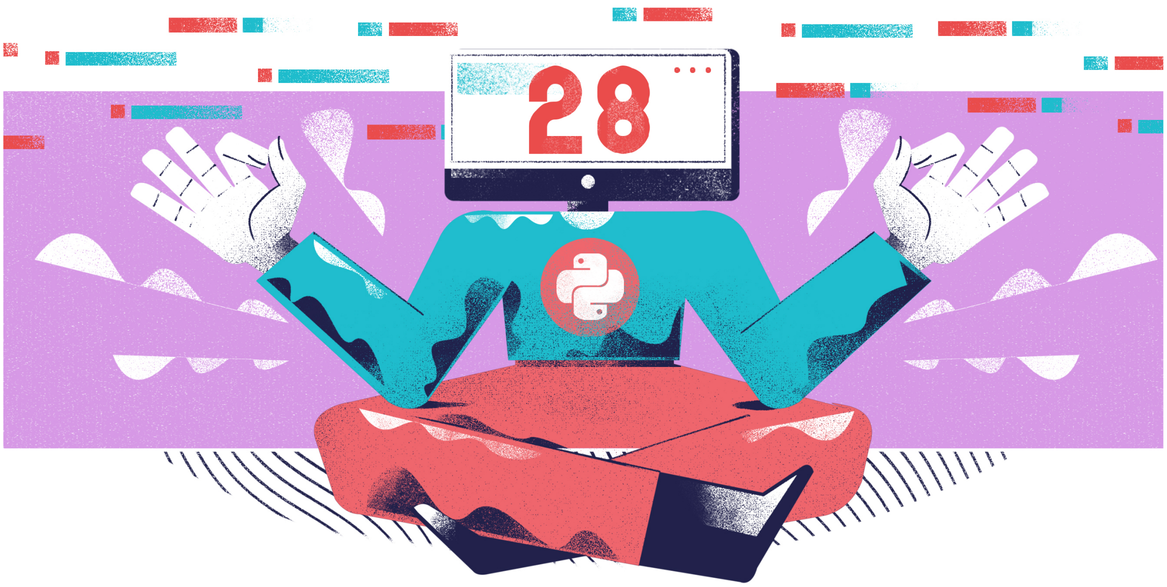
WORKING ON YOUR OWN

Day 28: Type Hinting



Welcome to day 28 of the 30 Days of Python series! Today we're going to be learning about type hinting in Python using type annotations.

Type annotations allow us to tell Python what we expect to be assigned to names at given points in our application. We can then use these annotations to check the program is in fact doing what we intend.

The benefits of type hinting

Type hinting can be very helpful for a number of reasons.

1. It makes it easier to understand our code, because our helpful variable names are now accompanied by a description of the sort

of data we expect to be assigned to them.

2. Most modern editors are able to make good use of our type annotations to provide more meaningful hints when we do things like calling functions.
3. We can use tools like `mypy` to check that our type annotations are being honoured, helping us catch bugs caused by passing around incorrect types.

One thing type annotations cannot do is actually prevent us from breaking the rules we outlined in our annotations. They're a development tool only, and don't have any effect on our code when we run the application.

Now that we have an idea about what type annotations can do for us, let's learn how to actually use them.

Installing `mypy`

If you're using an editor like PyCharm, type hinting is built in and ready to go. PyCharm will give you live hints while writing your code, so you don't need to do anything extra.

If your editor doesn't support this kind of live type hinting, or you want to be able to run a tool to give you comprehensive information about a given file, we need to install a tool like `mypy` .

We can install `mypy` by running the following command:

```
python -m pip install mypy
```

If you're not sure how to use this, there's [a tutorial in the Python documentation]() that talks about installing packages using the command line.

Once `mypy` has been installed, we can use it by writing `mypy` followed

Once `mypy` has been installed, we can use it by writing `mypy` followed by the name of the file we want to check. For example, if we wanted to check `app.py`, we'd write the following in the console:

```
mypy app.py
```

This will inform us of any problems that `mypy` was able to detect while evaluating this file.

Basic type hinting

Let's start by using type hinting to annotate some variables that we expect to take basic types.

```
name: str = "Phil"
age: int = 29
height_metres: float = 1.87
loves_python: bool = True
```

As you can see from the examples above, we can annotate a variable by adding a colon after the variable name, after which we specify the type.

Here I've indicated that `name` is a string, `age` is an integer, and `height_metres` should be a float. In this case, all of our type annotations align with the values that have been assigned, so we don't have any issues.

We can confirm this by running `mypy`.

```
> mypy app.py
Success: no issues found in 1 source file
```

Now let's look at what happens when things are not what we intended.

```
name: str = 29
age: int = 1.87
height_metres: float = "Phil"
```

All of our values have gotten jumbled up. Maybe we assigned these values from an iterable that contained the values in the wrong order.

Now if we run `mypy`, we get some errors.

```
> mypy app.py
app.py:1: error: Incompatible types in assignment (expression
has type "int", variable has type "str")
app.py:2: error: Incompatible types in assignment (expression
has type "float", variable has type "int")
app.py:3: error: Incompatible types in assignment (expression
has type "str", variable has type "float")
Found 3 errors in 1 file (checked 1 source file)
```

The errors are actually very helpful. They explain exactly what went wrong, and at the start of each line we get a reference to the file where the error happened, along with a line number.

Using this information, we can easily track down the source of this issue.

Adding some flexibility

Let's say we want to be a little more flexible in how we handle `height_metres`. I only really care that it's a real number, so instead of accepting just floats, I want to also accept integers.

The way we accomplish this is by using a tool called `Union` which we have to import from the `typing` module.

```
from typing import Union
from typing import Union


name: str = "Phil"
age: int = 29
height_metres: Union[int, float] = 1.87
```

Here I've added `Union[int, float]` as a type annotation for `height_metres`, which means we can accept either integers or floats. We can add as many types as we like to this `Union` by adding more comma separated values between the square brackets.

We can also get super flexible and use another tool called `Any`, which matched any type. You should be careful about using `Any`, because it largely removed the benefits of type hinting. It can be useful for indicating to readers that something is entirely generic though.

We can use `Any` like any of the other types:

```
from typing import Any

name: str = "Phil"
age: int = 29
height_metres: Any = 1.87
```

Annotating collections

Now that we've looked at annotating basic types, let's talk about how we might annotate that something should be a list, or maybe a tuple containing values of a specific type.

In order to annotate collections, we have to import special types from the `typing` module. For lists we need to import `List` and for tuples we need to import `Tuple`.

As you can see, the names make a lot of sense, but if you're ever in doubt about what you need to import, you can find the documentation for the `typing` module here.

Here is an example of a variable using a `List` annotation:

```python
from typing import List

names: List =  ["Rick", "Morty", "Summer", "Beth", "Jerry"]
```

If we wanted to specify which types should in the list, we can add a set of square brackets, much like we did with `Union` .

```python
from typing import List

names: List[str] =  ["Rick", "Morty", "Summer", "Beth", "Jerry"]
```

Leaving the brackets off is really the same thing as writing `List[Any]` .

If we want, we can allow a variety of types in a list by combining `List` and `Union` like this:

```python
from typing import List, Union

random_values: List[Union[str, int]] =  ["x", 13, "camel", 0]
```

When working with tuples, we can specify a type for each item in sequence, since tuples are of fixed length and are immutable.

For example, we can do something like this:

```python
from typing import Tuple

movie: Tuple[str, str, int] = ("Toy Story 3", "Lee Unkrich", 2010)
```

Note

In Python 3.9 we won't have to import things like `Tuple`, `List`, and `Dict` from the `typing` module. Instead, we'll be able to use the standard `tuple`, `list`, and `dict` types for annotation.

You can find out more here.

Creating type aliases

Let's consider a case where we want to store lots of movies like the one above in a list. We ran into this case several times in the early stages of the course. How would we annotate something like that?

Maybe something like this:

```
from typing import List, Tuple

movies: List[Tuple[str, str, int]] = [
        ("Finding Nemo", "Andrew Stanton", 2005),
        ("Inside Out", "Pete Docter", 2015),
        ("Toy Story 3", "Lee Unkrich", 2010)
]
```

This *does* work, but that type annotation is getting very hard to read. That's a problem, because one of the benefits of using type annotations is to *help* with readability.

In cases like this where we have complex type annotations, it's often better to define new aliases for certain type combinations. For example, I think it makes a lot of sense to call each of these tuples a `Movie`.

We can do this like so:

```
from typing import List, Tuple

Movie = Tuple[str, str, int]
Movie = Tuple[str, str, int]

movies: List[Movie] = [
        ("Finding Nemo", "Andrew Stanton", 2005),
        ("Inside Out", "Pete Docter", 2015),
        ("Toy Story 3", "Lee Unkrich", 2010)
]
```

Now tools like `mypy` are going to consider the term `Movie` as meaning `Tuple[str, str, int]`. If we try checking our code with `mypy`, we can see everything works just fine.

```
> mypy app.py
Success: no issues found in 1 source file
```

Now let's make a small change, just so we can assure ourselves that we're not getting false positives. To check, I'm going to change the date of Finding Nemo to the string, `"2005"`.

When we run `mypy`, we now can an error, just as we expected.

```
> mypy app.py
app.py:11: error: List item 0 has incompatible type "Tuple[st
r, str, str]"; expected "Tuple[str, str, int]"
Found 1 error in 1 file (checked 1 source file)
```

Annotating functions

Now let's get into annotating functions, which is where this kind of tool is most useful. Let's stick with our list of movie tuples for now, and let's add a function to print each movie in a given format.

```
from typing import List, Tuple

Movie = Tuple[str, str, int]
```

```
def show_movies(movies):
        for title, director, year in movies:
                print(f"{title} ({year}), by {director}")

movies: List[Movie] = [
        ("Finding Nemo", "Andrew Stanton", 2005),
        ("Inside Out", "Pete Docter", 2015),
        ("Toy Story 3", "Lee Unkrich", 2010)
]

show_movies(movies)
```

So, how do we annotate this function?

Annotating parameters is just like annotating any other variable. In this case we're passing in a list of movie tuples, and we already have a `Movie` annotation ready to go, so we can just write this:

```
from typing import List, Tuple

Movie = Tuple[str, str, int]

def show_movies(movies: List[Movie]):
        for title, director, year in movies:
                print(f"{title} ({year}), by {director}")

movies: List[Movie] = [
        ("Finding Nemo", "Andrew Stanton", 2005),
        ("Inside Out", "Pete Docter", 2015),
        ("Toy Story 3", "Lee Unkrich", 2010)
]

show_movies(movies)
```

Annotating return values

Let's add another couple of functions to our little application. I want to add a search function to determine if a given movie exists in the movie library, and I also want to add a function to handle the printing of a single movie.

My implementation is going to look something like this:

```
from typing import List, Tuple

Movie = Tuple[str, str, int]

def find_movie(search_term, movies):
        for title, director, year in movies:
                if title == search_term:
                        return (title, director, year)

def show_movies(movies: List[Movie]):
        for movie in movies:
                print_movie(movie)

def print_movie(movie):
        title, director, year = movie
        print(f"{title} ({year}), by {director}")

movies: List[Movie] = [
        ("Finding Nemo", "Andrew Stanton", 2005),
        ("Inside Out", "Pete Docter", 2015),
        ("Toy Story 3", "Lee Unkrich", 2010)
]

show_movies(movies)

search_result = find_movie("Finding Nemo", movies)

if search_result:
        print_movie(search_result)
else:
        print("Couldn't find movie.")
```

The `find_movie` function is relatively crude. It returns only perfect matches, and it can find only a single result, but it will do for his example.

The `print_movie` function is now dealing with a small part of what the original `show_movies` function was doing. Defining this second function allows us to print individual movies in other parts of our application, like when we get a movie back from `search_result` .

Let's start by annotating all the things we already know how to do.

```
from typing import List, Tuple, Union

Movie = Tuple[str, str, int]

def find_movie(search_term: str, movies: List[Movie]):
        for title, director, year in movies:
                if title == search_term:
                        return (title, director, year)

def show_movies(movies: List[Movie]):
        for movie in movies:
                print_movie(movie)

def print_movie(movie: Movie):
        title, director, year = movie
        print(f"{title} ({year}), by {director}")

movies: List[Movie] = [
        ("Finding Nemo", "Andrew Stanton", 2005),
        ("Inside Out", "Pete Docter", 2015),
        ("Toy Story 3", "Lee Unkrich", 2010)
]

show_movies(movies)

search_result: Union[Movie, None] = find_movie("Finding Nemo"
, movies)

if search_result:
        print_movie(search_result)
else:
        print("Couldn't find movie.")
```

Run this through `mypy` a few times with different values and make sure everything still works.

Now that we have the majority of the app type annotated, there's one thing we're missing. We're not currently telling Python what our functions should return.

We can do this by using `->` after the parentheses when defining our functions.

Most of our functions don't need a return annotation, because they implicitly return. However, our `find_movie` function does, and it can return two different values: a movie tuple, or `None`. It returns `None` in the case where no matching movie was found.

We can therefore annotate it like this:

```
def find_movie(search_term: str, movies: List[Movie]) -> Unio
n[Movie, None]:
        for title, director, year in movies:
                if title == search_term:
                        return (title, director, year)
```

We have a problem though. If we run `mypy`, it complains that we're missing a `return` statement.

```
> mypy app.py
app.py:5: error: Missing return statement
Found 1 error in 1 file (checked 1 source file)
```

This is `mypy` warning us that we've done something not in keeping with the Python style guide (PEP8). To quote the guide,

Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable)

This means we should be writing our function like this:

```
def find_movie(search_term: str, movies: List[Movie]) -> Unio
n[Movie, None]:
        for title, director, year in movies:
                if title == search_term:
                        return (title, director, year)

        return None
```

Now everything passes without issue:

```
> mypy app.py
Success: no issues found in 1 source file
```

Using `Optional`

In cases where have something like `Union[Movie, None]` , where one of the types in a `Union` is `None` , we have another tool we can use called `Optional` . If we write something like `Optional[Movie]` , this is the same thing as writing `Union[Movie, None]` .

```
def find_movie(search_term: str, movies: List[Movie]) -> Opti
```

```
  onal[Movie]:
          for title, director, year in movies:
                  if title == search_term:
                          return (title, director, year)

          return None
```

Exercises

There's only one exercise today, because it's a big one. Take your final solution to the day 14 project (easy or hard version) and implement type hinting for your code.

If you're unsure of how to do something, remember that you can always look at the `typing` module [documentation](https://docs.python.org/3/library/typing.html).

You can find our solution [here]().

Project

Today is another project day, so once you're done with today's exercise, make sure to check out [today's project]()!

Today we're going to be writing a program to automatically gather information from pages on the Internet.