Typechecking Python

Nathan Medros
medros@oxy.edu
Occidental College

1 Introduction and Problem Context

A large part of writing software is ensuring that the software is as correct and error-free as it can. While in simple scripts this can be fairly easily done by hand, the problem becomes exponentially harder as the size and complexity of the project increases. There may be certain states that the program can reach that are rarely encountered or near-impossible to find. Ideally, programmers would like to find these errors quickly and easily before they occur.

Finding these errors is a process known as program verification and the process of verifying programs without actually running the code is called static analysis. One specific type of error that is hard for humans to find but easy for computers to check are type errors. Type errors are generated when a program expects to perform valid operations on one type of data but is passed a different type of data for which those operations are invalid. A program that performs static analysis to search for type errors is called a typechecker. Here, I present a typechecker for the Python programming language.

While some languages have strong type systems that lend themselves well to typechecking, Python does not. However, Python consistently ranks at the top of the list of the world's most-used programming languages [13]. It is used throughout a wide range of industries and use cases from ML/AI to web development to embedded systems and far, far more. Due to this high level of adoption and use, it is therefore useful to have specialized tools to check for errors in Python code. The flexibility of Python's type system and its popularity and permeation makes it a particularly interesting problem to attack.

2 Technical Background

2.1 Types and Type Systems

Depending on who you ask, the question of what specifically a type is has various closely related answers. For our purposes, in the context of a program, a variable's **type** is both the range of values that it can assume [2] and a description of what the variable can do [10]. For example, the `Boolean` type can either be `True` or `False` and it can perform logical operations such `and` and `not`. This range of values depends on the context in which the type lives. For example, in some programming languages (such as Python), the `Integer` type matches the mathematical definition of an integer (i.e., $\mathbb{Z}$). However, in other programming languages (such as C), the `Integer` type can only assume whole number values in the range $[-1 \cdot 2^{31},\ 2^{31} - 1]$ (assuming a 32-bit integer).

Once we have types, we must then construct how to use and apply them. To do this, we have **type rules**. Type rules tell the language what you can do with a type and where and when you can do it. Breaking type rules results in a **type error**.

A **type system** is simply a collection of type rules. A programming language's type system tells the computer how to assign meaning to the various constructs in the language and, once that meaning is assigned, what you can do with them. The process of checking whether or not a program conforms to the rules of a type system and does not contain type errors is called **typechecking**. Not all programming languages have type systems, however. Notably, assembly is an untyped language. Everything in assembly is just bits and bytes and it is completely up to the programmer what can be done with them. Additionally, not all type systems are created equal. There is a wide range of strictness, methodology, and execution in various type systems.

A very relevant distinction to make is the difference between **static** and **dynamic** type systems. A programming language with a static type system may check the code for correctness before it is run, usually during a compile step. A dynamic type system on the other hand contains code that operates while the program is running that checks for correctness [2]. A good example of a dynamic language is JavaScript, which by default runs without any typechecking step. It is perfectly valid in JavaScript to assign a variable to an number and then later reassign it to a string. Conversely, TypeScript adds static typing to JavaScript. TypeScript is not usually run on its own but rather transpiled to JavaScript. It is during this transpilation step that typechecking is performed.

Listing 1: Two classes with the same structure

```
class S:
    x: int
    def f(): ...

class T:
    x: int
    def f(): ...
```

2.2 Comparing Types

When writing a system to check types, it is natural to first define the notion of comparing types. In order to do this, we first want to know what within the system defines a type. The two main methods of doing this in static type systems are through structural and nominal typing. In a system with **structural typing**, types are determined through their structure. That is, types are determined by the data that they hold. In contrast, a system with **nominal typing** determines types through their definition. A nominal type system will not consider two types the same unless they are explicitly defined to be.

Consider the code in Listing 1. Suppose we have two variables `x:T` and `y:S`. In a structural type system, we would have that `type(x)` $=$ `type(y)` as the structure of the types are the same, despite the fact that they were defined in two separate instances. However, in a nominal type system we have that `type(x)` $\neq$ `type(y)` as even though the types can assume all the same values and have all the same operations done on them, they were not explicitly defined to be the same so they are not.

Once we know how a system defines types, we can then define the subtype, supertype, and type equivalence relations. A type `S` is a **subtype** of another type `T` (`S` $<:$ `T`) if a term of type `S` can be safely used anywhere a term of type `T` can be used. This is known as the **safe substitution principle** [8]. A type `Y` is a supertype of a type `X` (`Y` $:>$ `X`) if `X` is a subtype of `Y`. Two types `A` and `B` are equivalent if `A` $<:$ `B` and `B` $<:$ `A`.

Many type systems contain notions of a **bottom type** and a **top type**. A bottom type is a type is a subtype of all other types. Conversely, a top type is a type that is a supertype of all other types. Mathematically speaking, the bottom type confers a meaning of falsehood to the thing it is applied to. For example, a function that returns the bottom type will never return. The top type acts a universe for the types within a system. Since all types are subtypes of the top type, the top type encompasses the least properties that all constructs in the language can be said to have. Furthermore, having top and bottom types are useful for type inference, allowing untyped variables to have an initial range of values that can be assumed [7].

Many type systems also have a **unit type**. A unit type is a type with a single term. That is, there is only ever a single value that has the type of the unit type.

2.3 Python's Type System

2.3.1 Gradual Typing

By default, Python is a dynamically typed language. Programmers can – and often do – write Python without needing to think about types. However, Python contains optional static typing through type annotations. This static type system must be enforced through an outside program as the Python interpreter ignores annotations. This means that programmers are allowed to freely mix static and dynamic types in the same program, allowing for both the flexibility of dynamic typing when it is needed and the correctness of guarantees of static typing when it is wanted. To reconcile the differences between the two, Python formally uses a method called **gradual typing** which is somewhere between fully static and fully dynamic typing.

Gradual typing allows for some variables in a program to be statically typed while leaving other types unknown. This allows for a combination of static and dynamic typechecking. Variables for which types are known before the program is run can be checked statically while types with unknown variables can be dynamically checked at runtime [6]. In Python, this unknown type is denoted `Any`. It is important to note, however, that `Any` is not a type in and of itself but rather the explicit lack of a type.

In order to integrate these gradual types into the wider type system, we must define the concepts of materialization and consistency.

A type `S` is a **materialization** of another type `T` if `S` results from replacing some occurrences of `Any` in `T` with a static type. Materialization is a reflexive relation, so every type is a materialization of itself. In a sense, if `S` is a materialization of `T`, then `S` is more precise than `T` [3]. As an example, the Python types `List[int]` and `List[Dict[str, bool]]` are both materializations of `List[Any]`.

Two types `A` and `B` are said to be **consistent** if there is a type `C` such that `C` is a materialization of `A` and `C` is a materialization of `B`. In the case where `A` and `B` are both fully static types, then they are consistent if and only if they are equivalent as fully static types have only one materialization, namely themselves [5]. The gradual Python types `List[Dict[str, Any]]` and `List[Any]` are consistent as there is a type that is a materialization of both of them like `List[Dict[str, int]]`.

We may finally define a third relation, called **consistent subtyping**. A type `A` is a consistent subtype of `B` if there exists a fully static materialization $\widetilde{A}$ of `A` and a fully static materialization $\widetilde{B}$ of `B` such that $\widetilde{A} <: \widetilde{B}$ [5].

The importance of all of this is that Python defines assignability in terms of consistent subtyping. If a variable has a type `T` or a function has an argument of type `T`, then any value with a type that is a consistent subtype of `T` can be used [12].

2.3.2 Types in Python

Everything in Python is an `object`. `objects` are pieces of data with attributes (data stored within the object) and methods (operations that can be done on the object). Further distinctions between data in a Python program results from the attributes an object has. For example, a `Callable` is an object that has a `__call__` method. Functions are thus objects with a `__call__` method. However, classes implicitly define `__call__` method in their constructor so they are also `Callables`. The distinction between various types in Python is a blurry one. One can define a `class` (itself an object) in such a way that its instance (which is also an `object`) can be treated as any other `object` in the language.

These special properties of various `objects` are constructed using **dunder** methods, so called because they start and end in double underscores. There are many dunder methods in Python such as `__call__` which we previously saw. Other common ones include `__len__` for `objects` that can be passed into the `len()` function and `__add__` for `objects` that are valid inputs to the binary + operation [1].

A result of this flexibility is that Python somewhat erases the line between nominal structural subtyping. Formally defining a class as inheriting from another class just means applying the previous class objects methods and attributes to the new one. One can explicitly define a class as `Sized` or you can just implement the `__len__` method. Both are valid.

2.3.3 Special Types in Python

Python contains a few special fully static types.

The `object` type acts as the top type for the language. All types are subtypes of the `object` type. A common misconception is that `Any` is also a top type in Python. This is wrong as `Any` is not a type but rather it denotes a lack of type.

Python has two bottom types, `Never` and `NoReturn`. These types are equivalent however `NoReturn` is used as a function return type and `Never` is used elsewhere. A function that has a return type of `NoReturn` should never return. This is usually used in functions that would cause the program to exit or raise an exception. A variable or function argument of `Never` means that nothing should ever be passed into it. One might use `Never` to hint to a typechecker that a condition should never be reached and if it is, then it returns a type error.

Python also contains a unit type `NoneType`, usually abbreviated with its sole term `None`.

Finally, Python also supports union types, which denote the ability of a type to contain multiple values. For example, if `x: int | str` then `x = 3` and `x = "Hello"` are both valid. The optional type is a special case of a union with `None` and is especially useful for values that have conditional initializations.

3 Prior Work

3.1 Theoretical Work

A rich literature exists on the subject of types, typechecking, and type systems. However, much of this work is theoretical and is based upon generalizations of a model of computation known as simply-typed lambda calculus. While these theoretical models match up well with some programming languages, notably Haskell, ML, and Coq, their correspondence with Python is not always clear. These models tend to view programs and data in terms of primitive types and function types and they provide operations with neat mathematical properties that can be used to create new types and reason about types. While Python is able to fit into these models, the inherent flexibility of the language makes it an uncomfortable correspondence. With that being said, there are three theoretical ideas that I've incorporated into this project.

3.1.1 Gradual Typing

As mentioned before, Python uses a gradual type system. This is an idea that has been discussed to a fair extent in the literature. The core difference between previous static and dynamic type systems is the addition of the unknown type (usually denoted ?) which lies outside the type hierarchy. The original idea was introduced in 2006 by Siek and Taha [11] however it lacked a concept of materialization. Rather, it simply left unknown types as unknown and left it up to the language to dynamically check these types at runtime. The concept was more fully developed by Garcia et al. in 2016 [5] who added the concept of materialization. This allowed for a more formal definition of consistency between types, providing a full formal gradual type system on which to base future research.

3.1.2 Bidirectional Typechecking

The core inspiration for the underlying algorithm of my typechecker is bidirectional typechecking as laid out in [4]. The idea behind it is that in many cases, typechecking can

be split into two separate processes: type checking (i.e., going from type to value) and type inference (i.e., going from value to type). The paper highlights the visual similarities between bidirectional typechecking and gradual typing however there is a crucial difference. Bidirectional typechecking modifies context based on information known during the typechecking step while gradual typing modifies context based on runtime information. While the effects of those two appear similar, this crucial difference of where the modification is happening makes a difference.

3.1.3 λ_π

While I did not incorporate many of their ideas into my project, λ_π, an operational semantics for the Python language [9]. This paper is unique in that it attempts to define a full theoretical model to interpret the Python language. Specifically, their treatment and explanations of what makes certain aspects of Python difficult to formalize was a very helpful reference during the initial stages of development and planning.

3.2 Other Typecheckers

Mine is not the only typechecker that has been implemented for Python, there are four main competing typecheckers in production use today: MyPy, Pyright, Pyre, and Pytype.

Mypy is written in Python and is itself a project of the Python Software Foundation and is developed alongside CPython. It is considered the reference implementation for Python typechecking as it implements exactly what is described in the Python Type Specification [12].

Pyright is written by Microsoft and emphasizes high performance. It is also written in Python however it is designed to be used both as a standalone command-line tool as well as a Visual Studio Code extension. Pyright is notable for being the most feature-complete of the four [14].

Pyre is written by Meta in OCaml. It also emphasizes high performance and it contains some unique features specific to machine learning and AI projects. Its most notable feature is that it is bundled with Pysa, which extends Python's type annotation system to check for security vulnerabilities. For example, it allows the programmer to indicate whether the inputs and outputs to a function are trusted or untrusted.

Finally, Pytype is written by Google in Python. It does three things that distinguish it from the other typecheckers mentioned here. First, it disregards gradual typing in favor of type inference. Rather than leave untyped terms untyped, it will attempt to guess the type of the term. Additionally, it generates type annotations for untyped code. Finally, it is intended to be as lenient as possible. If a piece of code doesn't directly contradict annotated types and won't result in a runtime error, Pytype won't raise an error.

4 Methods

My typechecker is written in Python. The primary purpose of this is to take advantage of Python's standard `ast` library which takes care of parsing the input into an AST. This allowed me to focus on the actual typechecking rather than parsing the language.

I have attempted to stay as true to the theoretical underpinnings of Python's type system as possible. At its core, my code looks at what type is being given and what type is expected and attempts to find a consistent subtyping relationship between them. If none can be found, I emit an error. I do, however, also attempt limited type inference following the theoretical framework of bidirectional type checking. When it js possible to unambiguously determine all possible values that an unannotated term can hold, my code will infer a conservative least type for that variable and then proceed to use that in future judgments.

I visit the AST nodes via depth first search. The order in which the nodes are visited is context-dependent – by default, nodes are visited left-to-right (that is, the order in which they appear in the source code) however this is not done in all cases. Variable assignments are checked right to left, with the right-hand side of the assignment being checked first so that its type can be inferred. This way, we have all the information needed to either infer or check the type of the left-hand side. Child nodes are always checked before their parent node so as to have complete information.

The typechecker has two main data structures, `Context` and `Variable`. Each `Variable` represents an object in the code. Types, functions, and normal variables are all stored using the `Variable` structure. This reflects how Python internally sees them. `Variable`s store their name, scope, and type. Their type is a reference to another instance of a `Variable` with the `Variable` representing the type `type` being self-referential.

The specifics of various types of types are stored in an optional internal variable in the `Variable` data structure as a subclass of a `Type` base class. There is a `Callable` type inheriting from the `Type` base class that stores function arguments. Additionally, there is a child class that implements `Union`s. Lastly, there is a `Any` class that represents the gradual type. To capture the consistent subtyping relationship between types without adding much extra work, the `Any` type is only ever equal to itself. For any non-`Any` type it compares as both a subtype and supertype, but not an equivalent type.

The `Context` class stores information about where the typechecker is and what it has already seen. The `Context` class contains things such as the current scope, the filename,

and all information necessary for qualified names (function name, class name, etc). It also contains information about the return type of the current scope to aid in checking functions. Most importantly, it contains the lookup table of all values in scope which can be used to make judgements. This includes previously defined variables and types as well as built-in types and functions.

5 Evaluation Metrics

There are three ways through which I am evaluating my typechecker. The first is through hand-crafted test cases. While this is the least comprehensive of the methods, it is the most granular and it lets me easily make sure that the typechecker is catching all the errors I intend it to. The second is through tests that other people have written. In particular, PyType contains a wealth of concise tests that test for specific errors and Pyright has a massive collection of over a 1,000 test files. These test cases are not only far more numerous than I could ever write myself, but they are also battle-tested and contain errors and edge cases that others have encountered. Finally, I also wanted to test my code on real-world scripts. Due to the limitations of my project, namely the lack of import support, it was hard to find many representative samples. However, I was able to run it on some of the helper scripts I made during the course of the project. While this doesn't yield many measurable results, it does serve as a proof of concept for real-world applications.

For the first two test methods, the metric of evaluation concerns the number of false positives, the number of false negatives, and the number of incorrect judgments.

One evaluation method that I considered was fuzzing. While fuzzing would be the most comprehensive possible test suite, I ultimately decided not to go with the idea. In order to fuzz test my project, I would first need to come up with a way of programmatically constructing valid and meaningful Python programs. This is not a trivial task and could itself be an entire comps project. Constructing Python code is itself not a hard problem given Python's helpful AST data structures provided in the standard library and its `ast.unparse` method however having these programs be meaningful in a helpful way is a nontrivial task. While I may be able to put the right kind of nodes in the right places, a program contains more than grammar. In the same way that inserting random nouns, verbs, and adjectives into English in the right order does not necessarily create a meaningful sentence, fuzzing would generate a large volume of test cases however very few of them would be actually worth testing. With that being said, a possible avenue for future testing in this space could be AI code generation, which is not only capable of generating meaningful programs but also is generating them through its test data, consisting of a large body of real-world code and thus real-world errors.

6 Results and Discussion

6.1 Goals

I had four main goals for my typechecker:

1. Check the consistency of statically annotated types in Python programs. Specifically, my goal is to check types annotated with built-in types and classes derived from built-in types.
2. Have an understanding of scope and locality.
3. Conservatively infer types in places without explicit static type annotations and there is a clear type capable of being inferred. For example, when given `a = 3`, the typechecker should be able to infer that `a` is an `int`.
4. Understand gradual types and forms and check for assignability using consistent subtyping as criteria.

6.2 Non-Goals

Additionally, I had a few features of Python that I considered non-goals of my project:

1. Imports and modules
2. Literal types
3. Classes derived from the `typing.Generic` base class.
4. Asynchronous functions.
5. Manual manipulation of dunder methods and attributes (e.g. `__slots__`)

6.3 Current Results

My typechecker is currently capable of checking Python programs for a variety of type errors. It is capable of walking through an AST and checking annotated assignments as well as inferring types for unannotated assignments with primitive types. It is capable of checking user-defined functions and understands scope in function bodies. It can also perform simple return type checks as well as check default arguments. Union types are fully implemented, including the `Optional` type which is defined as a Union with None. Finally, my typechecker has a rudimentary understanding of the gradual form `Any` and is capable of checking it when it appears on its own as well as when it appears in partially-defined types such as callable.

With that being said, there are things that I was unable to do. Firstly, I was unable to implement classes nor was I able to implement certain important types such as `list` and `dict`. Furthermore, due to this feature-incompleteness I was unable to perform rigorous testing beyond hand-written test cases.

6.4 Discussion

The main reason that I was unable to implement classes and thus `lists` and `dicts` was that early on in development, I had assumed that classes, functions, and primitive types could be implemented separately and thus I organized my data around that assumption. This, of course, was not a good assumption to make. As mentioned previously, Python views everything as `objects`. One consequence of this is that the lines between any constructs within the language are incredibly blurry. Classes, when called as type constructors, are functions. In addition, primitive types such as `int` and `str` are also classes with their own methods. As I began implementing more and more, I began to spend more time implementing special cases for each type of type. In addition, because I was treating all of these types as separate, I had to manually compare every type with every other type which, besides being tedious, was an error-prone process.

6.5 Future Work

As of writing, I am currently working on reworking the way that types are stored to fit more closely with the way Python sees them. Rather than having separate classes for different types of types, I am instead taking a field-based approach where types (and, indeed, all constructs in the language) are determined by their methods, attributes, and the values they hold. This greatly simplifies things, especially in terms of comparisons. Instead of having to have special cases for type constructors, built-in functions, and regular functions when checking function definitions, I can simply check their common `__call__` field. In addition, structural types now become a simple containment check, rather than the more complicated process they would be previously. In addition, this method will increase the flexibility of the system, allowing for the easier implementation of features I had previously not considered adding such as imports, modules, and literal types.

7 Ethical Considerations

Our modern world is highly controlled by computers and the software inside of them. Thus, software errors can easily become real-world problems causing millions of dollars worth of damage to equipment and systems and causing bodily injury or even death. For critical systems, it is thus important to use tools that can provide formal guarantees of correctness, such as typecheckers. However, should these programs themselves contain bugs, it could result in the unintentional release of error-prone software.

As a program that is intended to help catch errors in other programs, errors or incorrect judgments in the typechecker

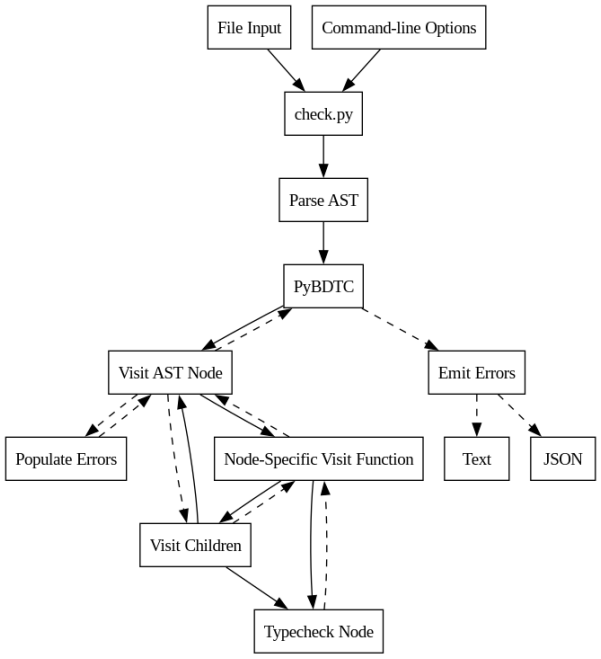

Figure 1: High-level flow of data. Solid arrows denote context and AST information while dotted arrows denote error information.

itself can easily propagate into other software and increase in severity. To mitigate this, I have made my code open source, allowing anyone to inspect its logic and programming and verify its correctness for their use case. In addition, I make no guarantees of totality or correctness, and I have publicly stated the design limitations of the typechecker and what is is supposed to be able to check and what it is not supposed to check.

A Replication Instructions

The code for this project can be found at `https://github.com/ndm767/PyBDTC`. The project requires and targets Python 3.12. Its only other external dependency is the `colorama` package for colored terminal output which can be acquired using pip or the package manager of your choice using the supplied `requirements.txt`. The project can be run using the command `python3 check.py [input_file]`. An input file with no type errors will not produce any output. Debug options relating to viewing the internal state of the typechecker can be found by running `python3 check.py --help`.

B Code Architecture

A diagram of the code architecture is shown in Figure 1. The core of the program are the `visit_*` functions. Each AST node is passed to a top-level `visit` function which then dispatches it to a node-specific visit function. For example, annotated assignments (`ast.AnnAssign`) are dispatched to `visit_AnnAssign`. These functions are defined in the `pybdtc.nodes` folder where they are sorted based on the categories used in the Python `ast` documentation. These visit functions may then recursively call the top-level `visit` function on child nodes.

There are two main data structures that control the flow of information in the project. The `Context` data structure flows "down" from parent node to child node. New variables are added to the context when they come into scope. When a new scope is reached, a copy of the current context is made which is then modified by the child nodes. After the child nodes return, the old context returns, deleting everything that is no longer in scope.

The `Errors` data structure flows "up" from the child nodes to the parent nodes. Each `visit` function has its own `Errors` data structure that it hen adds its errors to. This data structure is then returned by the `visit` functions where it is then merged into the parent node's visit functions. When it is possible to do so, the errors are populated with filenames and line numbers to provide context as to where the error is occurring. When the program terminates, the final `Errors` data structure contains all of the consolidated type errors encountered in the program and may emit them in the format of the user's choice.

References

[1] *3. Data model*. en. URL: https://docs.python.org/3/reference/datamodel.html.

[2] Cardelli, Luca. "Type Systems". In: *Computer science handbook*. Ed. by Tucker, Allen B. Chapman and Hall/CRC, 2004.

[3] Castagna, Giuseppe et al. "Gradual typing: a new perspective". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 16:1–16:32. DOI: 10.1145/3290329. URL: https://dl.acm.org/doi/10.1145/3290329.

[4] Dunfield, Jana and Krishnaswami, Neel. "Bidirectional Typing". en. In: *ACM Computing Surveys* 54.5 (June 2022), pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3450952. URL: https://dl.acm.org/doi/10.1145/3450952.

[5] Garcia, Ronald, Clark, Alison M, and Tanter, Éric. "Abstracting gradual typing". In: *ACM SIGPLAN Notices* 51.1 (2016), pp. 429–442.

[6] jsiek. *What is Gradual Typing — Jeremy Siek*. en-US. Mar. 2014. URL: https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/ (visited on 12/14/2024).

[7] Pierce, Benjamin C. *Bounded Quantification with Bottom*. Tech. rep. 492. Computer Science Department, Indiana University, 1997.

[8] Pierce, Benjamin C. et al. "Sub: Subtyping". In: *Programming Language Foundations*. Software Foundations series, volume 2. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf. Electronic textbook, May 2018. URL: https://softwarefoundations.cis.upenn.edu/plf-current/Sub.html.

[9] Politz, Joe Gibbs et al. "Python: the full monty". en. In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. Indianapolis Indiana USA: ACM, Oct. 2013, pp. 217–232. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509536. URL: https://dl.acm.org/doi/10.1145/2509136.2509536.

[10] Rémy, Didier. "Type systems for programming languages". In: *MPRI lecture notes* (2013), p. 64.

[11] Siek, Jeremy G and Taha, Walid. "Gradual Typing for Functional Languages". en. In: (2007).

[12] *Specification for the Python type system — typing documentation*. URL: https://typing.readthedocs.io/en/latest/spec/index.html.

[13] *TIOBE Index (Python)*. en-US. URL: https://www.tiobe.com/tiobe-index/python/.

[14] *Type System Test Results*. URL: https://htmlpreview.github.io/?https://github.com/python/typing/blob/main/conformance/results/results.html.