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:
Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3426422.3426981

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 mypy"

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[6].

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 Python[7], 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

$$
\begin{array}{lll}
\beta ::= \texttt{int} \mid \texttt{float} \mid \texttt{str} \mid \texttt{bool} \mid \texttt{None} & & \text{base types} \\
\tau ::= \beta \mid \texttt{C} \mid \texttt{C}[\sigma] \mid \texttt{Optional}[\sigma] & & \text{+ constructed types} \\
\quad \mid \texttt{List}[\sigma] \mid \texttt{Dict}[\sigma_1, \sigma_2] \mid \texttt{Tuple}[\overline{\sigma}] & & \\
\quad \mid \texttt{Callable}[[\overline{\sigma_1}], \sigma_2] \mid \texttt{Union}[\overline{\sigma}] & & \\
\sigma ::= \tau \mid \texttt{TypeVar} \mid \texttt{Any} & & \text{+ type variables and Any}
\end{array}
$$

Figure 1. Type syntax (simplified). `C` is a user-defined class.

$$
\begin{array}{ll}
f ::= \texttt{def f}(\overline{\texttt{x} : \sigma_1}) \rightarrow \sigma_2 : s & \text{function definition} \\
v ::= \texttt{x} : \sigma = e & \text{variable declaration} \\
e ::= \texttt{const} \mid \texttt{x} \mid e.attr \mid e_1[e_2] \mid e_1(e_2) & \text{expressions} \\
\quad \mid [e_1, ... e_n] \mid \{e_1 : e'_1, ..., e_n : e'_n\} & \\
s ::= \texttt{x} = e \mid \texttt{x}.attr = e \mid \texttt{x}[e_1] = e_2 & \text{statements} \\
\quad \mid \overline{s} \mid \texttt{if } e\texttt{:}\ s_1 \texttt{ else:}\ s_2 \mid \texttt{return } e &
\end{array}
$$

Figure 2. Language syntax (simplified).

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 interprocedural 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 τ can be instantiated and types in $\sigma - \tau$ cannot be instantiated. Types in σ 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 Γ where Γ maps variables to types. Expressions have no effect on Γ but declarations and statements may change Γ, as MyPy and PyType infer types for variables.

2.2 Expressions

Table 1 illustrates expression typing. Given a type environment Γ, helper function $type(\Gamma, e)$ returns the type of expression e. For expressions, $type$ 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 Γ 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 $\texttt{List}[\sigma]$ or $\texttt{Dict}[\_, \sigma]$, it returns the element type σ.

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 ($\vee$), PyType creates a `Union`, whereas MyPy finds the least common ancestor in the subtype lattice.

[6]https://py3typeinthewilddls20.github.io/Py3TypeInTheWildDLS20/
[7]https://www.python.org/dev/peps/pep-0020/

Table 1. Expression typing.

Expression	MyPy and PyType typing	Code example	Type for example
`const`	$type(\Gamma, \texttt{const})$	`1`	$type$(`1`) = `int`
`x`	$\Gamma(\texttt{x})$	`x = 'Ni'`	$type$(`x`) = `str`
$e.attr$	$type(\Gamma, e.attr)$	`a.f = 1`	$type$(`a.f`) = `int`
$e_1[e_2]$	σ, where $type(\Gamma, e_1)$ = `List`[σ] or $type(\Gamma, e_1)$ = `Dict`[_, σ]	`li = [1,2]`	$type$(`li[i]`) = `int`
$e_1(e_2)$	σ, where $type(\Gamma, e_1)$ = `Callable`[_, σ]	`def id(x)->int:return x`	$type$(`id(5)`) = `int`
$[e_1, ..., e_n]$	`List`[$type(\Gamma, e_1) \vee ... \vee type(\Gamma, e_n)$]]	`[1,2]`	$type$(`[1,2]`) = `List[int]`
$\{e_1 : e'_1, ..., e_n : e'_n\}$	`Dict`[$type(\Gamma, e_1) \vee ... \vee type(\Gamma, e_n), type(\Gamma, e'_1) \vee ... \vee type(\Gamma, e'_n)$]]	`{'Ni': 1}`	$type$(`{'Ni':1}`)=`Dict[str, int]`

Table 2. Statement typing.

Statement	MyPy	PyType
`x` = e	Γ[`x` $\leftarrow type(\Gamma, e)$], if `x` $\notin \Gamma$	Γ[`x` $\leftarrow type(\Gamma, e)$]
	–, if $type(\Gamma, e)$ <: Γ(`x`)	
	error, otherwise	
`x`.$attr$ = e	Γ[`x`.$attr \leftarrow type(\Gamma, e) \vee \sigma$] where Γ(`x`.$attr$) = σ	Γ[`x`.$attr \leftarrow type(\Gamma, e) \vee \sigma$] where Γ(`x`.$attr$) = σ
`x`[e_1] = e_2	–, if $type(\Gamma, e_2)$ <: σ, where Γ(`x`)=`List`[σ] or `Dict`[_, σ]	Γ[`x` $\leftarrow type(\Gamma, e_2) \vee \sigma$], where Γ(`x`)=`List`[σ] or `Dict`[_, σ]
	error, otherwise	
`if` e: s_1 `else`: s_2	require same type on both branches	compute the join ($\vee$) after the if/else

Table 3. Typing examples.

`x` = e	`x.attr` = e	`x`[e_1] = e_2
`1 def foo(x, y : int):`	`1 ...`	
`2   i = input()`	`2 a = A(1)`	
`3   if (i == 'Camelot'):`	`3 b = a`	
`4     z = 1`	`4 i = input()`	`1 li = [1]`
`5   else:`	`5 if (i == 'Camelot'):`	`2 #MyPy: OK, type(li) = List[int]`
`6     z = 'coconut'`	`6   a.attr = 'coconut'`	`3 #PyType: OK, type(li) = List[int]`
`7 #MyPy: ERROR in line 6`	`7 else:`	`4`
`8 #PyType: OK,`	`8   a.attr = 1`	`5 li[1] = 'coconut'`
`9 #  type(z) = Union[int, str]`	`9 #MyPy: OK,`	`6 #MyPy: ERROR`
`10`	`10 #  type(a.attr) = object`	`7 #PyType: OK,`
`11   x = 1`	`11 #  type(b.attr) = Any`	`8 #  type(li) = List[Union[int,str]]`
`12   x = 'coconut'`	`12 #PyType: OK,`	
`13 #MyPy: OK, type(x) = Any`	`13 #  type(a.attr) = type(b.attr)`	
`14 #PyType: OK, type(x) = str`	`14 #  = Union[str, int]`	

Consider the list literal `[42,'parrot']`. PyType types `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[i]` 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 Γ, and MyPy and PyType differ significantly. Consider assignment `x` = e. MyPy updates Γ *only if* `x` is not in Γ (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 Γ, MyPy leaves Γ intact and proceeds; otherwise, it issues an error (more on the error code later). PyType, on the other hand, *unconditionally* updates Γ 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 = '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

`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[8], 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 `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, `Union[str,int]`. While MyPy does not detect that `a.attr` and `b.attr` are aliases and types `b.attr` with the default attribute type `Any`, PyType detects the aliasing and types `b.attr` as `Union[str,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 σ_1 is a subtype of σ_2 in the sense of MyPy and PyType. The key idea is that σ_1 is a subtype of σ_2 if "a σ_1 object can be used where a σ_2 object is expected". This is the standard notion of *true subtyping*. Below is a core subset of subtyping rules:

1. Subtype checks involving `Any` always succeed: $\sigma <:$ `Any` and `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(...,B[T],...)` implies that `C`$[\sigma] <:$ `B`$[\sigma]$. For example, `List[A] <: Sequence[A]`.
4. Generic instantiations are invariant in their type arguments except for the case when one of the arguments is `Any`. E.g., `List[int]` $\not<:$ `List[float]`, but `List[int]` $<:$ `List[Any]` and `List[Any]` $<:$ `List[int]` both hold.
5. `Union`$[\sigma_1, ...\sigma_n] <:$ `Union`$[\sigma'_1, ...\sigma'_m]$ iff for every $\sigma_i \in \{\sigma_1, ...\sigma_n\}$ there is a $\sigma_j \in \{\sigma'_1, ...\sigma'_m\}$ 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
def f(x : int):
  z : int = 1
  z = 'coconut'
  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 `Any`. In contrast,

```python
def foo(x: int):
  print(x)
def f(x : int):
  z : int = 'coconut'
  return foo(z)
```

fails in Line 5 because the actual argument type `str` is incompatible with the formal parameter type `int`. Somewhat surprisingly, but consistently with the semantics we outlined above, PyType disregards the `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 4. Error codes.

Expr/Stmt	MyPy error code	PyType error code
e.*attr*	ATTR-DEFINED or UNION-ATTR if *type*(Γ, e), or one of its members, has no attribute *attr*	ATTRIBUTE-ERROR
$e_1[e_2]$	INDEX if *type*(Γ, e_1) is neither `List[_]` nor `Dict[_,_]`	UNSUPPORTED-OPERANDS
$e_1(e_2)$	OPERATOR if *type*$(\Gamma, e_1) \neq$ `Callable[_,_]`	NOT-CALLABLE
$e_1(e_2)$	ARG-TYPE if *type*$(\Gamma, e_1) =$ `Callable`$[[\sigma], \_]$ and *type*$(\Gamma, e_2) \not<: \sigma$	WRONG-ARG-TYPES
`x = []`	VAR-ANNOTATED if $\mathtt{x} \notin \Gamma$	–
`x = {}`	VAR-ANNOTATED if $\mathtt{x} \notin \Gamma$	–
`x` = *e*	ASSIGNMENT if *type*$(\Gamma, e) \not<: \Gamma(\mathtt{x})$	–
`x`.*attr* = *e*	ASSIGNMENT if *type*(Γ, e) is not compatible with *attr* annotation	–
`x`$[e_1] = e_2$	LIST-ITEM or DICT-ITEM if $\Gamma(\mathtt{x}) =$ `List`$[\sigma]$ or $\Gamma(\mathtt{x}) =$ `Dict`$[\_, \sigma]$ and *type*$(\Gamma, e_2) \not<: \sigma$	–
`return` *e*	RETURN-VALUE if *type*(Γ, e) is not compatible with function annotation	BAD-RETURN-TYPE

[8]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[9] 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.

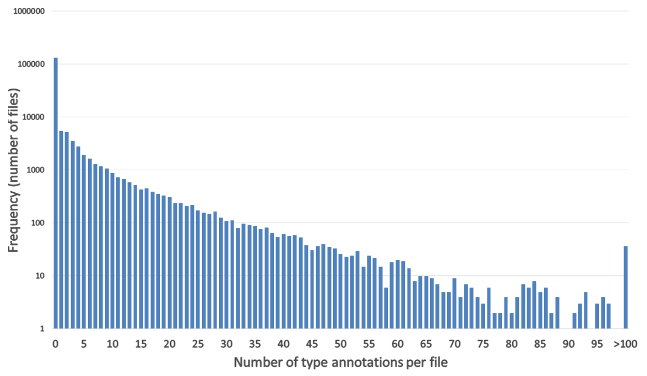

Figure 3. Number of type annotations per file across 2,678 repositories (log scale).

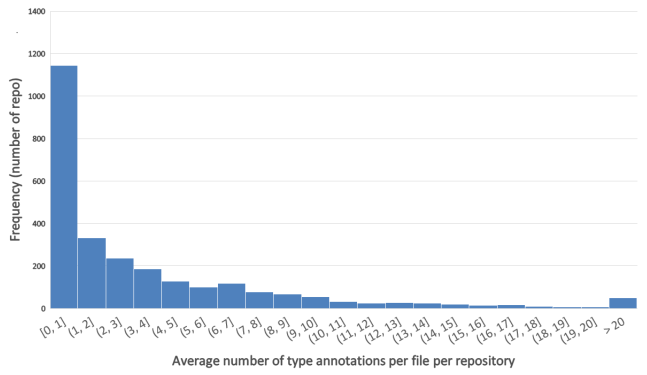

Figure 4. Histogram of the average number of type annotations per file per repo, across 2,678 repositories.

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

[9]https://github.com/wala/graph4code/blob/master/extraction_queries/bigquery.sql

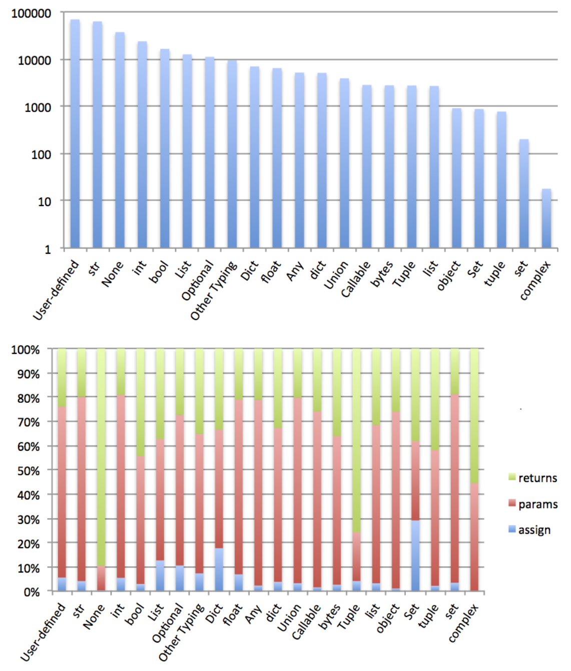

Figure 5. User type hints. The top graph is a histogram of annotations per type (log scale); the bottom graph shows the distribution of individual type annotations among function parameters, function return types, and assignments.

Table 5. Comparing developer-written types against PyType-inferred types over a set of 4,079 files.

user type	inferred type					
	Any	None	match (but not Any or None)			other
			exact	Optional	top-level	non-match
Any	34.0%	13.0%	0	0	0	37.0%
None	0.3%	1.0%	0	0	0	0.0%
not Any or None	13.0%	0.0%	1.0%	0.0%	0.4%	0.6%

(a) Inferred or user-annotated types (76,313)

user type	inferred type					
	Any	None	match (but not Any or None)			other
			exact	Optional	top-level	non-match
Any	1.6%	0	0	0	0	0.1%
None	1.8%	6.0%	0	0	0	0.1%
not Any or None	77.0%	0.3%	6.1%	0.2%	2.6%	3.6%

(b) Only user-annotated types (12,529)

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

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 run-time 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.

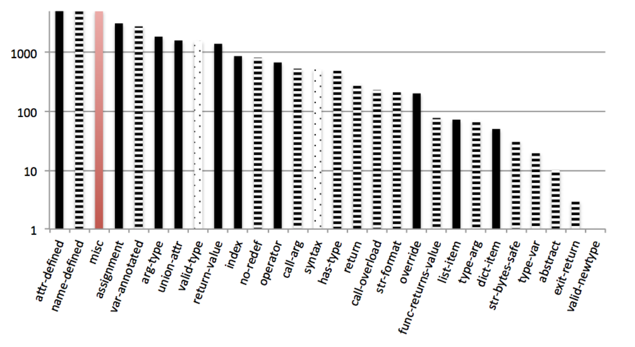

Figure 6. MyPy error code distribution (log scale). Solid black bars denote deep semantic errors, dotted bars denote syntax errors, striped bars denote shallow semantic errors, and solid red bars denote import/other errors.

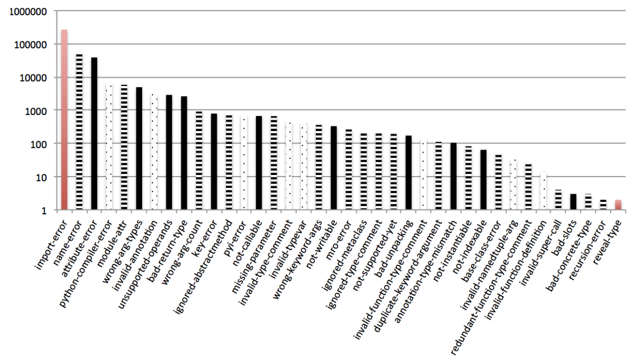

Figure 7. PyType error code distribution (log scale).

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 in the discussion below. We did not hand-analyze NAME-DEFINED, MISC, and ATTR-DEFINED from MyPy or ATTRIBUTE-ERROR from PyType, because these errors are largely caused by missing imports and thus shed little light on the quality of type annotations or type checking tools.

4.2 Examples of MyPy Errors

This section presents our hand-analysis of MyPy errors into false positives, likely runtime errors, and incorrect type annotations. Furthermore, this section exemplifies each error code with snippets of idiomatic Python code written by programmers that do not pass the MyPy type checker. Doing so may help programmers use MyPy more effectively and may help designers of type checkers improve their tools.

4.2.1 ASSIGNMENT. There are 3,104 ASSIGNMENT errors. We sampled 15 and found 15 false positives, 0 likely runtime errors, and 0 incorrect type annotations. In all but one case, the false positive was due to the issue of redefinitions with incompatible types.[11] Below is a typical example:

```
1 values = map(repr, result[0].values())
2 values = zip(labels, values)
3 values = [... for (label, value) in values]
```

MyPy infers type `Iterator[str]` for variable `values` based on the signature for `map`. This clashes with the type MyPy infers for the expression in Line 2, based on the signature of `zip: Iterator[Tuple[Any, str]]`. This is a false positive, as the use of `values` in Line 3 is consistent with the return value of `zip`.

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

```
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 `re.match` fails to find a match and returns None, there is a runtime error at Line 2. Interestingly, in 4 out of 10 references to `groupdict` in the same file, the developers had added a check for None:

```
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 in the above example. 89% of the UNION-ATTR errors, 1,418 out of 1,592, involved `Optional[` *type*`]` and None.

4.2.3 ARG-TYPE. There are 1,851 ARG-TYPE errors. We sampled 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[` *type*`]` argument for a parameter expecting type. In 2 cases, the type error was due to an incorrect annotation. For example, error 'Argument 1 to "Fernet" has incompatible type "ByteString"; expected "bytes"' at

```
1 ..., encryption_key, ... = encryption_attrs(
2                               hw_session, label)
3 fer = Fernet(encryption_key)
```

is due to the incorrect annotation `ByteString` in the return type of `encryption_attrs(...)`. That function retrieves `encryption_key` from `base64.b64decode()`, which is of type `bytes`. The actual type of `encryption_key` is `bytes` as expected by the `Fernet` constructor.

Out of the 8 false positives, 7 were caused by a None check immediately preceding the call that passed the variable of type `Optional[` *type*`]` as an argument. `Optional[` *type*`]` featured prominently in ARG-TYPE errors as well; there were 647 errors that involved `Optional[` *type*`]`.

The last false-positive error was 'Argument 1 to "to_bytes" of "int" has incompatible type "object"; expected "int"' for

```
1 int.to_bytes(o['valueSat'], 8, byteorder='little')
```

where `o` is the following dictionary:

```
1 { 'address': 'XmqUtfzxgSx7WzYkEd14ug2UrJgaCmANzV',
2   'valueSat': 1664710 }
```

As shown in Sect. 2, MyPy infers type `object` for the dictionary element which is the join of all element types. It cannot distinguish that the reference element at `'valueSat'` is indeed `int`, as expected. This is expected behavior of a static type system though.

4.2.4 RETURN-VALUE. There are 1,399 RETURN-VALUE errors. We sampled 15 and found 1 false positive, 0 likely runtime errors, and 14 incorrect type annotations. Those 14 were caused by a mismatch between the returned value and the return type annotation. For example:

```
1 def load_proper_noun_data() -> List[str]:
2   ...
3   ret = set()
4   ...
5   return ret
```

triggers type error 'Incompatible return value type (got "Set[Any]", expected "List[str]")'. It is unclear whether any of these will trigger a runtime error, it depends on the expectations of the caller. We observed several cases where the return annotation was incorrect and was ignored by the code

[11] https://mypy.readthedocs.io/en/stable/common_issues.html#redefinitions-with-incompatible-types

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

```
new_tree = ET.ElementTree(new_root)
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

```
optimize_all(expression, None,
             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:

```
dotfile = NamedTemporaryFile(suffix='.dot')
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 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]
```

Table 6. Summary of hand-examined error reports.

	false positives	true positives	
		likely runtime errors	incorrect annotations
MyPy	52 (49%)	29 (28%)	24 (23%)
PyType	34 (44%)	32 (42%)	11 (14%)

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`:

```
arglist = get_arglist(I_type, T_type)
if T_type is None:
    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.

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. Tab. 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 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 or 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:

```
1 values1 = map(repr, result[0].values())
2 values2 = zip(labels, values1)
3 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:

```
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:

```
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:

```
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

[12]see https://en.wikipedia.org/wiki/Pylint and https://www.slant.co/topics/2692/~best-python-code-linters for a summary.
[13]Only MyPy and PyLint can report more-precise positions.

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 unsupported-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 $\boxed{1}$), 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 $\boxed{8}$) 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, 535 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:

```
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.[14] 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.[15] The second cause is that MyPy does not type-check unannotated functions. E.g.,

```
def g(b):
  return 1
def f(a:int) -> str:
  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.

[14]As a side note, MyPy automatically upgrades the type of parameter a to an Optional[int]. Had a been a local variable, the assignment of None would have been a type error, as in Sect. 2.

[15]They recognize this in **https://github.com/python/mypy/issues/2350**

Operator: There are 32 matching pairs between the 617 MyPy operator errors and 2,525 PyType unsupported-operands. The overlap is relatively small due to the difference in the inference semantics. Apart from unary and binary operations, unsupported-operands also checks if the variable type supports indexing, which map to 105 out of 820 MyPy's index errors. For example

```
1 def __ne__(self, other):
2   if hasattr(other,"__getitem__") and len(other)==2:
3     return self.x != other[0] or self.y != other[1]
4   else:
5     return True
```

PyType has inferred that other has type int. Although Line 2 prevents invalid indexing, PyType still reports a unsupported-operands false positive error at Line 3. MyPy does not check this function because it is unannotated.

Arg-type: There are 345 matching pairs between 1,554 MyPy arg-type errors and 4,657 PyType wrong-arg-types errors. Both categories detect actual argument types that are incompatible with the corresponding formal parameter type. Similar to other deep semantic errors, the difference in the number of errors is due to the different inference semantics, and MyPy not checking unannotated functions. The large number of PyType wrong-arg-types errors is partly because PyType delays the error report until the object is actually used; when the object is passed as an argument, it is used and PyType flags the error.

Var-annotated: There are 2,743 MyPy var-annotated errors but only 6 of them match with PyType errors. (var-annotated is a shallow semantic error, but since it is one of MyPy's most frequent errors, we have included it in Fig. 8.) The 6 matches are unrelated to the var-annotated error on the same line. As shown in Sect. 2, MyPy reports var-annotated when an empty list or dictionary is assigned to a new variable without annotation; PyType does not consider this an error.

assignment, list-item, and dict-item: As discussed in section Sect. 2, PyType allows changing the type of a variable and the type of a collection element. Thus, these three MyPy error codes have no matching PyType categories.

5.1.2 Shallow Semantic Errors. There is little overlap between MyPy's and PyType's shallow semantic errors, as shown in **Py3TypeInTheWildDLS20** (graph $\boxed{3}$ and $\boxed{4}$). 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.

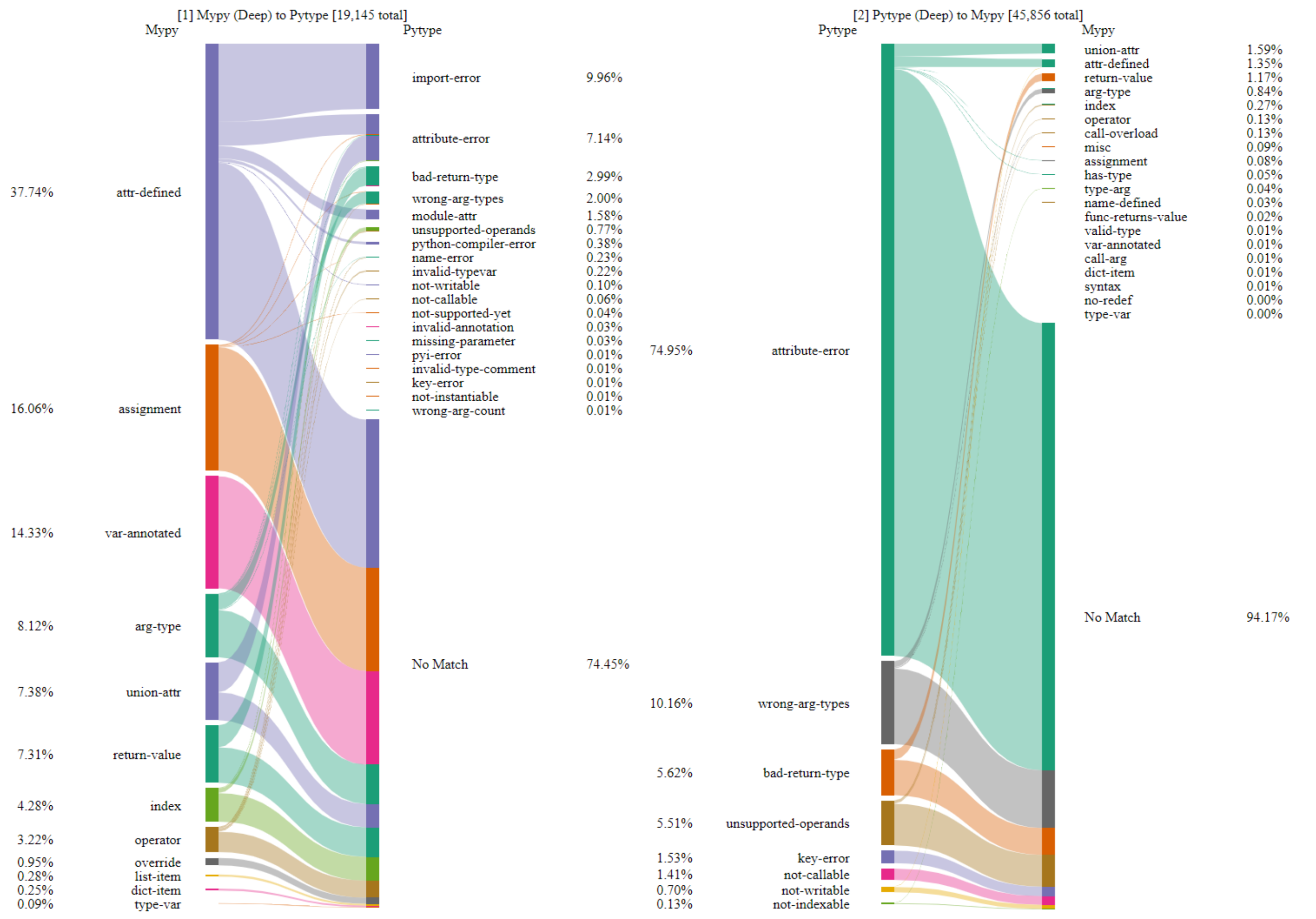

Figure 8. Comparison between MyPy's and PyType's errors that appear on the same line. Left: matches between MyPy's 19,145 deep semantic errors and PyType. Right: matches between PyType's 45,856 deep semantic errors and MyPy.

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:

```
def __init__(self, title: string,
             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 IMPORT-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

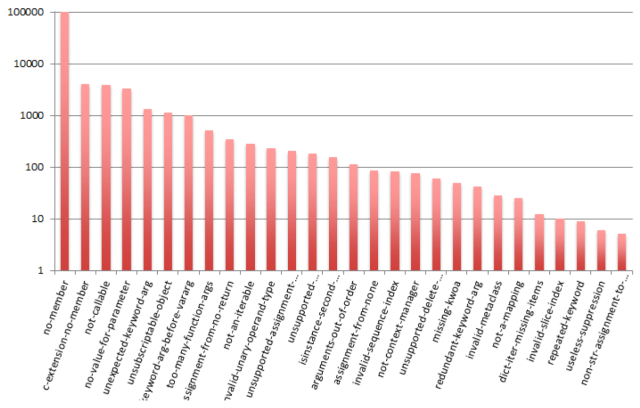

Figure 9. PyLint error code distribution of TYPECHECK subcategory across 2,673 Python-3 style annotated repositories.

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[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 $\boxed{9}$) 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.

[16] https://github.com/PyCQA/pylint/issues/3512

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 $\boxed{14}$) 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 [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 does not focus on Python 3 types; it pre-dates PEP 484 [18]. Richards et al. analyze runtime traces of JavaScript programs and find that dynamic objects and `eval` are rampant [16]. Bierman et al. define a core calculus for TypeScript, which adds types to JavaScript via a compile-and-erase approach [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 [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 [4] is a Python subset that was initially designed for PyPy [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 [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 [19]. Having been published in the same year as PEP 484 [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]. TypeWriter infers 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

[1] 2020. Infer: Eradicate. https://fbinfer.com/docs/eradicate/

[2] 2020. The Checker Framework. https://checkerframework.org/

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Conference on Programming Language Design and Implementation (PLDI)*. https://arxiv.org/abs/2004.10657

[4] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/1297081.1297091

[5] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-based Null Safety for Java. In *Foundations of Software Engineering (FSE)*. 740–750. https://doi.org/10.1145/3338906.3338919

[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science and Engineering (CISE)* 13, 2 (2011), 31–39. https://doi.org/10.1109/MCSE.2010.118

[7] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference for Object-Oriented Programming (ECOOP)*. 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[8] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Workshop on Machine Learning and Programming Languages (MAPL)*. 1–10. http://doi.acm.org/10.1145/3211346.3211349

[9] Levin Fritz and Jurriaan Hage. 2017. Cost versus Precision for Approximate Typing for Python. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*. 89–98. https://doi.org/10.1145/3018882.3018888

[10] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Conference on Computer Aided Verification (CAV)*. 12–19. https://doi.org/10.1007/978-3-319-96142-2_2

[11] A. Holkner and J. Harland. 2009. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*. 17–25. https://crpit.scem.westernsydney.edu.au/abstracts/CRPITV91Holkner.html

[12] Eva Maia, Nelma Moreira, and Rogério Reis. 2011. A Static Type Inference for Python. In *Workshop on Dynamic Languages and Applications (DYLA)*. http://scg.unibe.ch/download/dyla/2011/dyla11_submission_3.pdf

[13] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language. In *European Conference for Object-Oriented Programming (ECOOP)*. 104–131. https://doi.org/10.1007/978-3-642-31057-7_6

[14] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. 201–212. https://doi.org/10.1145/1390630.1390656

[15] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Foundations of Software Engineering (FSE)*. "https://doi.org/10.1145/3368089.3409715"

[16] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*. 1–12. https://doi.org/10.1145/1806596.1806598

[17] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Dynamic Languages Symposium (DLS)*. 944–953. https://doi.org/10.1145/1176617.1176753

[18] Guido van Rossum, Jukka Lehtosalo, and Lukasz Langa. 2014. PEP 484 – Type Hints. https://www.python.org/dev/peps/pep-0484/

[19] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Dynamic Languages Symposium (DLS)*. 45–56. http://doi.acm.org/10.1145/2661088.2661101

[20] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Foundations of Software Engineering (FSE)*. 607–618. http://doi.acm.org/10.1145/2950290.2950343