Python News

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. pcnorton@rbox.co.

In this column, I'm covering a bit of Python news, with some info about type checking in more depth.

Time for Python 3

For years now it's been made very clear that Python 2 is coming to the end of support—this was put in writing in PEP 373 [1] . The original date was pushed back to 2020 to give everyone some more time to move to Python 3. Most projects that are still actively maintained have put in the effort to support Python 3, and Python 3 hasn't been standing still. It's adding features like async support and syntax for supporting static type checking (more on this in a moment) that makes it a more modern language than Python 2.

To put the cherry on top, now that it's almost 2020, developers of some prominent Python projects have announced that they're going to discontinue support for Python 2 in future release of their project. In case your projects could be affected by this, go take a look at the projects listed at the Python 3 Statement website (https://python3statement.org/). Many fundamental projects have decided that after performing the work to be compatible with both Python 2 and Python 3 for some time (years and years in some cases), they want to reduce their workload by just supporting Python 3. This seems only fair. Python 2 has had an extraordinary lifetime, and now it's time to retire it with grace. I encourage you to take a look at python3statement.org and to understand if the projects you rely on directly or indirectly will impact your work, and to plan accordingly.

Type Hints in Python 3

I'm in a situation shared by many of my peers where we're still planning our transition to Python 3 for most of our infrastructure code. As part of getting our stories together for upgrading, I'm thinking about the fun stuff that has been created as Python 2 has gone stale.

So I'd like to take a look at one of these cool features I'm anxious to put to good use: static type checking. Static type checking in Python makes it possible for a process that reads code to check that all types passed into a function, and all return values from the function, are appropriate, and alert the developer to deviations that would cause bugs. By using static type checking, you can eliminate a lot of bugs without ever having to run the program. In the way Python implements this, the static checker is an external process—it's not Python itself that checks it before running. So whether or not you use this feature, your code will still run.

Some languages have incorporated static type checking from the outset, but this is not how Python was developed. Historically, Python is among the languages that is dynamically runtime type-checked, which means that it's common to have crashes when there is a severe enough type mismatch. Because of the way that static type checking is being added to Python late in its development, its power to catch problems has been limited by speed of development of the type checking tools, and the rate of adoption and use of those tools. The tools are actually being developed at a really fantastic pace, but many libraries and other code bases can only adopt type checking as they move to recent Python 3 versions.

We'll look at the basic idea of static type checking in Python and at a cool feature that could be added. Unfortunately, this column is not going to be able to cover Python type-checking features in depth. For that there is a lot of excellent documentation written on how static type checking can be used in recent versions of Python when you're ready to use it—the official documentation is thorough and very deep. So that's not what I'm going to write about here.

The basic observation that makes static type checking attractive is that as a Python programmer you know that the following code will run:

```
def badlen(container):
    return len(container)
```

but you also know that the built-in `len()` is only useful on certain types. You probably also know that objects of those types have the *dunder* (double underscore) method `__len__()` to provide their length. And you also know that invoking the `len()` built-in function on an inappropriate type causes a runtime TypeError:

```
>>> badlen(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in badlen
TypeError: object of type 'int' has no len()
```

People who come from static (type-checking-wise) languages often look at Python and its peers and ask why this is acceptable, when clearly other languages can catch this sort of error before they are ever run. The obvious answer is that part of what Python provides is a very dynamic programming environment where a lot of the knowledge required for static checking is not possible. The success of Python makes it clear that static type checking isn't the most important feature to make the language usable and productive.

Since compile-time type checking promises to reduce or eliminate this class of error, it would make Python better to have it, so a lot of work went into discussing what would be necessary to add it without causing any extra work for people who won't use it while bringing benefits to people who do want it.

The guiding principle behind Python development has long been that, to the extent possible, Python should try to advance with backwards compatibility in mind. To this end, a bit of syntax was created and specified which allowed function annotations in PEP 3107 [2]. With that in place, PEP 484 was hashed out; it introduced a standard for type "hints" using the PEP 3107 annotations. Although allowing us as developers to communicate what the type is, the interpreter in effect completely ignores all of this information at runtime. Instead, its purpose is to allow tooling to be built to verify that annotated functions comply with the types that are described. With these checkers in place, even better tests can be created.

So with the introduction of PEP 484 and a common standard for type hinting, tools can be built ensuring that functions are using type hints to get the right input types and therefore are returning the right output types.

This might seem like a small refinement of a very popular language; after all, Python isn't the only language that has succeeded by growing its user base without static type checking. However, statically checkable, and therefore avoidable, type errors are a very common source of bugs, so in the long term, opting into this is likely to be a huge benefit to those who use it.

So what do type hints look like? They can change the declaration of a variable in a function call, for example, from `variable_name` to `variable_name: type`, like this:

```
def betterlen(container: list):
    return len(container)
```

That tells the type checker that the function takes a list. Usually lists are of a particular type, though, and we can ask the type checker to check for an appropriate type of list, or we can make it clear that we're not concerned about the type of list. This is normal Python behavior. To make this possible, there is the `typing` module, which provides definitions of objects that the type checker can use to allow you to declare how thoroughly you want to check your lists, dictionaries, or other container types.

To enable this, you include the `typing` module in your code and import type specifications, which provide the specificity for the structure and types of the things they contain. In this case, we're going to start with a specification of `Any`, which explicitly says "accept that this list can contain elements of anything, it's fine." But this could also be used to be more specific about only particular built-in types or user-defined types. It looks like this:

```
from typing import Any, List

def goodlen(container: List[Any]):
    return len(container)
```

By invoking a static analyzer (`mypy` in this case) on a chunk of code with a type mismatch, like this:

```
goodlen(7)
```

it can describe the problem it sees without actually running the code:

```
$ mypy simply_doesnt_work.py
simply_doesnt_work.py:8: error: Argument 1 to "goodlen" has
incompatible type "int"; expected "List[Any]"
```

Whee! That was pretty easy. For a basic introduction, the next step is to go one more step and specify the return value, which we haven't done yet. We want to specify the return type, too, because the current state of the `goodlen()` function creates a dead-end for the type checker. Because the return type isn't declared, the type checker graph bottoms out and can't do further checking at this point.

So to help the checker, you can add a return type simply by adding a `-> type`. For a length, we'll always be returning an integer; a simple case looks like this:

```
def betterlen(container: List[Any]) -> int:
    return len(container)
```

The more annotations that are added to a code base, the more automatically simple but critical mistakes can be avoided before your code is ever run.

By itself, this has benefits for unit and integration tests. You can just start adding harmless annotations, and start to check whether your libraries, dependencies, etc. are doing the right thing.

But there's another very interesting thing that is possible, which, hopefully, Python will adopt in the future. It's presently available in Rust, so let's use that as the example.

You may have heard of Rust, the language, since it's received a lot of attention since it hit 1.0 in 2015. In case you haven't had a chance to look into it, I think it's fair to say that its goal is to be a language that can achieve the performance and control of C or C++, while providing the memory safety of a garbage-collected language like Java, Python, or Go. In addition, Rust also eliminates other risks present in most other mainstream programming languages.

As part of providing this attractive sounding set of goals, Rust includes strong compile-time type checking as a fundamental feature. Rust also incorporates a very interesting idea: exhaustive checking of all possibilities in a match (as I understand it, this originated in the ML languages). This is needed because a lot of bugs are created when a series of conditional statements—e.g., in Python an `if… elif… else`—is produced that due to oversight, or changes in the set of possible choices, ends up not covering all of the possibilities.

To make this work, Rust uses a clever trick. The implementation of this clever trick is the `match` expression, which is like a `case` or a `switch` in other languages. But instead of being just another way of writing if...else if...else if...else, it makes sure that when a match is invoked, it can identify that all possible matches have been covered. So if the type being matched is an unsigned 32-bit integer, then the compiler knows that if you haven't specified either all numbers from 0 to 2^{32}-1 or used a default match (Rust does this with the underscore target in a match—this is the equivalent of an `else` in Python), then you have left possible values which haven't been accounted for, and it will refuse to let that code compile or run.

Another clever extension is combining this with `enums`, or an enumerated set of possible values that are declared up-front. With an `enum`, the compiler knows whether or not all possible arms of the possible matches with enum values have been checked, because the enum can only have a fixed number of possibilities. A quick example of what this could look like in Rust is:

```
enum BreadSpreads {
    Butter,
    Margarine,
    CreamCheese,
    Nutella
}

fn breakfast_bread(spread: BreadSpreads) {
    println!("Breakfast bread with {}",
        match spread {
            BreadSpreads::Butter => "butter",
            BreadSpreads::Margarine => "margarine",
            BreadSpreads::CreamCheese => "cream cheese",
            BreadSpreads::Nutella => "nutella"
        }
    )
}

fn main() {
    let butter_spread = BreadSpreads::Butter;
    let margarine = BreadSpreads::Margarine;
    breakfast_bread(butter_spread);
    breakfast_bread(margarine);
}
```

This is very straightforward and not particularly noteworthy when it is working. What is more interesting is that if you change the `breakfast_bread` function by removing any of the arms of the match (let's use `Margerine` for this example), the compiler will refuse to compile it. It will tell you that the code is broken and save you from having to discover the problem in production:

```
$ cargo build
   Compiling breadspread v0.1.0 (/home/spacey/dvcs/pcn/login/
2019-6/breadspread)
error[E0004]: non-exhaustive patterns: 'Margerine' not covered
  --> src/main.rs:10:15
   |
1  | / enum BreadSpreads {
2  | |     Butter,
3  | |     Margerine,
   | |     --------- not covered
```

```
4  | |    CreamCheese,
5  | |    Nutella
6  | | }
   | |_- 'BreadSpreads' defined here
...
10 |         match spread {
   |             ^^^^^^ pattern 'Margerine' not covered
   |
   = help: ensure that all possible cases are being handled,
possibly by adding wildcards or more match arms

error: aborting due to previous error

For more information about this error, try 'rustc --explain
E0004'.
error: Could not compile 'breadspread'.
```

This feature of the Rust compiler works because the set of possible enums can't change once they've been declared. Of course, being able to change that after runtime would break guarantees that Rust provides with this little trick. So generally, the compiler looks at the match to make sure that you have accommodated each possible variation that the enum could take, because as an enum those possibilities are, well, enumerated in the code. In addition, as with most case/switch/if...then...else constructs, you have the equivalent of an else clause, so this need for an exhaustive match doesn't require you to write out a match for every possible case individually. It just requires that you don't leave off the equivalent of the else clause and leave cases uncovered. It doesn't protect the programmer from every mistake, but it prevents cases from being missed.

So it's interesting to ask, would this be possible in Python and how much would it help? And what would it look like if it was being used? Until recently the nearest available data types to structures and enums are dictionaries or sets (or possibly classes built on these), however these are not static enough, so they can't be used for this kind of type checking. There is no mechanism for the type checker to exhaustively test all of the possible variations with a dictionary, for instance, since the possibilities are unknowable at check time.

So since there are are other motivations to want an enumeration type, PEP 435 [4] was written and proposed, and an enumeration type was added in Python 3.4. Since this piece is in place, it seems likely that there will be a way in the near future to ask Python type checkers to exhaustively check enums and to alert to this common type of bug.

I expect that the static type checking features of Python 3 will improve and provide better safety in the future. I think it's interesting to think about how the type checkers could influence future programming practices in Python. It could become more common for Python to develop recommended idioms that will help to restrict the breadth of possible mistakes we make, similar to being able to check all branches of if/elif/else statements to provide better information for a static type checker to feed on. It will be interesting to see whether or not some of the ideas of what's Pythonic will change based on what's best for modern type checking.

References

[1] PEP 373: https://www.Python.org/dev/peps/pep-0373/.

[2] PEP 3107: https://www.Python.org/dev/peps/pep-3107/.

[3] PEP 484: https://www.Python.org/dev/peps/pep-0484/.

[4] PEP 435: https://www.Python.org/dev/peps/pep-0435/.