Python Type Checking Guide Documentation

Release 1.0

Chad Dombrova

Mar 03, 2017

Contents

1	Background		
	1.1 Why Should I Care?	. 3	
	1.2 Enter PEP 484	. 4	
2	The Tools	5	
	2.1 mypy	. 5	
	2.2 PyCharm	. 6	
	2.3 pytype	. 7	
	2.4 Comparison	. 7	
3	Type Classes	9	
	3.1 Foundational Types	. 9	
	3.2 Concrete Collection Types	. 12	
	3.3 NamedTuple	. 12	
	3.4 Ordinary Classes	. 13	
4	Indices and tables	15	

Contents:

Background

Why Should I Care?

You may have heard about type-hints being added to python 3.5, and wondered "why you should I care?". Well, the answer is much the same as for documenting your code: type checking saves you time by preventing mistakes and removing guesswork. Up until now, the best that we had in terms of type specifications was a handful of conventions which were ambiguous at best, and now we have a standard to build tools around.

Take the numpy docstring convention as an example. They kindly give us some basic typing examples, such as this:

```
Parameters
-----
filename : str
copy : bool
dtype : data-type
iterable : iterable object
shape : int or tuple of int
files : list of str
```

But there's no guidance on how to combine these into more complex recipes. As a result, I often see ambiguous type specifications that look like this:

```
list of str or int
```

Is the int *in* the list or *out* of it?

Moreover, there are no examples of how to handle complex tuples, dictionary key and value types, callable signatures, or how to specify types which are classes rather than instances.

The upshot is that with this much ambiguity, programmatic type checking is pretty unreliable. To improve the situation, some IDEs like PyCharm have proposed their own convention which is a step forward, but do you really want to make your modules IDE-specific?

Enter PEP 484

With PEP 484, Guido and crew have figured all this out for you and created a standard for type annotations which is now part of python 3.5. If you're using python 3.5 or greater, you can write function definitions like this:

Neat. But what about those of us still stuck on 2.x?

This is where we need to stop and clarify the difference between type annotations and type checking. With the addition of pep484, python 3.5 gained two things:

- 1. a standard for describing types (e.g. Union[str, List[str]].)
- 2. syntax support for annotating function arguments and return values with type descriptions

Noticeably lacking here is actual *type checking*, i.e. inspection of code to enforce that arguments and assignments match their declared types. The developers of python left that role be filled by third-party tools.

Back to python 2.x: A standard for describing types in unambiguous terms is a big deal even without the syntax support added in 3.5, and the good news is that the means for creating these type definitions, the typing module, is available for python 2.7. However, the lack of syntactical support means that the type-checkers must provide their own conventions for associating type descriptions with arguments and return values in python 2.7 code.

So, without further ado, let's get to the tools.

The Tools

There are a handful of tools for performing pep484-compatible type checking. Each has its pros and cons. Both perform static code analysis, which means they are not actually importing and running your modules: instead they parse and analyze your code. This is safer, but it means that dynamically generated objects can not be inspected.

mypy

mypy is a command-line tool much like a linter that scans your code and prints out errors. The developers of mypy are leading the charge on type-checking. PEP 484 was originally inspired by mypy and Guido himself is now currently involved in its development.

Below are your options support by mypy for adding type annotations to functions in python 2.7 (more info here):

Single-line:

```
def doit(inputs, enabled):
    # (Union[str, List[str]], Dict[str, bool]) -> Iterable[str]
    "Do something with those inputs"
    ...
```

The bummer with this is it can get very long, and it's hard to visually associate the argument with the type.

Multi-line:

```
def doit(inputs, # type: Union[str, List[str]]
        enabled # type: Dict[str, bool]
    ):
    # type: (...) -> Iterable[str]
    "Do something with those inputs"
    ...
```

A bit more verbose, but more legible.

One aspect of mypy which may make it difficult for you to integrate into your build/release cycle is that python 3.5+ is required to run it, even if you're analyzing python 2.7 code.

PyCharm

PyCharm is my new favorite IDE. Its code analysis goes deep, and it saves my ass daily. Now that I've been making it aware of my argument and return types, it's basically SkyNet (or will be, with just a few more upgrades...).

As of this writing, PyCharm supports both the single-line and multi-line styles above, as well as PEP484-compatible types delivered via docstrings. For the latter you have to be pretty anal about your formatting: If you're too loosey-goosey the parser will give up. PyCharm can parse four styles of docstrings.

reStructureText style

```
def doit(inputs, enabled):
    """Do something with those inputs
    :param inputs: input names
    :type inputs: Union[str, List[str]]
    :param enabled: mapping of input names to enabled status
    :type enabled: Dict[str, bool]
    :rtype: Iterable[str]
    """
    ...
```

Ugly, but gets the job done. Epydoc-style docstrings are the same but with an @ instead of the leading :.

google style

```
def doit(inputs, enabled):
    """Do something with those inputs
Args:
    inputs (Union[str, List[str]]): input names
    enabled (Dict[str, bool]): mapping of input names to
        enabled status
Returns:
    Iterable[str]: enabled inputs
"""
...
```

Compact, but legible.

numpy style

```
def doit(inputs, enabled):
    """Do something with those inputs
    Parameters
    -------
    inputs : Union[str, List[str]]
        input names
    enabled: Dict[str, bool]
        mapping of input names to enabled status
    Returns
    -------
    Iterable[str]
        enabled inputs
    """
    ...
```

My personal favorite.

The main downside with PyCharm for PEP484-style type-checking is that it's still playing catchup with mypy. Some pretty fundamental features are still missing:

- Type
- Type aliases
- TypeVar
- Generics

Plus, I'd love to see more visual feedback

If nothing else comes from writing this, it will be worth it if a few people click on the links above and make some noise on those issues.

pytype

I'm including pytype from Google for the sake of completeness. It's a command-line tool like mypy. The main thing it has going for it is that it can be run using python 2.7, unlike mypy which can only be run using python 3.5+ (both tools can analyze python 3.x code).

Comparison

PyCharm gives you near instant feedback about type incompatibilities in the context of your code, which creates an addictive feedback loop that encourages ever more type-hinting. Mypy on the other hand is a bit of a pain. You have to run it manually, then dig through its cryptic output and look up corresponding line numbers. It's really meant to be integrated into your build/release process.

I also really like that PyCharm let's me continue to specify types within docstrings. For existing code, basic types are already working within PyCharm, so I just need to upgrade the more exotic recipes to the new standard. Also, I prefer to have type info adjacent to the description of the type.

The main downside of PyCharm is that it is not as thorough as mypy and there are still a number of extremely important features that are not implemented at this moment, though I have confidence that it will improve in the short term. mypy is also capable of statically typing individual variables not just function arguments and returns.

There's nothing stopping you from using both in tandem – PyCharm as the immediate first line of defense and mypy as a more thorough check run by continuous integration.

Type Classes

The first thing to understand is that type annotations are actual python classes. You must import them from typing to use them. This is admittedly a bit of a nuisance, but it makes more sense when you consider that the syntax integration in python 3.5 means you're attaching objects to function definitions just as you do when providing a default value to an argument. In fact, you can use typing.get_type_hints() function to inspect type hint objects on a function at runtime, just as you would inspect argument defaults with inspect.getargspec().

Type classes fall into several categories, which we'll review below.

Foundational Types

The core set of types is pretty well covered in the mypy docs, but I'll give a brief overview below.

Any

Represents any type.

If a function returns None you should specify this explicitly, because if omitted it defaults to Any which is more permissive.

Unlike Any, object is an ordinary static type, and only operations valid for all types are accepted for object values.

Any is thus more permissive.

Callable

Used to denote a function or bound method with a particular signature.

Here's a simple function and how to encode that as a type annotation:

```
def repeat(s, count):
    # type: (str, int) -> str
    return s * count
```

Callable[[str, int], str]

Or, if you only care about the return result:

Callable[..., str]

Union

Used when there is more than one valid type.

```
Union[str, List[str]]
```

Optional

Shorthand for a type which is allowed to be None.

These are equivalent:

Optional[int]
Union[int, None]

In mypy None is by default a valid value for every type, but due to popular demand that is going to change, though I'm not sure in what time frame. It's already possible to change the behavior of the type-checker using a flag. Thus, if you're getting started now, its best to get in the habit of adding the Optional type modifier to denote a type that includes None.

Туре

Used to denote that a type should be an uninstantiated class.

```
Type[MyClass]
Type[Union[MyClass, OtherClass]]
```

Type Aliases

This is a technique rather than a type. Remember how we discussed that type definitions are regular python objects? Well, that means you can assign them to module-level variables and use these variables in your annotations. This is handy if you have a lot of functions that take the same complex recipe.

(broken in pycharm)

```
from typing import Dict, List, Union
PropertiesType = Dict[str, List[str]]
PropertiesListType = List[Dict[str, PropertiesType]]

def process_properties(props):
    # type: (PropertiesListType) -> None
    ...
```

Generic

This is the base class for all the collection classes covered below. It's what gives them the bracket syntax for typespecialization (e.g. Container[int]). My epiphany with type-hinting came when I realized that subclasses of Generic are not just for defining type-hints. By using Generic as an alternative base class to object when creating your own collection classes, your classes can be used both as a collection (by instantiating it as you normally would) and as a type annotation (by using [] on the class itself). Check out the Stack example in the mypy docs to see an example.

(broken in pycharm)

TypeVar

TypeVar lets you create relationships and restrictions between an argument and other arguments or return values.

For example, let's say that you have a function which takes a value of any type, and returns a value of the same type.

If we use Any then we fail to make that relationship:

```
def passthrough(input):
    # type: (Any) -> Any
    return input
```

Both input and result may be any type, but there's nothing to indicate that they will always be the same type as each other.

To give the type checker more context, we create a TypeVar and share it between annotations.

```
T = TypeVar('T')
def passthrough(input):
    # type: (T) -> T
    return input
```

This is called a generic function. Of course, it gets more interesting than this. A TypeVar can be restricted in the same way as any other value:

TypeVar('T', bound=Callable[[int, str], bool])

TypeVars are often used with Generic collections (discussed more below) to form a relationship between the collection and another argument or return values. Here's a solid example from the docs on generics:

```
from typing import TypeVar, Sequence
T = TypeVar('T')
def first(seq: Sequence[T]) -> T:
    return seq[0]
```

Concrete Collection Types

The concrete collection types are intended to be used as stand-ins for certain key collections for the purpose of typehinting. They cannot be instantiated: For that, you need to continue to use their "real" counterparts.

In an ideal world, all of the collections in python's standard library would subclass from Generic, which would allow the same class to serve as both implementation and type annotation. Perhaps if type hinting takes off this will be addressed one day, in the meantime we have this split.

The concrete collection types:

- Tuple
- Dict
- DefaultDict
- List
- Set

These are pretty straight-forward to use. You can glean all you need from a few simple examples:

Example	Explanation
list	list of any type, possibly heterogeneous
List[Any]	same as above
List[int]	list containing only integers
dict	dictionary with any key or value
Dict[Any, Any]	same as above
Dict[str, int]	dictionary whose keys are strings and values are integers
tuple	tuple with any quantity of any type
Tuple[Any,]	same as above
Tuple[int]	tuple with a single integer. ex: (1,)
Tuple[int,]	tuple with any number of int
Tuple[int, str]	tuple whose first element is an integer and second is a string

NamedTuple

typing.NamedTuple is an alternative to collections.namedtuple that supports type-checking.

Under the hood it wraps collections.namedtuple and tags the resulting class with an attribute to track the field types, but in reality, that's not even necessary as the static code analysis won't have access to it.

Here's an example adapted from the docs. The Point class defined in the following code is opaque to type-checking:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(x=1, y='x')
p.y / 2.0 # fails at runtime
```

By swapping it with typing.NamedTuple, the Point class can now be used as a type annotation in functions and instantiation of the type can be properly validated.

```
from typing import NamedTuple
Point = NamedTuple('Point', [('x', int), ('y', int)])
```

```
p = Point(x=1, y='x') # issue detected by mypy
p.y / 2.0
```

Ordinary Classes

As you might expect, any class can be used as a type identifier. This restricts objects to instances of this class and its subclasses.

The two tools - mypy and PyCharm - differ in how they find objects specified in type annotations.

With mypy, the name given *must* be a valid identifier for that object in the current module. For example, this works:

```
import zipfile
def zipit(arg):
    # type: (zipfile.ZipFile) -> None
    return
```

But this does not:

```
import zipfile

def zipit(arg):
    # type: (ZipFile) -> None
    return
```

This is because <code>ZipFile</code> does not identify any object at the scope of the <code>zipit</code> function (to be honest, I'm actually not entirely sure how the scoping works in mypy, but it has a module scope for sure). This behavior makes sense if you think of the type-comments as placeholders for the python 3.5 syntax additions. Again, it helps to think of type hints the same way that you would default arguments. In that light, I think it's intuitive that it would not work without first importing <code>zipfile</code>:

```
def zipit(arg: zipfile.ZipFile) -> None:
    return
```

This rule actually applies to any object defined externally to a comment-based type annotation, such as type aliases, but it comes into play most often with custom classes.

PyCharm is a bit more forgiving than mypy. If prefix your object with a dotted module or package name, it will find the object within that module, assuming your project search paths are setup correctly. Of course, if you plan to use both tools in conjunction, you'll have to shoot for the lowest common denominator, which is mypy.

Indices and tables

- genindex
- modindex
- search