/home/gahjelle/python/stubs
```

You can set the variable permanently by adding the line to your `.bashrc` file. On Windows you can click the start menu and search for *environment variables* to set `MYPYPATH`.

Next, create a file inside your stubs directory that you call `parse.pyi`. It must be named for the package that you are adding type hints for, with a `.pyi` suffix. Leave this file empty for now. Then run Mypy again:

Shell

```
$ mypy parse_name.py
parse_name.py:14: error: Module has no attribute "parse"
```

If you have set everything up correctly, you should see this new error message. Mypy uses the new `parse.pyi` file to figure out which functions are available in the `parse` package. Since the stub file is empty, Mypy assumes that `parse.parse()` does not exist, and then gives the error you see above.

The following example does not add types for the whole `parse` package. Instead it shows the type hints you need to add in order for Mypy to type check your use of `parse.parse()`:

Python

```
# parse.pyi

from typing import Any, Mapping, Optional, Sequence, Tuple, Union

class Result:
    def __init__(
        self,
        fixed: Sequence[str],
        named: Mapping[str, str],
        spans: Mapping[int, Tuple[int, int]],
    ) -> None: ...
    def __getitem__(self, item: Union[int, str]) -> str: ...
    def __repr__(self) -> str: ...

def parse(
    format: str,
    string: str,
    evaluate_result: bool = ...,
    case_sensitive: bool = ...,
) -> Optional[Result]: ...
```

The ellipsis `...` are part of the file, and should be written exactly as above. The stub file should only contain type hints for variables, attributes, functions, and methods, so the implementations should be left out and replaced by `...` markers.

Finally Mypy is able to spot the bug we introduced:

Shell

```
$ mypy parse_name.py
parse_name.py:16: error: Incompatible return value type (got
                         "Result", expected "str")
```

This points straight to line 16 and the fact that we return a `Result` object and not the name string. Change `return result` back to `return result["name"]`, and run Mypy again to see that it's happy.

Typeshed

You've seen how to use stubs to add type hints without changing the source code itself. In the previous section we added some type hints to the third-party Parse package. Now, it wouldn't be very effective if everybody needs to create their own stubs files for all third-party packages they are using.

Typeshed is a Github repository that contains type hints for the Python standard library, as well as many third-party packages. Typeshed comes included with Mypy so if you are using a package that already has type hints defined in Typeshed, the type checking will just work.

You can also contribute type hints to Typeshed. Make sure to get the permission of the owner of the package first though, especially because they might be working on adding type hints into the source code itself—which is the preferred approach.

Other Static Type Checkers

In this tutorial, we have mainly focused on type checking using Mypy. However, there are other static type checkers in the Python ecosystem.

The **PyCharm** editor comes with its own type checker included. If you are using PyCharm to write your Python code, it will be automatically type checked.

Facebook has developed **Pyre**. One of its stated goals is to be fast and performant. While there are some differences, Pyre functions mostly similar to Mypy. See the documentation if you're interested in trying out Pyre.

Furthermore, Google has created **Pytype**. This type checker also works mostly the same as Mypy. In addition to checking annotated code, Pytype has some support for running type checks on unannotated code and even adding annotations to code automatically. See the quickstart document for more information.

Using Types at Runtime

As a final note, it's possible to use type hints also at runtime during execution of your Python program. Runtime type checking will probably never be natively supported in Python.

However, the type hints are available at runtime in the `__annotations__` dictionary, and you can use those to do type checks if you desire. Before you run off and write your own package for enforcing types, you should know that there are already several packages doing this for you. Have a look at **Enforce**, **Pydantic**, or **Pytypes** for some examples.

Another use of type hints is for translating your Python code to C and compiling it for optimization. The popular **Cython** project uses a hybrid C/Python language to write statically typed Python code. However, since version 0.27 Cython has also supported type annotations. Recently, the **Mypyc** project has become available. While not yet ready for general use, it can compile some type annotated Python code to C extensions.

Conclusion

Type hinting in Python is a very useful feature that you can happily live without. Type hints don't make you capable of writing any code you can't write without using type hints. Instead, using type hints makes it easier for you to reason about code, find subtle bugs, and maintain a clean architecture.

In this tutorial you have learned how type hinting works in Python, and how gradual typing makes type checks in Python more flexible than in many other languages. You've seen some of the pros and cons of using type hints, and how they can be added to code using annotations or type comments. Finally you saw many of the different types that Python supports, as well as how to perform static type checking.

There are many resources to learn more about static type checking in Python. PEP 483 and PEP 484 give a lot of background about how type checking is implemented in Python. The Mypy documentation has a great reference section detailing all the different types available.

your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# In Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Geir Arne Hjelle**

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

» More about Geir Arne

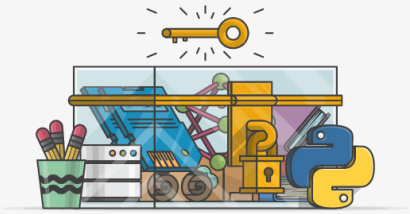
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Brad

 Joanna

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate

Recommended Video Course: Python Type Checking