Python 3 Types in the Wild: A Tale of Two Type Systems

Ingkarat Rak-amnouykit
Rensselaer Polytechnic Institute  
New York, USA  
rakami@rpi.edu

Daniel McCrevan
Rensselaer Polytechnic Institute  
New York, USA  
mccred@rpi.edu

Ana Milanova
Rensselaer Polytechnic Institute  
New York, USA  
milanova@cs.rpi.edu

Martin Hirzel
IBM TJ Watson Research Center  
New York, USA  
hirzel@us.ibm.com

Julian Dolby
IBM TJ Watson Research Center  
New York, USA  
dolby@us.ibm.com

Abstract
Python 3 is a highly dynamic language, but it has introduced a syntax for expressing types with PEP484. This paper explores how developers use these type annotations, the type system semantics provided by type checking and inference tools, and the performance of these tools. We evaluate the types and tools on a corpus of public GitHub repositories. We review MyPy and PyType, two canonical static type checking and inference tools, and their distinct approaches to type analysis. We then address three research questions: (i) How often and in what ways do developers use Python 3 types? (ii) Which type errors do developers make? (iii) How do type errors from different tools compare?

Surprisingly, when developers use static types, the code rarely type-checks with either of the tools. MyPy and PyType exhibit false positives, due to their static nature, but also flag many useful errors in our corpus. Lastly, MyPy and PyType embody two distinct type systems, flagging different errors in many cases. Understanding the usage of Python types can help guide tool-builders and researchers. Understanding the performance of popular tools can help increase the adoption of static types and tools by practitioners, ultimately leading to more correct and more robust Python code.

CCS Concepts: • Software and its engineering → Language features: • Theory of computation → Type structures.

Keywords: Python, type checking, type inference

ACM Reference Format:

1 Introduction
Dynamic languages in general and Python in particular\(^1\) are increasingly popular. Python is particularly popular for machine learning and data science\(^2\). A defining feature of dynamic languages is dynamic typing, which, essentially, forgoes type annotations, allows variables to change type and does nearly all type checking at runtime. This is in contrast to static typing, which (typically) requires type annotations, fixes variable types, and does nearly all type checking before program execution. Dynamic typing allows for rapid prototyping, whereas static typing flags errors early and generally improves program correctness and robustness.

To enable static checking, Python has introduced PEP 484 \([18]\), which gives a syntax for optional type annotations. It leaves the semantics of type checking largely unspecified, and multiple type checking and inference tools have been developed \([3, 9, 10, 12, 15, 20]\). MyPy\(^3\) and PyType\(^4\) appear to be the canonical tools in this space\(^5\). They are relatively robust, actively developed and maintained, and they have established themselves as baseline tools for the evaluation of new Python type inference analyses \([3, 10]\).

This paper presents a study of Python 3 type usage by developers, as well as a study of the performance of MyPy and PyType on a corpus of 2,678 repositories (with a total of 173,433 files) that have partial type annotations. Surprisingly, only a small percentage, 2,678 out of over 70,000 repositories

---

\(^1\) see http://pypl.github.io/PYPL.html
\(^2\) see https://www.aitrends.com/data-science/here-are-the-top-5-languages-for-machine-learning-data-science/
\(^3\) http://mypy-lang.org/
\(^4\) https://github.com/google/pytype
\(^5\) PEP 484 "is strongly inspired by myppy"
we started out with, have (partial) type annotations. Also surprisingly, annotated repositories rarely type-check. This paper is accompanied by a web page with detailed data.

We believe that our study can benefit the Python community in two ways. Understanding the usage of types can help guide type-system designers and tool builders. Understanding the performance of popular tools can help increase developers’ adoption of static types and tools, and ultimately lead to more correct and robust Python code. While there are other studies of the behavior of dynamic languages, ours is the first study of Python 3 types in the wild.

This paper reviews the semantics of MyPy and PyType and contrasts the two type systems. It then proceeds to address three research questions:

- RQ1: How often and in what ways do developers use types?
- RQ2: Which type errors do developers make?
- RQ3: How do type errors from different tools compare?

We find that developers write type annotations that are user-defined class types more frequently than individual simple types (e.g., int, str). Developer-written type annotations are difficult to infer, while at the same time PyType can infer non-trivial types for a large number of variables that lack developer-written annotations. We find that MyPy and PyType exhibit false positives, due to their static nature, but at the same time they flag many likely runtime errors. Among other findings, we observe that the Optional[<type>] type, which indicates that a variable is either None or of type, is a significant source of both false positives and likely runtime errors. Lastly, we perform a larger-scale empirical study of MyPy and PyType errors and show that the two type systems flag largely disjoint sets of errors. Arguably, having two fundamentally different type systems violates the Zen of Python⁷, which famously states that “There should be one— and preferably only one –obvious way to do it.”

2 Background

This section describes a core Python syntax and the semantics of MyPy and PyType as gleaned from the documentation and trial and error. We stress that our intention is not to give complete formal treatment of the two type systems. For instance, we omitted class definitions and included only limited forms of assignments to properties or indices. MyPy, PyType, and the Python language are expansive and rapidly evolving. Our intention is to highlight the two type systems, compare and contrast them, and set the stage for our study of Python types and Python type checking in the wild.

MyPy and PyType both combine manual type annotations based on PEP 484 [18] with type checking; however, their approaches to type checking differ. Philosophically, MyPy provides a conventional static type system in which variables can be declared to have fixed types, and misuse of such types is an error. PyType conforms more to legal Python usage, where types can change in a function. More concretely, there are three broad differences in their semantics: (1) MyPy’s analysis is intra-procedural, while PyType does some inter-procedural reasoning. (2) PyType is generally less strict than MyPy, and (3) MyPy reports errors early whereas PyType delays error reporting.

2.1 Syntax

Fig. 1 defines a core Python type syntax. It defines a hierarchy of types where types in $\tau$ can be instantiated and types in $\sigma - \tau$ cannot be instantiated. Types in $\sigma$ can be used as type annotations and as arguments of generics.

We consider the core Python syntax in Fig. 2 and proceed to describe the semantics of MyPy and PyType over this syntax. As is standard, each system evaluates expressions and statements in a type environment $\Gamma$ where $\Gamma$ maps variables to types. Expressions have no effect on $\Gamma$ but declarations and statements may change $\Gamma$, as MyPy and PyType infer types for variables.

2.2 Expressions

Table 1 illustrates expression typing. Given a type environment $\Gamma$, helper function $\text{type}(\Gamma, e)$ returns the type of expression $e$. For expressions, $\text{type}$ is essentially the same for MyPy and PyType. (List and dictionary literals are an exception.) However, even though the semantics of retrieval are essentially the same, the types that are retrieved generally differ because the two type systems construct $\Gamma$ in different ways. As an example, given $e_1$ [$e_2$], the system retrieves the type of $e_1$ from its environment, and if the type is indexable, i.e., it is either $\text{List}[\sigma]$ or $\text{Dict}[\langle \sigma, \sigma \rangle]$, it returns the element type $\sigma$. MyPy and PyType differ in how they type list literals $[e_1, ..., e_n]$ and dictionary literals $\{e_1 : e'_1, ..., e_n : e'_n\}$. To implement the join ($\cap$), PyType creates a $\text{Union}$, whereas MyPy finds the least common ancestor in the subtype lattice.

\[ \beta ::= \text{int} | \text{float} | \text{str} | \text{bool} | \text{None} \quad \text{base types} \]
\[ \tau ::= \beta | \text{C}[\sigma] | \text{Optional}[\sigma] | \text{List}[\sigma] | \text{Dict}[\langle \sigma, \sigma \rangle] | \text{Tuple}[\sigma] | \text{Callable}[\langle \tau, \sigma \rangle] | \text{Union}[\sigma] \quad \text{+ constructed types} \]
\[ \sigma ::= \tau | \text{TypeVar} | \text{Any} \quad \text{+ type variables and Any} \]
Consider the list literal `[42, "parrot"]`. PyType types list `li` as `List[Union[int, str]]` but MyPy types it as `List[object]`, which is the join of `int` and `str` and the top of the type hierarchy. Now let `i` be an integer variable. PyType types expressions `li[1]` and `li[0]` as `Union[int, str]` and `int`, respectively (the latter is somewhat surprising). MyPy types both expressions as `object`. On a side note, PyType types `[1, 1.5]` as `List[Union[int, float]]`, while MyPy types it as `List[float]`. Dictionary literals are treated analogously.

### 2.3 Statements

Table 2 illustrates statement typing. Statements build $\Gamma$, and MyPy and PyType differ significantly. Consider assignment $x = e$. MyPy updates $\Gamma$ only if $x$ is not in $\Gamma$ (i.e., (1) it is not annotated by the user, (2) it is not assigned earlier in the code, and (3) it is not an unannotated function parameter, in which case MyPy assigns type `Any`). If the type of the right-hand side expression $e$ is a subtype of the type of $x$ in $\Gamma$, MyPy leaves $\Gamma$ intact and proceeds; otherwise, it issues an error (more on the error code later). PyType, on the other hand, *unconditionally* updates $\Gamma$ for any subsequent code.

Table 3 shows code examples that illustrate the key differences between the MyPy and PyType type systems. Consider the left-most column, which illustrates the case $x = e$. MyPy assigns a single type to a given variable, a standard practice in many type systems. It therefore reports an error in Line 6, at $z = \text{coconut}$, because $z$ has already been mapped to `int` in Line 4. In contrast, PyType maintains separate environments for the true and false branches; it updates $z$ along the false branch, then merges the two environments to type $z$ as

---

**Table 1.** Expression typing.

<table>
<thead>
<tr>
<th>Expression</th>
<th>MyPy and PyType typing</th>
<th>Code example</th>
<th>Type for example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const</code></td>
<td>(\Gamma(x))</td>
<td>1</td>
<td>(\text{type}(l) = \text{int})</td>
</tr>
<tr>
<td><code>x</code></td>
<td>(\Gamma(x))</td>
<td><code>x = \text{\textquotesingle}Ni\text{\textquotesingle}</code></td>
<td>(\text{type}(x) = \text{str})</td>
</tr>
<tr>
<td><code>e.attr</code></td>
<td>(\text{type}(\Gamma, e.\text{attr}))</td>
<td><code>a.f = 1</code></td>
<td>(\text{type}(a.f) = \text{int})</td>
</tr>
<tr>
<td><code>e1[e2]</code></td>
<td>(\sigma, \text{where } \Gamma(e1) = \text{List}[\sigma] \text{ or } \text{type}(\Gamma, e1) = \text{Dict}[_, \sigma])</td>
<td><code>li = [1, 2]</code></td>
<td>(\text{type}(\text{li}[i]) = \text{int})</td>
</tr>
<tr>
<td><code>e1; e2</code></td>
<td>(\sigma, \text{where } \Gamma(e1) = \text{Callable}[_, \sigma])</td>
<td><code>def id(x)-&gt;\text{int}: return x</code></td>
<td>(\text{type}(\text{id}(5)) = \text{int})</td>
</tr>
<tr>
<td><code>[e1; e2; ...; e_n]</code></td>
<td>(\text{List}[\text{type}(\Gamma, e_1) \lor ... \lor \text{type}(\Gamma, e_n)])</td>
<td><code>('\text{\textquotesingle}Ni\text{\textquotesingle}: 1)</code></td>
<td>(\text{type}('\text{\textquotesingle}Ni\text{\textquotesingle}: 1)) = \text{Dict[\text{str}, \text{int}])</td>
</tr>
</tbody>
</table>

---

**Table 2.** Statement typing.

<table>
<thead>
<tr>
<th>Statement</th>
<th>MyPy</th>
<th>PyType</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = e</code></td>
<td>(\Gamma[x \leftarrow \text{type}(\Gamma, e)], \text{if } x \notin \Gamma) (\longrightarrow) (\text{if } \text{type}(\Gamma, e) &lt;: \Gamma(x)) (\text{error, otherwise})</td>
<td>(\Gamma[x \leftarrow \text{type}(\Gamma, e)])</td>
</tr>
<tr>
<td><code>x.\text{attr} = e</code></td>
<td>(\Gamma[x.\text{attr} \leftarrow \text{type}(\Gamma, e) \lor \sigma] \text{ where } \Gamma(x.\text{attr}) = \sigma) (\text{error, otherwise})</td>
<td>(\Gamma[x.\text{attr} \leftarrow \text{type}(\Gamma, e) \lor \sigma] \text{ where } \Gamma(x.\text{attr}) = \sigma)</td>
</tr>
<tr>
<td><code>x[e1] = e2</code></td>
<td>(\text{error, otherwise})</td>
<td>(\Gamma[x \leftarrow \text{type}(\Gamma, e_2) \lor \sigma], \text{where } \Gamma(x) = \text{List}[\sigma] \text{ or } \text{Dict}[_, \sigma])</td>
</tr>
<tr>
<td><code>if e: s1 else: s2</code></td>
<td>require same type on both branches</td>
<td>compute the join ((\lor)) after the <code>if</code>/<code>else</code></td>
</tr>
</tbody>
</table>

---

**Table 3.** Typing examples.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = e</code></td>
<td><code>x.\text{attr} = e</code></td>
<td><code>x[e1] = e2</code></td>
</tr>
<tr>
<td>1) <code>def foo(x, y : \text{int}):</code></td>
<td>1) <code>a = A(1)</code></td>
<td>1) <code>li = [1]</code></td>
</tr>
<tr>
<td>2) <code>i = input()</code></td>
<td>2) <code>b = a</code></td>
<td>2) <code>\#MyPy: OK, \text{type}(li) = \text{List}[\text{int}]</code></td>
</tr>
<tr>
<td>3) <code>if (i == \text{\textquotesingle}Camelot\text{\textquotesingle}):</code></td>
<td>3) <code>i = input()</code></td>
<td>3) <code>\#MyPy: \text{OK}, \text{type}(li) = \text{List}[\text{int}]</code></td>
</tr>
<tr>
<td>4) <code>z = 1</code></td>
<td>4) <code>\text{if} (i == \text{\textquotesingle}Camelot\text{\textquotesingle}):</code></td>
<td>4) <code>\#PyType: \text{OK}, \text{type}(li) = \text{List}[\text{int}]</code></td>
</tr>
<tr>
<td>5) <code>\#MyPy: \text{ERROR in line 6}</code></td>
<td>5) <code>a.\text{attr} = \text{\textquotesingle}coconut\text{\textquotesingle}</code></td>
<td>5) <code>\#PyType: \text{OK}</code></td>
</tr>
<tr>
<td>6) <code>z = \text{\textquotesingle}coconut\text{\textquotesingle}</code></td>
<td>6) <code>\#MyPy: \text{OK},</code></td>
<td>5) <code>\text{li}[1] = \text{\textquotesingle}coconut\text{\textquotesingle}</code></td>
</tr>
<tr>
<td>7) <code>\#MyPy: \text{ERROR in line 6}</code></td>
<td>7) <code>\#type(a.\text{attr}) = \text{object}</code></td>
<td>6) <code>\#MyPy: \text{ERROR}</code></td>
</tr>
<tr>
<td>8) <code>\#PyType: \text{OK},</code></td>
<td>8) <code>\#type(b.\text{attr}) = \text{Any}</code></td>
<td>7) <code>\#PyType: \text{OK}</code></td>
</tr>
<tr>
<td>9) <code>\#type(z) = \text{Union}[\text{int}, \text{str}]</code></td>
<td>9) <code>\#type(li) = \text{List}[\text{Union}[\text{int, str}]]</code></td>
<td><code>\# type(li) = \text{List}[\text{Union}[\text{int, str}]]</code></td>
</tr>
<tr>
<td>10) <code>\#PyType: \text{OK,}</code></td>
<td>10) <code>\#type(b.\text{attr}) = \text{Any}</code></td>
<td></td>
</tr>
<tr>
<td>11) <code>x = 1</code></td>
<td>11) <code>\# type(li) = \text{List}[\text{Union}[\text{int, str}]]</code></td>
<td></td>
</tr>
<tr>
<td>12) <code>x = \text{\textquotesingle}coconut\text{\textquotesingle}</code></td>
<td>12) <code>\#PyType: \text{OK}</code></td>
<td></td>
</tr>
<tr>
<td>13) <code>\#MyPy: \text{OK, \text{type}(x) = \text{Any}}</code></td>
<td>13) <code>\# type(a.\text{attr}) = \text{type(b.\text{attr})}</code></td>
<td></td>
</tr>
<tr>
<td>14) <code>\#PyType: \text{OK, \text{type}(x) = \text{str}}</code></td>
<td>14) <code>= \text{Union[\text{str, int}]]</code></td>
<td></td>
</tr>
</tbody>
</table>
Union[<int, str>]. Thus, PyType maintains different typings for the same variable at different program points; however, it is not clear precisely how it does this\(^3\), and there is no formal definition in the documentation. Note that MyPy does not issue an error on Line 12; this is because \(x\) is a parameter, and untyped parameters always map to \(\text{Any}\). Had \(x\) been just an unannotated local variable, Line 12 would have triggered an error. PyType, on the other hand, treats \(x\) just as it would have treated a local variable — it updates the type at the reassignment.

Next, consider the middle column, illustrating \(x\).attr = \(e\). Both MyPy and PyType compute the join (\(\vee\)), but they do so differently. While MyPy types \(a\).attr as object, PyType assigns a more specific type, \(\text{Union}[\text{str}, \text{int}]\). While MyPy does not detect that \(a\).attr and \(b\).attr are aliases and types \(b\).attr with the default attribute type \(\text{Any}\), PyType detects the aliasing and types \(b\).attr as \(\text{Union}[\text{str}, \text{int}]\).

Finally, consider the right-most column, which illustrates \(x[e[1]] = e_2\). MyPy sets the type of the list element at list initialization and disallows the addition of incompatible elements. In contrast, PyType refines the type of the list element as we add new elements.

### 2.4 Subtyping

We write \(\sigma_1 <: \sigma_2\) to denote that \(\sigma_1\) is a subtype of \(\sigma_2\) in the sense of MyPy and PyType. The key idea is that \(\sigma_1\) is a subtype of \(\sigma_2\) if a \(\sigma_1\) object can be used where a \(\sigma_2\) object is expected”. This is the standard notion of true subtyping. Below is a core subset of subtyping rules:

1. Subtype checks including \(\text{Any}\) always succeed: \(\sigma <: \text{Any}\) and \(\text{Any} <: \sigma\). This immediately renders the type system unsound, in the sense that an expression may produce a value whose type differs from its static type.
2. Subtyping is reflexive: \(\sigma <: \sigma\).
3. Class types form a hierarchy as defined by the subclassing relation: \(C(\ldots, B[T], \ldots)\) implies that \(C[\tau] <: B[\tau]\). For example, \(\text{List}[\text{A}] <: \text{Sequence}[\text{A}]\).
4. Generic instantiations are invariant in their type arguments except for the case when one of the arguments is \(\text{Any}\). E.g., \(\text{List}[\text{int}] <: \text{List}[\text{float}]\), but \(\text{List}[\text{int}] <: \text{List}[\text{Any}]\) and \(\text{List}[\text{Any}] <: \text{List}[\text{int}]\) both hold.
5. \(\text{Union}[\sigma_1, \ldots, \sigma_n] <: \text{Union}[\sigma'_1, \ldots, \sigma'_m]\) iff for every \(\sigma_i \in \{\sigma_1, \ldots, \sigma_n\}\) there is a \(\sigma_j \in \{\sigma'_1, \ldots, \sigma'_m\}\) such that \(\sigma_i <: \sigma_j\).

### 2.5 Error Codes

Table 4 describes the error codes of MyPy and PyType. As discussed earlier, MyPy reports errors at assignments, specifically, if the right-hand side of the assignment is not compatible with the left-hand side. In contrast, PyType never reports errors at assignments. Instead, it updates the type of the left-hand side, disregarding user annotations when necessary. PyType is “more dynamic” in nature, as it propagates the object “downward” and delays error reporting until the object is actually used, e.g., as an argument, as a function value, or as an indexable value. To illustrate:

```python
1 def f(x : int):
2    z : int = 1
3    z = 'coconut'
4    return z
```

This example fails in MyPy but is type-correct in PyType because the final type of \(z\) does not conflict with the return type of \(f\), which is not manually annotated and hence defaults to \(\text{Any}\). In contrast,

```python
1 def foo(x : int):
2    print(x)
3    def f(x : int):
4        z : int = 'coconut'
5        return foo(z)
```

fails in Line 5 because the actual argument type is \(\text{str}\) incompatible with the formal parameter type \(\text{int}\). Somewhat surprisingly, but consistently with the semantics we outlined above, PyType disregards the \(\text{int}\) annotation on \(z\).

We classify the error codes into the following categories (for mapping concrete error codes into each category, see Fig. 6 and 7):

<table>
<thead>
<tr>
<th>Expr/Stmt</th>
<th>MyPy error code</th>
<th>PyType error code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.attr</td>
<td>ATTR-DEFINED OF UNION-ATTR</td>
<td>ATTRIBUTE-ERROR</td>
</tr>
<tr>
<td>e[1]</td>
<td>INDEX</td>
<td>UNSUPPORTED-OPERANDS</td>
</tr>
<tr>
<td>e[1]</td>
<td>ERROR</td>
<td>NOT-CALLABLE</td>
</tr>
<tr>
<td>e[1]</td>
<td>ARG-TYPE</td>
<td>WRONG-ARG-TYPES</td>
</tr>
<tr>
<td>return e</td>
<td>RETURN-VALUE</td>
<td>BAD-RETURN-TYPE</td>
</tr>
</tbody>
</table>

\(^3\)trivial aliasing can result in wrong types, see https://github.com/google/pytype/issues/616
1. **Syntax errors.** These are standard parse errors. For example, MyPy’s `syntax` flags Python parse errors.

2. **Shallow semantic errors.** While these are flagged by semantic analysis, that analysis is less sophisticated than full-fledged type analysis. For instance, it can mostly perform its reasoning locally or based on names. Typically there are no false positives. An example is MyPy’s `call-arg` which checks that the number and names of arguments at the function call match the function definition.

3. **Deep semantic errors.** These are standard type errors, where flagging the error requires deep semantic analysis that infers a type for variables/expressions. For example, MyPy raises `attr-defined` at expression `e.attr` if its inferred type for `e` does not have attribute `attr`. Tab. 4 lists the essential deep semantic errors in MyPy and PyType.

4. **Import/other errors.** These are import or other (rare) errors that do not fit into the above categories.

   While there is no universally agreed upon distinction between these error categories, we found the categorization useful for the purpose of this paper. The distinction is not always that sharp, but it helps characterize type errors and tools. Sect. 4 makes use of this categorization of MyPy and PyType error codes.

### 3 How Often and in What Ways Do Developers Use Python 3 Types? (RQ1)

We examine a collection of 70,826 Python repositories from public GitHub from 49,546 organizations. Only 2,678 had (partial) Python 3-style type annotations. We refer to this set as the set of **typed repositories**. Of these, 195 are malformed repositories, meaning that MyPy could not further type check the repositories due to specific errors “duplicate module” and “no parent module”. We used a flag to suppress unresolved imports; however, some subsequent errors, such as `attr-defined`: “Module has no attribute xyz” and `name-defined` will still be reported. We analyzed all user annotations on variables across the 2,678 typed repositories which contain 173,433 files.

#### 3.1 Statistic Across Typed Repositories

While the repositories are from GitHub, we collected them with a query to Google BigQuery that focused on recent Python repositories that had been watched more than once. The query was issued in August 2019, but reflects a capture of GitHub from March 20, 2019, by Google; it was meant to fetch Python code in the wild, so they are an arbitrary fetch of Python code and we did not filter for type annotations. Indeed, one goal of our study is to find out how much Python types are currently used.

Overall, we were able to find little configuration of the tools we evaluate. The initial query gathered only Python files, but, by checking the current version of the repositories on GitHub, only 101 out of 2,678 typed repositories have the standard `mypy.ini` at the top level. Furthermore, we examined 50 random repositories. Three of them have `mypy.ini`, and six of them either have MyPy in their requirement or mention it in the repository. Similarly, ten repositories have PyLint in their requirement, or mention it somewhere. Only one repository mentioned PyType, and did so in an issue.

Fig. 3 and Fig. 4 illustrates the concentration of annotations over 2,678 typed repositories.Fig. 3 shows the concentration by file. About 80% of all files do not have any type annotation, 5% have fewer than 11 annotations, and only 0.5% have more than 40 annotations. Fig. 4 shows the concentration by repository. 1,144 repositories have an average of less than 1 annotation per file, and 50 repositories have more than 20 annotations per file. Overall, the vast majority of files and repositories have a low concentration of type annotations; few files and repositories use type annotations extensively.

#### 3.2 How Do Developers Write Types?

Fig. 5 groups developer-written type annotations into the most popular categories of types, including simple types...
such as `str`, `int`, etc., and built-in composite types such as `List`, `Dict`, etc. Category “Other Typing” includes the
types defined in the typing package\(^{10}\) but excludes `List`, `Dict`, and the other frequent types that are plotted separately.
Type annotations that do not fall into any of these categories are counted in “User-defined”. “User-defined” includes user-defined classes, classes defined in built-in Python modules and third-party modules, as well as type variables and type aliases. In general, it is impossible to accurately distinguish these subcategories based on string matching, as developers can use arbitrary names for user-defined classes, type variables, type aliases, etc. While a deeper semantic analysis may be able to tease those apart, this is beyond the scope of our current work.

Fig. 5 (top) illustrates the frequency of each type category. As expected, `str`, `int`, and `bool` feature prominently. Built-in collections `List`, `Dict`, and `Tuple` and their legacy counterparts `list`, `dict`, and `tuple` feature prominently as well. MyPy accepts the lowercase annotations and treats them as `Any`-instantiated versions of their uppercase counterparts. There are 69,063 annotations with types in category “User-defined”. We examined two repositories, and the user-defined types were predominantly non-local classes. There are 11,114 `Optional` types, making `Optional` one of the most prominent annotations. Fig. 5 (bottom) illustrates the distribution of annotations among function parameters, function return types, and global assignments. Parameters dominate nearly every category because they account for the largest share of all type annotations. However, `None` and `Tuple` (unsurprisingly) are predominantly used in function return types.

To further understand the nature of developer-written annotations, we conducted an additional experiment. We selected 4,079 random files out of the 173,433 typed-repository files, stripped the annotations, and ran PyType on each file, which essentially amounts to intra-module type inference. Tab. 5 shows the results. Tab. 5(a) compares all places in the code that had either developer-written or PyType-inferred types. Tab. 5(b) focuses only on places where the original code had an explicit developer-written type annotation. PyType infers a “meaningful” type (i.e., non-`Any` and non-`None`) for 39% of all developer-written or PyType-inferred type annotations (sum of all columns except `Any` and `None`). On the other hand, looking at the developer-written annotations alone, inference matches a “meaningful” user type for only 6.1%; when we include `Any` and `None` matches, it matches 13.7% (Tab. 5(b)). It fails to infer a type for 77% of the user-written annotations. This leads to the conclusion that (1) user annotations are infrequent (as they are only a small fraction of the PyType-inferred ones) and (2) user annotations are

\(^{10}\)https://docs.python.org/3/library/typing.html

<table>
<thead>
<tr>
<th>user type</th>
<th>inferred type</th>
<th>match (but not <code>Any</code> or <code>None</code>)</th>
<th>exact</th>
<th>Optional</th>
<th>top-level</th>
<th>other non-match</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Any</code></td>
<td>34.0%</td>
<td>13.0%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>37.0%</td>
</tr>
<tr>
<td><code>None</code></td>
<td>0.3%</td>
<td>1.0%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td><code>not Any or None</code></td>
<td>13.0%</td>
<td>0.0%</td>
<td>1.0%</td>
<td>0.0%</td>
<td>0.4%</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>user type</th>
<th>inferred type</th>
<th>match (but not <code>Any</code> or <code>None</code>)</th>
<th>exact</th>
<th>Optional</th>
<th>top-level</th>
<th>other non-match</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Any</code></td>
<td>1.6%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1%</td>
</tr>
<tr>
<td><code>None</code></td>
<td>1.8%</td>
<td>6.0%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.1%</td>
</tr>
<tr>
<td><code>not Any or None</code></td>
<td>77.0%</td>
<td>0.3%</td>
<td>6.1%</td>
<td>0.2%</td>
<td>2.6%</td>
<td>3.6%</td>
</tr>
</tbody>
</table>

Table 5. Comparing developer-written types against PyType-inferred types over a set of 4,079 files.
difficult to infer and are therefore likely to be informative (as annotations appear to carry non-local information that is beyond the power of PyType’s abstract interpretation-based static analysis). This matches the results of our earlier experiment (Fig. 5), where we found that there is a large number of “User-defined” types and that those user-defined types tend to be non-local classes.

3.3 Are Typed Repositories Type Correct?
Only 318, or 15%, of the 2,678 repositories are type correct in MyPy; the remaining repositories produce a total of 41,607 errors of which 22,556 are non-import related errors. Fig. 6 shows the distribution of errors across typed repositories.

A key question arises: why are so few repositories type-correct? One hypothesis that we extend is that MyPy, by virtue of being a static type system, may be too conservative, producing false-positive warnings more often than catching actual run-time errors. Therefore, developers are discouraged from spending valuable time fixing type errors. At the same time, developers still see value in writing type annotations as they serve as documentation. We conduct additional studies towards this question and present our results in Sect. 4.

Another hypothesis that we extend is that MyPy is not the tool of choice for developers and that developers type-check their code with other tools that are less strict or more readily available in popular IDEs. Specifically, we studied PyLint and PyType. We explored several questions. Does the code in typed repositories type-check with any of these alternative tools? How do these tools compare to each other, e.g., do they catch the same set of errors or do they catch distinct sets of errors? We conduct additional studies and present our results in Sect. 4.3 and in Sect. 5.

4 Which Type Errors Do Developers Make? (RQ2)
When a developer goes through the trouble of adding manual type annotations, why are there so many type errors from MyPy? Are these errors mostly false positives, or do developers really make that many mistakes? What happens to the false positive rate when using PyType instead of MyPy for type-checking? How do the answers to the above questions vary when we break them down by different error codes?

This section examines type errors in 2,678 Python github repositories, each of which contains at least one Python 3 type annotation. We first count errors by their error codes, which can be done automatically and is thus easy to do at scale. Then, we classify errors into three categories: false positives vs. two kinds of true positives, namely likely runtime error vs. incorrect type annotation. This latter analysis requires manual inspection and cannot be fully automated, so we sampled 15 errors for each of several of the most common or most interesting error codes, and hand-inspected those.

![Figure 6. MyPy error code distribution (log scale). Solid black bars denote deep semantic errors, dotted bars denote syntax errors, striped bars denote shallow semantic errors, and solid red bars denote import/other errors.](image)

![Figure 7. PyType error code distribution (log scale).](image)

4.1 Error Code Distributions
Figure 6 shows error code distributions for MyPy and Figure 7 shows error code distributions for PyType. In both cases, we ran the respective tool to type-check all 2,678 typed repositories. Each occurrence of an error counted separately; for instance, when one of the repositories had 100 errors of a given error code, we add 100 to the count for that error code. Overall, the raw numbers are higher for PyType, because MyPy ignores untyped functions.

There are nine MyPy error codes with more than 1,000 errors each. Out of these, four indicate deep semantic errors: ASSIGNMENT, ARG-TYPE, UNION-ATTR, and RETURN-VALUE. There are also nine PyType error codes with more than 1,000 errors each. Out of these, three indicate deep semantic errors: WRONG-ARG-TYPES, UNSUPPORTED-OPERANDS, and BAD-RETURN-TYPE. We hand-analyze all of these in the following subsections. We also hand-analyzed MyPy’s VAR-ANNOTATED errors. These are mostly caused by initializing variables from empty collections (e.g., `lst = []` or `dct = {}`) and do not constitute deep semantic errors, so we did not include them.
We conjecture that the code failed during testing leading to
the addition of the None-check. In contrast, paths with no
runtime checks may have not been exercised with a None
value during testing. In 5 cases, the type error was a false
positive, because there was a runtime check for None. Typically,
the check immediately preceded the access of the attribute,

\[\text{as in the above example.} \]

89\% of the UNION-ATTR errors, 1,418
out of 1,592, involved Optional\[\text{type}\] and None.

### 4.2.3 ARG-TYPE

There are 1,851 ARG-TYPE errors. We sam-
plesd 15 and found 8 false positives, 5 likely runtime errors,
and 2 incorrect type annotations. The 5 likely runtime errors
we observed were analogous to the errors in Sect. 4.2.2—
passing an Optional\[\text{type}\] argument for a parameter ex-
pecting type. In 2 cases, the type error was due to an inco-
rrect annotation. For example, error 'Argument 1 to "Fernet"
has incompatible type "Optional\[\text{bytes}\]"'; expected "bytes" at

```python
..., encryption_key, ... = encryption_attr(
  fer = Fernet(encryption_key)
```

is due to the incorrect annotation 

### 4.2.2 UNION-ATTR

There are 1,592 UNION-ATTR errors. We sam-
plesd 15 and found 5 false positives, 10 likely runtime errors,
and 0 incorrect type annotations. A typical example for a
likely runtime error is:

```python
1 m = re.match(RE_COMP_LOCUS, raw)
2 d = m.groupdict()
3 d[‘length’] = int(d[‘length’])
```

It leads to the error "None of "Optional[Match[Any]]"
has no attribute "groupdict". If \text{re.match} fails to find a match
and returns None, there is a runtime error at Line 2. Inter-
estingly, in 4 out of 10 references to groupdict in the same
file, the developers had added a check for None:

```python
1 if m is None:
2   return {‘definition’: None}
3 d = m.groupdict()
```

We conjecture that the code failed during testing leading to
the addition of the None-check. In contrast, paths with no
runtime checks may have not been exercised with a None
value during testing. In 5 cases, the type error was a false
positive, because there was a runtime check for None. Typically,
the check immediately preceded the access of the attribute,

\[\text{as in the above example.} \]

89\% of the UNION-ATTR errors, 1,418
out of 1,592, involved Optional\[\text{type}\] and None.

### 4.2.3 ARG-TYPE

There are 1,851 ARG-TYPE errors. We sam-
plesd 15 and found 8 false positives, 5 likely runtime errors,
and 2 incorrect type annotations. The 5 likely runtime errors
we observed were analogous to the errors in Sect. 4.2.2—
passing an Optional\[\text{type}\] argument for a parameter ex-
pecting type. In 2 cases, the type error was due to an inco-
rrect annotation. For example, error 'Argument 1 to "Fernet"
has incompatible type "Optional\[\text{bytes}\]"'; expected "bytes" at

```python
..., encryption_key, ... = encryption_attr(
  fer = Fernet(encryption_key)
```

is due to the incorrect annotation 

### 4.2.2 UNION-ATTR

There are 1,592 UNION-ATTR errors. We sam-
plesd 15 and found 5 false positives, 10 likely runtime errors,
and 0 incorrect type annotations. A typical example for a
likely runtime error is:

```python
1 m = re.match(RE_COMP_LOCUS, raw)
2 d = m.groupdict()
3 d[‘length’] = int(d[‘length’])
```

It leads to the error "None of "Optional[Match[Any]]"
has no attribute "groupdict". If \text{re.match} fails to find a match
and returns None, there is a runtime error at Line 2. Inter-
estingly, in 4 out of 10 references to groupdict in the same
file, the developers had added a check for None:

```python
1 if m is None:
2   return {‘definition’: None}
3 d = m.groupdict()
```

We conjecture that the code failed during testing leading to
the addition of the None-check. In contrast, paths with no
runtime checks may have not been exercised with a None
value during testing. In 5 cases, the type error was a false
positive, because there was a runtime check for None. Typically,
the check immediately preceded the access of the attribute,
as the callers expected the actual return type (see bytes vs. ByteString example in Sect. 4.2.3.

### 4.3 Examples of PyType Errors

Like the previous section did for MyPy, this section presents our hand-analysis of PyType errors into false positives, likely runtime errors, and incorrect type annotations. We again illustrate these with real-world examples, which we hope are useful for the users and developers of type-checking tools.

#### 4.3.1 wrong-arg-types

There are 4,938 wrong-arg-types errors. We sampled 15 and found 8 false positives, 6 likely runtime errors, and 1 incorrect type annotation. One reason for false positives is incorrect or missing library stubs. For example, in

1. new_tree = ET.ElementTree(new_root)
2. ET.dump(new_tree)

ET.dump does accept an ElementTree argument, however PyType issues a wrong-arg-types error. Another reason for false positives is intricate control flow checks that render the code safe, checks that one cannot reasonably expect a static type system to reason about. For example, PyType issues a wrong-arg-types error at call

1. optimize_all(expression, None,
   2.     specific=False, general=True)

This is because the second parameter of optimize_all, rels, is of type ContextDict but is passed None. Any static type system is expected to flag this as an error. There is no runtime error though, because all access to rels in optimize_all is guarded by specific==True, and in this context of invocation we have specific=False.

We observed many likely runtime errors as well. They were overwhelmingly due to misuse of standard libraries:

1. dotfile = NamedTemporaryFile(suffix='\dot\')
2. dotfile.write('digraph G {\n
Here NamedTemporaryFile.write expects a bytes argument and passing a string results in a runtime error. We observed the string/bytes misuse several times, across different repositories. In one interesting case, PyType recognized that built-in function open was used with the binary flag b, and it flagged several write’s to the corresponding file because they passed str arguments instead of bytes arguments. We also observed library functions with Boolean formal parameters being called with actual parameters that are integers 0 or 1.

#### 4.3.2 unsupported-operands

There are 2,887 unsupported-operands errors. We sampled 15 and found 6 false positives, 9 likely runtime errors, and 0 incorrect type annotations. In several cases, the error was due to a mismatch of library versions.

1. tb = traceback.extract_stack()
2. for back in tb:
3.     key = back[:3]

#### 4.4 Discussion and Analysis

Sect. 4 started by asking why there are so many type errors from MyPy. Given what we have learned, the answer is that many programmers do not run MyPy on their code and MyPy has many false positives. While programmers can make false positives go away by changing their code, it is less work to just ignore them. Table 6 summarizes the quantitative findings from our manual inspection of 105 MyPy errors and 77 PyType errors. In addition to the 60 MyPy errors discussed

<table>
<thead>
<tr>
<th></th>
<th>false positives</th>
<th>true positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyPy</td>
<td>52 (49%)</td>
<td>29 (28%)</td>
</tr>
<tr>
<td>PyType</td>
<td>34 (44%)</td>
<td>32 (42%)</td>
</tr>
</tbody>
</table>

The above code was written against version 2.4 of Python’s traceback library, where traceback.extract_stack() returns a list of (filename, line number, function name, text) tuples, each tuple representing a stack frame summary. In version 3.5 of the library, the signature changes and now traceback.extract_stack() returns a list of FrameSummary objects. PyType, which checks against extensive Python 3 stubs, issues an unsupported-operands error in line 5 as FrameSummary is not indexable.

There were interesting likely runtime errors in scipy:

1. w = fftfreq(n)*h*2*pi/period*n
2. w[0] = 1
3. w = 1j/tanh(w)
4. w[0] = 0j

w, which is initialized as a complex number in Line 3, is not indexable. PyType reports an unsupported-operands error in Line 4. If the function executes, there is a runtime error.

With respect to the false positives, we again observed mostly complex control flow we cannot expect a static type system to handle. Consider another example from scipy:

1. arglist = get_arglist(I_type, T_type)
2. if T_type is None:
3.     dispatch = "%s" % (I_type,)

PyType reports an error inside get_arglist because T_type, which is an Optional type, is used in a string concatenation. A detailed look reveals that the string concatenation is guarded by checks (on different values, not the T_type parameter), and the checks ensure that T_type is not None at the string concatenation.

#### 4.3.3 bad-return-type

There are 2,627 bad-return-type errors. We sampled 15 and found 4 false positives, 2 likely runtime errors, and 9 incorrect type annotations. Unsurprisingly, the nature of the errors we saw is similar to that of MyPy’s return-value errors.

### Table 6. Summary of hand-examined error reports.

<table>
<thead>
<tr>
<th></th>
<th>false positives</th>
<th>true positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyPy</td>
<td>52 (49%)</td>
<td>29 (28%)</td>
</tr>
<tr>
<td>PyType</td>
<td>34 (44%)</td>
<td>32 (42%)</td>
</tr>
</tbody>
</table>
in Sect. 4.2 and the 45 PyType errors discussed in Sect. 4.3, we also inspected various other categories of deep semantic errors from both tools, omitted from the detailed discussion but included in Table 6.

About 49% of the MyPy errors and 44% of the PyType errors we examined were false positives. MyPy’s false positives were overwhelmingly due to either redefinition of variables with incompatible types or Optional[<type>] being guarded by None checks. One highly frequent error, var-annotated, does not catch actual runtime errors, and another frequent error, assignment, exhibits false positives. We found less frequent error categories, particularly union-attr and override, to be more likely to report type errors that may lead to runtime errors.

Even though PyType avoids the false positives of MyPy’s assignment, list-item, and dict-item as it pushes errors down towards operations, its false positive rate is not much lower. This indicates that false positive MyPy assignment errors have likely turned into false positive PyType wrong-argument-types of unsupported-operands errors.

Both MyPy’s arg-type and PyType’s wrong-argument-types flagged a good number of likely runtime errors each, and the nature of errors was similar. In both cases, false positives often involved Optional[<type>] types. MyPy and PyType flagged potential flows of None values to operations that expected non-None arguments but they were rendered false positives due to sometimes complex non-None checks.

Both MyPy’s return-value and PyType’s bad-return-type revealed large percentages of incorrect user annotations: a method m actually returns a value incompatible with the annotation on the return.

We conclude that MyPy would benefit if it allowed redefinition of variables with incompatible types in some form. This could be accomplished by using Static Single Assignment (SSA) form for local variables in straight-line code. For example, if it could rewrite the earlier example as follows:

```python
values1 = map(repr, result[0].values())
values2 = zip(labels, values1)
values3 = [...] for (label, value) in values2
```

MyPy would infer separate types for values1, values2, and values3, and check corresponding usage with these types. The above example is type correct, but suppose for the sake of argument values2 was used as a string in Line 3. MyPy would flag the error at the usage point in Line 3. assignment errors we observed typically involved straight-line code like this, which will benefit from simple SSA.

Another lesson learned is that Optional[<type>] features prominently, both in user annotations (see Fig. 5) and in MyPy and PyType type errors. Optional[<type>] was a large source of error messages, particularly, 90% of the union-attr errors were due to the option of None. In some cases the error messages signaled a likely runtime error, for example, passing an Optional argument, which can be None, to a library method that expected a non-None parameter. In other cases the errors were false positives, as the runtime check for non-None value immediately preceded the expected non-None assignment. We conjecture that both MyPy and PyType would benefit from reasoning about local checks for non-None values. The long history of work on non-null types in Java [1, 2, 5, 14] can bring insights into Python type checking.

Both type systems produce useful warnings as well. Even those errors that were false positives may have been true positives at first and only made safe with complex checks after developers experienced runtime errors.

5 How Do Type Errors From Different Tools Compare? (RQ3)

Do different type-checking tools just differ at the surface, e.g., by using different error codes for the same errors? If yes, what is the mapping between error code? When one tool reports more errors than another, does it more-or-less report a superset? Or if no, do the tools differ more fundamentally, using error codes with little correspondence or overlap? We counted matches between errors reported on the same source code line to answer these questions. This section directly compares MyPy to PyType. We also include PyLint as it is the default linter in Visual Studio Code and other IDEs and has received praise in developer blogs[12]. This section studies the same repositories as the earlier sections in this paper. We first describe how we run the tools, then compare MyPy to PyType in Sect. 5.1 and MyPy to PyLint in Sect. 5.2.

Tools. MyPy, PyType, and PyLint each have a plethora of command-line options that impact the number and kinds of messages reported, so we detail our exact usage here. We use MyPy 0.770 with the following flags:

```bash
mypy --show-error-codes --namespace-packages --ignore-missing-imports --show-column-numbers PATH/file1.py PATH/file2.py PATH/file3.py ...
```

We use PyType 2020.04.01 with the following flags:

```bash
pytype --keep-going PATH/file1.py
```

We run PyType on each file because it stops checking when it finds the first error in a file.

We use PyLint 2.5.2 with the following flags:

```bash
pylint -d all -e typecheck --unsafe-load-any-extension=y PATH/file1.py PATH/file2.py PATH/file3.py ...
```

We are interested in errors reported by MyPy, PyType, and PyLint on the same line of code[13] in order to compare the tools. This section counts errors differently than Sect. 4: Fig. 6


and Fig. 7 counted every error, while here multiple errors of the same category on the same line count as one error.

5.1 MyPy vs. PyType

Excluding import-related errors, MyPy reports 23,665 errors and PyType reports 65,938 errors. For example, there are 617 of MyPy’s operator errors versus 2,887 of PyType’s similar supported-operands errors.

5.1.1 Deep Semantic Errors. Fig. 8 contrasts deep semantic errors from the two tools. Our supplementary repository, Py3TypeInTheWildDLS20, contains a dynamic version of this figure (graph 1), which breaks down each MyPy category into corresponding matching PyType subcategories. There is a good connection between MyPy’s arg-type, union-attr, return-value and the corresponding PyType error codes. Other error codes such as assignment are not captured by PyType. Most notably, the figure shows that the vast majority of MyPy errors remain unmatched by PyType, and analogously, the majority of PyType errors remain unmatched by MyPy. In the repository Py3TypeInTheWildDLS20 (graph 8) plots only matching errors, excluding the dominating No Match category; they emphasize matching errors across all error categories. We detail the deep semantic error categories below.

**return-value**: Out of 1,399 MyPy return-value errors, 553 match with PyType’s bad-return-type. The remaining 864 errors are due to the different semantics (Sect. 2). To illustrate, consider the following code:

```python
1 def f(a:int=None) -> str:
2     a = 'str'
3     return a
```

MyPy rejects this function, flagging both the assignment at Line 2 and the return in Line 3. PyType, however, does not report an error.

On the other hand, there are 2,577 total PyType bad-return-type. Most of those unmatched with MyPy errors have two causes. PyType views a function consisting of a pass statement as returning None and rejects any mismatched return type. MyPy allows this. The second cause is that MyPy does not type-check unannotated functions. E.g.,

```python
1 def g(b):
2     return 1
3 def f(a:int) -> str:
4     return g(a)
```

PyType infers that g returns int and flags a bad-return-type error. MyPy does not check g assigning return type Any which agrees with str.

5.1.2 Shallow Semantic Errors. There is little overlap between MyPy’s and PyType’s shallow semantic errors, as shown in Py3TypeInTheWildDLS20 (graph 3 and 4). 85.2% of MyPy’s shallow semantic errors do not match with PyType’s errors, and 95.7% of PyType’s do not match with MyPy’s. The number of shallow semantic errors is small. There are 2,472 and 3,132 MyPy and PyType errors respectively compared to 19,145 and 45,856 deep semantic errors.

There is one noticeable matching error. MyPy’s call-arg checks the number and names of arguments at function calls. Out of 411 errors, 45 match to PyType’s missing-parameter and 29 match to wrong-arg-count.
Each tool has different categories for uncommon cases. For example, there are 3 MyPy exit-return errors that get reported because the \_exit\_ function that always returns False has return type bool. PyType does not catch this error.

### 5.1.3 Syntax Errors

In Py3TypeInTheWildDLS20 (graph 8), 2 of the 5 large categories, valid-type and syntax, account for 631 and 417 matching pairs, or 18.6% and 12.3% of total matchings. They catch wrong annotation syntax and wrong language syntax. 97.0% of valid-type and 93.0% of syntax match to one of 3 PyType categories: invalid-annotation, python-compiler-error, and name-error. For example:

```python
1 def \_init\_(self, title: string,
2     size: int, parent=None): ...
```

fails both with MyPy’s valid-type and with PyType’s invalid-annotation. The tight connections between these 2 MyPy error codes and 3 PyType error codes indicates that MyPy and PyType capture syntax errors in a similar way.

### 5.1.4 Import Errors

Py3TypeInTheWildDLS20 (graph 7) shows the overall relationship between MyPy and PyType errors when excluding the dominating no match category. Out of 3,270 MyPy misc matching errors, 84.2% and 11.7% match to PyType import-error and invalid-annotation, respectively. There are 7,839 name-defined errors, 2,488 of those match to PyType errors, and 97.5% match to PyType’s name-error. MyPy’s attr-defined matches with PyType’s importer-error and attribute-error for 65.7% and 20.2% respectively. Both tools are fairly consistent with reporting import-related errors.

### 5.1.5 Conclusion

The results of the comparison support our argument from Sect. 2 that MyPy and PyType are essentially two different type systems. For most major programming languages, such a finding would come as a shock, so it is somewhat surprising to find Python in this situation. One of the lessons learned is that MyPy behaves more like a traditional type system and PyType behaves more like a static analysis tool. These seem like two directions that types can
go for dynamic languages like Python. This leads to differences in reported errors, as well as in when and where errors get detected. MyPy and PyType would serve the Python community better if they increased the overlap between reported errors. We hope that the comparisons in this section can serve as a starting point for such a convergence journey.

There is overlap in deep semantic errors ARG-TYPE/WRONG-ARG-TYPES and RETURN-VALUE/BAD-RETURN-VALUE, but there are also significant numbers of no-matches. The tools define different kinds of shallow semantic errors that generally do not overlap. For syntax errors, the tools overlap. Lastly, PyType reports larger numbers of errors because unlike MyPy, it also checks unannotated functions.

5.2 MyPy vs. PyLint

Next, we compare error messages from MyPy and PyLint. PyLint is a static analyzer that offers a wide range of checking. Unlike the other tools, PyLint checks the quality of the code; for example, it enforces a coding standard and reports code smells. There are about 200 error codes. We are interested in the error codes in the TYPECHECK category, which most closely relates to the MyPy and PyType errors we study.

Fig. 9 illustrates the distribution of errors in TYPECHECK. The largest error category, no-member, is reported when an unknown member in a variable is accessed. Similarly to MyPy, many are import-related. Moreover, no-member is one of the most common false positive errors in PyLint\textsuperscript{16}. Thus, we excluded this category from the comparison.

5.2.1 Deep Semantic Errors. Out of MyPy’s 19,145 deep semantic errors, 89.9% do not match with PyLint’s errors and 9.1% match with NO-MEMBER. Py3TypeInTheWildDLS20 (graph 9) illustrates. For the remaining 1%, there is a small overlap between MyPy’s INDEX and PyLint’s UNSUBSCRIPTABLE-OBJECT which account for 0.3% of the MyPy errors.

5.2.2 Matching Errors. Excluding import-related errors, there are only 190 MyPy errors (out of 23,665) that match with PyLint’s TYPECHECK errors. Py3TypeInTheWildDLS20 (graph 14) plots the matching MyPy errors.

5.2.3 Conclusion. Generally, PyLint fails to detect deep semantic errors. It catches some shallow semantic errors but the overlap is relatively small. The advantage of PyLint is that it offers quality checking which can be appealing to users who are looking for a linter as well as some shallow semantic checking.

6 Related Work

Our paper should be viewed in a tradition of studies about the behavior of dynamic programming languages in the wild. Such studies help guide language designers, compiler writers, and tool builders with data on how certain language features actually get used. Holkner and Harland\textsuperscript{11} measured the dynamic behavior of Python programs, finding that use of dynamic objects (e.g., adding fields) and dynamic code (e.g., eval) is not only prevalent during program startup but continues throughout program execution. Unlike our paper, theirs did not focus on Python 3 types; it pre-dates PEP 484\textsuperscript{18}. Richards et al. analyze runtime traces of JavaScript programs and find that dynamic objects and eval are rampant\textsuperscript{16}. Bierman et al. define a core calculus for TypeScript, which adds types to JavaScript via a compile-and-erase approach\textsuperscript{7}. Just like with Python 3, this means type annotations are ignored at runtime, leading to unsoundness. The type checker does not statically check downcasts or conversions to and from the Any type, nor are they checked at runtime, and thus, the type of a runtime value of an expression can differ from its static type. Their paper does not include a corpus study. Morandat et al. study the R programming language and find that besides dynamic types in the usual sense, a distinguishing feature of R is lazy evaluation\textsuperscript{13}. To the best of our knowledge, ours is the first study of how real-world programs use Python 3 types.

There have been multiple interesting typed Python dialects. RPython\textsuperscript{4} is a Python subset that was initially designed for PyPy\textsuperscript{17}, a Python just-in-time compiler written in Python. RPython restricts Python’s dynamic features to simplify type inference and enable efficient execution on the CLI and the JVM. Cython is a Python superset that is used in popular libraries for scientific computing and machine learning\textsuperscript{6}. Cython adds explicit type annotations to enable compiling to efficient C code. Reticulated Python offers gradual typing for Python: it statically type-checks code based on type annotations, then inserts dynamic checks at the boundaries to untyped code\textsuperscript{19}. Having been published in the same year as PEP 484\textsuperscript{18}, it uses different type names than those supported by Python 3 today. Whereas these papers offer typed Python dialects, our paper studies types in the mother language from which these dialects calved.
There have been several papers on type inference for Python, in spite of (or perhaps because of?) Python’s dynamic features and the unsoundness of Python 3 types. Maia et al.’s type inference pre-dates Python 3 type annotations and focuses on RPython instead, offering fairly standard type rules [12]. Xu et al. augment standard type rules with a probabilistic approach to exploiting additional information such as identifier naming conventions [20]. Fritz and Hage implement type inference via abstract interpretation and experiment with tuning the precision by modifying flow sensitivity and context sensitivity [9]. Hassan et al. implement type inference via a MaxSMT solver, maximizing optional equality constraints while satisfying all mandatory type constraints [10]. Dolby et al. use type inference to find bugs in Python-based deep learning code by inferring tensor shapes and dimensions [8]. Allamanis et al. use deep learning to implement Python type inference, based on manual type annotations as ground truth labels [3]. TypeWriters infer the argument types and return types of functions using probabilistic type prediction and search-based refinement [15]. In contrast, our paper studies how types are being used and how MyPy and PyType (the most popular practical type checking and inference tools) behave on a large corpus of open-source repositories.

7 Conclusion

This paper presents a study of a comprehensive corpus of open-source code with Python 3 type annotations. The picture is mixed. On the one hand, types can already help catch many bugs, such as in the use of Optional types that may be None and in function return type annotations. On the other hand, type checking tools frequently disagree with user annotations and with each other. Most open-source projects do not yet use Python 3 types, and of those that do, few type-check. We hope that our paper will help guide practitioners to make the best use of what is available today while inspiring researchers to improve upon the status quo. With apologies to Dickens: It was the best of types, it was the worst of types ...

References