Bachelor's thesis

Bachelor's Programme in Computer Science

Python Type Annotations: Adoption and Benefits

Kerkko Pelttari

July 29, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Bachelor's Programme in Computer Science

Tekijä — Författare — Author

Kerkko Pelttari

Työn nimi — Arbetets titel — Title

Python Type Annotations: Adoption and Benefits

Ohjaajat — Handledare — Supervisors

Prof. Dorota Glowacka, Dr. Yang Liu

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Bachelor's thesis	July 29, 2025	15 pages, 1 appendix pages

Tiivistelmä — Referat — Abstract

This thesis is a literature review on Python type annotations, their use and benefits. Background on Python, type systems, type annotations and type checking is provided. Then the relevant literature is discussed and analysed. The target audience is users and researchers of Python.

Python is a widely utilised general purpose programming language. Its type system is strong and dynamic. In 2015 support for optional type annotations was implemented. Type annotation's effectiveness in improving program correctness is the main motivation for this study.

Python is well-known as simple, concise and readable. Type annotations may potentially conflict with these properties; however, the gradual and optional features of type annotations let developers implement type annotations where they provide the most value.

Type annotations have risen in popularity from 2015 to 2022. Nevertheless, only a minority of Python source code utilises type annotations. Python type inference and checking has been integrated into code editors, and this helps developers work with both typed and untyped source code.

Existing research has determined that it is possible to find approximately 11% software defects (bugs) by utilizing type annotations. Type annotations can be utilised to document public interfaces and to improve correctness in selected parts of a software.

ACM Computing Classification System (CCS)
Software and its engineering → Software notations and tools → General programming languages → Language features

Software and its engineering → Software creation and management → Software development techniques

Avainsanat — Nyckelord — Keywords

Python, Type systems, Type annotations, Type checking, Gradual typing

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Lyhennelmä

Johdanto

Python on laajasti käytetty ohjelmointikieli. Vuonna 2015 siihen lisättiin tuki valinnaisille tyyppivihjeille, jotka mahdollistavat uudenlaisen tyyppitiedon lisäämisen lähdekoodiin.

Pythonin kasvava suosio ja Pythonin tyypitysratkaisujen kasvava suosio nostavat esiin ristiriidan: Miksi käyttää dynaamisesti tyypitettyä kieltä ja tyypittää koodi staattisesti?

Staattisissa kielissä ohjelma tyypitetään kääntäessä. Dynaamisessa kielessä tyypit päätetään ajonaikaisesti. Valinta näiden välillä vaikuttaa siihen milloin tyyppivirheitä havaitaan, ja siirtää oikeellisuuden varmistamista käännös- ja ajonaikaisen välillä.

Staattisen tai dynaamisen ohjelmointikielen valinnalla vaikuttaa joskus olevan tilastollisesti havaittava vaikutus ohjelmakoodin oikeellisuuteen [15, 20]. Kuitenkin tulos paljon viitatusta vuoden 2014 tutkimuksesta [20] ei toistunut vuonna 2019 suoritetussa toistotutkimuksessa [1]. Havaitaan että ohjelmointikielillä kirjoitettujen ohjelmien oikeellisuuden vertailu on vaikeaa, sillä ohjelmointikielillä on monenlaisia eroja jotka tekevät oikeellisuuserojen vaikutuksen eristämisestä haastavaa.

Python tyyppivihjeillä on vähittäisesti tyypitetty ohjelmointikieli (tyypitetty Python). Sillä kirjoitetut ohjelmat omaavat sekä staattisen että dynaamisen kielen piirteitä. Vähittäisen tyypityksen ratkaisuista, kuten JavaScript ohjelmien tyypityksen laajentamisesta TypeScriptillä, voidaan havaita tilastollisesti merkittäviä parannuksia ohjelmointivirheiden (bugien) havaitsemisessa. [6]. Tyyppivihjeillä varustetulle Pythonille on löydetty lupaavia tuloksia bugien havainnoinnista tyypittämättömään Pythoniin verrattuna [10, 19]

Tyyppijärjestelmät

Tyyppijärjestelmät antavat organisoida ja kategorisoida ohjelman tilaa, informaatiota, funktioita, ja näihin liittyviä rakenteita [5]. Niiden avulla ohjelmointikieli voi antaa takeita ohjelman toiminnasta, valita miten dataa käsitellään, ja tyyppi-informaatio voi toimia tarkistettavavana lisäinformaationa ohjelman rakenteesta.

Käytännössä kaikki korkean tason ohjelmointikielet hyödyntävät tyyppijärjestelmää. Tyyppijärjestelmien ominaisuudet, hienojakoisuus, rajoitteet ja päättelylogiikka vaihtelevat. Ohjelmointikielet voidaan jakaa erinäisiin akseleihin: vahvasti ja heikosti tyypitettyihin, ja staattisesti, dynaamisesti tai vähittäin tyypitettyihin.

Vahvassa tyyppijärjestelmässä ei sallita käsittelemättömiä tyyppivirheitä [2]. Tyyppivirheen tapahtuessa vahvasti tyypitetyn ohjelmointikielen on pakko antaa virhe ohjelmoijalle käsiteltäväksi, tai kaatua. - Heikosti tyypitetyssä ohjelmointikielessä tyyppivirheen tapahtuessa se voidaan nostaa käsiteltäväksi virheeksi, mutta yleensä suoritetaan jonkinlainen implisiittinen tyyppimuunnos. Tämä voi joskus olla käytännöllistä, mutta voi johtaa virheelliseen toimintaan jonka tarkka syy on vaikea löytää.

Staattisesti tyypitetyssä ohjelmointikielessä jokaisella symbolille on löydyttävä tai voitava päätellä oikea tyyppi tyyppitarkistuksessa, joka yleensä suoritetaan ohjelmakoodin kääntämisen yhteydessä. Teoriassa staattisesti tyypitetylle kielelle voidaan tarjota kattavampia ohjelmointityökaluja, sillä työkaluilla on enemmän informaatiota käytettävissä.

Dynaamisesti tyypitetyssä kielessä tyypit symboleille selviävät vasta suorituksen aikana. Tämä mahdollistaa kevyemmän syntaksin, jossa ohjelmoijan ei tarvitse käyttää aikaa tyyppimäärittelyihin [3].

Vähittäiset tyypitysjärjestelmät muodostavat ylijoukon sekä staattisista että dynaamisista ohjelmointikielistä [21]. Vähittäisesti tyypitetyllä kielellä voi kirjoittaa sekä täysin staattisia että täysin dynaamisia ohjelmia, ja näiden välimuotoja. Erityisenä hyötynä tämä mahdollistaa ohjelmien tyypittämisessä keskittyä hyödyllisimpiin osioihin, ja kehittää tyyppikattavuutta vähitellen [21]. Usein vähittäiset tyyppijärjestelmät toteutetuaan olemassaolevan ohjelmointikielen päälle, kuten Pythonissakin.

Python on korkean tason tulkattu ohjelmointikielen joka on tiivis ja ilmaisuvoimainen. Pythonin tyyppijärjestelmä on vahva ja dynaaminen. Sitä käytetään esimerkiksi tieteelliseen laskentaan, koneoppimiseen, verkkopalvelukehitykseen ja komentosarjaohjelmointiin (scripting).

Tyypit Pythonissa

Tyypitetyssä Pythonissa on kaksi tyyppijärjestelmää: 1) Ajonaikainen, dynaaminen, ohjelman toimintaa ohjaava järjestelmä. 2) Valinnainen tyyppivihjeisiin ja päättelyyn pohjaava järjestelmä joka lisättiin vuonna 2014 [7]. Nämä tyyppivihjeet eivät itsessään vaikuta oh-

jelman ajonaikaiseen käytökseen, mutta niitä hyödynnetään työkaluissa ja tarkistimissa.

Keskitymme tarkastelemaan tyyppijärjestelmää 2). Tyyppivihjeiden tarkistamisen vapaaehtoisuus noudattaa *vähittäistakuuta* (gradual guarantee), jonka mukaan saman ohjelman tyypitetyn ja tyypittämättömän version eroavaisuuksien tulisi rajoittua niiden tyyppeihin [21]. Pythonissa tyyppivihjeitä merkitään kaksoispisteellä muuttujille ja argumenteille `x: int` ja nuolella funktioiden palautusarvoille `def f() -> int: return 1`.

Pythonille on kolme laajasti käytössä olevaa työkalua jotka tyyppitarkistavat: mypy, Pyright, ja PyCharm. Mypy on Python Software foundationin kehittämä työkalu jota voi käyttää komentoriviltä tai integroituna kehitystyökaluihin [11]. Pyright sisältyy Microsoftin Visual Studio Code ohjelmointitekstieditoriin [14]. PyCharm on JetBrains s.r.o:n ohjelmointiympäristö, jossa on kattavat ominaisuudet Python-koodin muokkaukseen [8]. Kaikki työkaluista hyödyntävät tyyppivihjeitä ja tyyppipäättelyä tarjotakseen ohjelmoijille tukea tyyppivirheiden löytämiseen ja tehokkaaseen ohjelmointiin.

Kaikki mainitut tyyppejä tarkistavat työkalut hyödyntävät tyyppipäättelyä [8, 11, 14]. Tyyppivihjeiden lisäksi ne pystyvät siis päättelemään muuttujien tyyppejä niihin asetettujen arvojen avulla, ja palautusarvoja palautettujen muuttujien tyyppien avulla. Tyyppipäättely mahdollistaa tyypityksen hyötyjä myös tyypittämättömään koodiin, koodin täydennykseen ja virheiden havainnointiin.

Python tyyppivihjeiden tutkimus

Yleisin metodi Python tyyppivihjeiden oikeellisuuden tutkimiseen on ottaa satunnaisotanta Python ohjelmia ja ajaa tyyppitarkistuksia sitä vasten [3, 12, 19, 22]. Tämän avulla saadaan dataa tyyppikattavuudesta ja tyyppivirheistä. Kun halutaan tutkia miten hyvin tyyppitarkistimet löytävät korjattavaksi priorisoituja ohjelmointivirheitä, voidaan ottaa otanta korjatuista virheistä. Lisäämällä ohjelmointivirheen sisältävään ohjelmakoodiin tyyppivihjeet voidaan tyyppitarkistimella tarkistaa olisiko virhe näkynyt tyyppivirheiden muodossa.

Funktioiden tyyppikattavuus on noussut 0% lisäämisvuonna 2015 noin 8% vuoteen 2021 mennessä, kuten näkyy kuvaajassa 3.1. Toinen tuoreempi tutkimus selvitti tyyppikattavuutta yhdessä suuressa Python-ohjelmassa (Tensorflow) [12]. Tässä tutkimuskohteessa tyyppivihjeiden määrä nousi 5% vuonna 2019 39% vuoteen 2022 mennessä. Tyyppikattavuus on ollut selkeästi nousussa, mutta voidaan myös huomata että tyypitetyn koodin

osuus ei ollut vielä kovin suuri vuonna 2022.

Vuoden 2020 tutkimuksessa tyyppivihjeiden hyödyntämisestä havaittiin että vaikka tyyppivihjeet yleistyvät, vain 15% otannan ohjelmistoprojekteista oli virheettömiä Mypy-työkalulla tehdyssä tarkistuksessa [19]. Vuoden 2022 tutkimuksessa 9655 suosituimmasta GitHubissa olevasta Python projektista havaittiin että 78% muutoksista sisälsi tyyppivirheitä [3]. Ainakin osa löydetyistä tyyppivirheistä on vääriä hälytyksiä (false positive), mutta näiden osuudesta ei ole tietoa. Tyypillisiä syitä väärille hälytyksille on erot tyyppitarkistimessa tai sen asetuksissa.

Tyyppivihjeitä hyödyntämällä joissain tilanteissa voidaan löytää jopa 11% korjaamista vaativista ohjelmointivirheistä [10]. Toisessa analyysissä jossa otanta kohdistui tyyppeihin liittyviin ohjelmointivirheisiin, niistä 35% voitiin löytää tyyppitarkistimella ilman vihjeitä, ja 73% vihjeiden avulla.

Sekä kokeneet että kokemattomat ohjelmoijat tekevät paljon tyyppeihin liittyviä virheitä ohjelmoidessaan Pythonia [10]. Yleisesti esiintyvät virheet liittyvät muuttujien käyttämiseen ennen arvon asettamista, odottamattomiin tyhjiin arvoihin, tai muuttujien uudelleenkäyttöön niiden tyypin muuttumisen jälkeen. Koska kaiken tasoiset ohjelmoijat kohtaavat tyyppeihin liittyviä virheitä ohjelmoidessaan, tyyppitarkistamisesta voi olla kaikille hyötyä.

Pohdinta: Miksi käyttää Pythonia ja tyyppivihjeitä?

Miksi Pythonia käytetään? 1) Pythonin suunnittelutavoitteet ovat johtaneet sen kieleksi joka on yksinkertaista luettavaa, nopeata kirjoitettavaa, ja helppo opppia. Harvalla ohjelmointikielellä on yhtä korkea priorisointi kehittäjäkokemuksessa. 2) Pythonin helppous opppia tekee siitä suositun ensimmäisen ohjelmointikielen, jonka ansiosta sille löytyy paljon kehittäjiä. 3) Sille löytyy paljon hyödyllisiä paketteja, jotka tukevat laajaa määrää käyttötapauksia.

Tutkimuskysymyksessä asetettu ristiriita: "Miksi käyttää dynaamisesti tyypitettyä kieltä ja tyypittää koodi staattisesti?", ei oikeastaan kestä syvempää tarkastelua. Tyypitetty Python ei ole staattisesti tyypitetty kieli, vaan vähittäisesti. Tämä sallii kehittäjien saada sekä dynaamisen että staattisen tyyppijärjestelmän hyötyjä. Tätä indikoi tyyppivihjeiden matalahko osuus ja tyyppivirheiden esiintyvyys myös tyypitetyissä Python lähdekoodeissa [3, 19].

Ilmiötä voidaan mallintaa vertaamalla Pythonia muihin ohjelmointikieliin, ja tästä havaitaan että Python on tiivis ja luettava kieli jossa kuitenkin helposti jää ohjelmointivirheitä ajonaikaisiksi. Tyyppivihjeet sallivat ohjelmoijan muuttaa Pythonin ominaisuuksia, ja vaihtaa tiiviyttä parannettuun ohjelman oikeellisuudessa.

Yhteenveto

Pythonia käytetään useista syistä. Se on kätevän luettavaa, ilmaisuvoimaista ja helppokäyttöistä.

Tyyppivihjeet tekevät Pythonista vähittäin tyypitetyn ohjelmointikielen joka pystyy olemaan täsmällisempi, luettavampi ja itsedokumentoiva. Tyyppivihjeiden suosio onkin kasvussa [9, 10]. Tyyppitarkistuksesta on tullut helpompaa ja huomaamattovampaa, kun moni ohjelmointiympäristö oletuksena hyödyntää sitä. Tämä auttaa ohjelmoijia hyödyntämään tyyppi-informaatiota ja löytämään virheitä.

Tyypitetty Python antaa ohjelmoijien hyödyntää Pythonin ydinvahvuuksia, ja sallii vähittäisellä tyypityksellä ohjelmoijan hyötyä staattisten ja dynaamisten tyyppijärjestelmien hyödyistä.

Tyyppivihjeet kannattaa ottaa käyttöön, jos projektissa halutaan hyötyjä oikeellisuudessa, staattisessa analyysissä, ja koodin huollettavuudessa. Tyyppivihjeiden käyttöönottoon tarvitsee varata kehitysaikaa ja sitä kannattaa suunnitella. Ohjelmakoodista kannattaa arvioida millä alueilla tyypeistä on suurin hyöty ja missä ne on helpointa ottaa käyttöön. Tyyppivihjestrategian voi odottaa elävän sen mukaan kun niitä otetaan käyttöön, ja tyypitystyö saattaa paljastaa tarpeita ohjelman rakenteen parantelulle.

Jatkotutkimukselle on paljon tilaa Pythonin tyypityksen saralla. Tyyppitarkistimista voisi tehdä uusia vertailuja, vihjeistä maksimihyödyn saamista voisi tarkastella, ja tilastollinen analyysi tyyppivihjeiden käytöstä vuoden 2022 jälkeen antaisi tilannekuvaa uusista trendeistä. Tyypitystyökaluja voisi myös verrata muihin laadunvarmistuksen työkaluihin, kuten yleisiin tarkistusohjelmiin (linter) tai automaattitestaukseen, ja selvittää mikä on optimi aika- ja vaiva-allokaatio näiden välillä. Ohjeille tyyppivihjeiden käyttöönotosta on myös tilaa.

Contents

1 Introduction

Python is a general purpose programming language that is in widespread use. Since 2015 there has been support for optional type annotations (type hinting), which enables adding additional type information to the source code to aid development.

Python's growing market share, along with the growing popularity of typing solutions for Python, seemingly contain a contradiction: Why adopt a dynamic language, if you are going to statically type the code anyway?

In *statically typed* languages, the source code is assigned types at compile time, with type annotations and type inference. A *dynamically typed* language assigns types at runtime. This affects the way programmers interact with type errors and shifts the focus point of guarantees from compilation time to runtime.

The choice between a statically or dynamically typed language can have a modest but statistically significant effect on code quality [15, 20]. However in a reproduction study [1] the result from [20] could not be reproduced. The authors concluded that analysing quantitative differences from large-code samples from different programming languages can easily contain statistical errors that make the results misleading or overconfident.

Python with the optional types in use (*typed Python*) can be described as a gradually typed language. *Gradual typing* is a mix of dynamic and static typing. Applying gradual typing to existing dynamic languages, such as using TypeScript over JavaScript, can provide statistically significant improvements in detecting software defects (bugs) [6]. Similar results have been found for typed Python [10, 19].

This thesis aims to provide an overview for Pythons type system and type annotations, and their usage and benefits. Chapter 2 provides technical background for type systems, Python, its type system, and type-checking solutions for Python. Chapter 3 analyses the research and practical applications of type annotations, and their implementations and impact. Then discussion 4 presents motivations and reasons for adopting Python and type annotations. Chapter 5 proceeds to synthesize the research insights into answers.

2 Background

2.1 Correctness and defects

In study of algorithms and theory of computation correctness of a program refers to it behaving according to the specification. A program can be proven correct by proving that it halts and that it outputs the result demanded by the specification [13].

When analysing real-world software engineering formal proofs are rarely available, and weaker guarantees are worth inspecting. In this thesis *correctness* and *more correct* are used to describe relative correctness [4]. Programs that are more correct follow their specification better, and thus have fewer *defects*.

2.2 Type systems

Type systems are tools that enable abstract organization and categorization of data and related tools [5]. They allow the programming language to give guarantees about data structures, they enable varying behaviour based on the specified type, and type data can work as inferred, controllable comments. The information describing the type properties of a symbol is called *type information*. Type information restricts the values a variable can hold or the types of data a function can receive or output. In the context of types, a *correct type* describes the data accurately.

Effectively all high-level programming languages make use of type systems. The systems vary in features, granularity, constraints and inference logic. Languages can be classified as static, dynamic or gradual by how their type system is applied, and to strong and weak languages based on the strength of the type system guarantees. The guarantees that type systems give on program behaviour can improve the correctness of programs. 2.1 provides examples of programming languages with different type system features.

Type system features	Languages
Static	C, Java
Dynamic	Python, JavaScript
Gradual	TypeScript, Typed Python
Strong	Python, Java
Weak	C, JavaScript, TypeScript

Table 2.1: Examples of programming languages with varying type system features.

2.2.1 Type checking

Type checking refers to the process that programming languages use to ensure the that types are correct [5], and follow the rules of the type system. Type checking strategy varies based on the language's type system's features: static languages check at compile time, dynamic languages check at runtime, and gradual systems check at both, but with different granularity for each. Type checking raises *type errors* when a type is incorrect. Programming languages vary in implicit conversions between types when an operation is applied to differently typed arguments.

A reason to avoid implicit conversion of arguments is to improve correctness, avoiding unchecked runtime type errors which can lead to undetected erroneous behaviour. A language that aims for no unchecked runtime type errors has a *strong type system* [2]. This means that runtime type errors must stop execution or cause checkable exceptions. Languages that allow unchecked runtime type errors and implicit runtime conversion between most types have *weak type systems*. In weak type systems, type errors will rarely stop the execution of a program. While seeming useful this can lead to undetected defective behaviour.

2.2.2 Static, dynamic and gradual typing

Static typing requires a user to specify types at compile time, or for the compiler to have the ability to correctly infer types for all symbols. Tools for static languages can provide better suggestions and error detection since more data is available for analysis. Compilers can also perform more effective optimizations for code based on type information. It is commonly believed that statically typed languages have better performance when optimized, and

programming with them results in more correct programs. The first is often true but can vary case-by-case [15]. The second is hard to analyse empirically [1] because many other qualities affect the correctness of written programs.

In a dynamically typed language, the final assignment of types happens at runtime. Runtime assignment of types can be idiomatic if unknown data is received into the program and parsed in runtime. Dynamic languages feature lightweight syntax that can make it faster to start development [3]. Dynamic languages are often taught as first programming languages, to avoid needing to teach students type system complexities that distract from learning core programming concepts.

Gradual typing can be thought of as a superset of static and dynamic typing [21]. In a gradually typed language one can write a statically typed program, a dynamically typed program, or various in-between programs. The unique advantage of gradual typing is the ability to type important sections of a larger program and evolve these typed sections and their proportion gradually [21]. Gradual typing is often implemented as a superset of an existing programming language.

2.3 Python

Python is a high-level interpreted programming language whose syntax uses significant whitespace for conciseness. It has a strong and dynamic type system. Significant usage domains include scientific computing, machine learning, web backend development and scripting. The concise syntax and capability to make programs executable quickly have been significant advantages in surpassing the market share of stricter, compiled languages such as Java, C# and C++.

2.3.1 Types in Python

Type annotated Python has two type systems which have different scopes:

1. The runtime type system is semantically and behaviourally relevant. The type of a Python symbol at runtime contains data on its values shape and possible behaviour for the object. Available and allowed operations are interpreted based on the symbols runtime type.

2. The optional annotation system was added in 2014 after Python developers drafted

Python Enhancement Proposal 484 - Type hints [7]. Ability to do improved static analysis, the possibility for improved refactoring, runtime type checks, and code generation were the stated motivation. Static analysis was stated to be the most important one. A prototypal type checker and the ability to add type metadata through a generic mechanism already existed, but this proposal standardized the way to annotate Python code with type data, enabling improved tooling development across the field.

Enforcement and performing any static analysis on the annotations described in 2) is completely optional and up to the developer [17].

In this thesis, we focus on discussing the annotation-based higher-level gradual type system. Python's implementation of type annotations does not affect program behaviour in the way runtime types do. Type annotated Python code can be run without ever executing a type check, and without making use of the annotated type data. This follows the *gradual guarantee* [21], where a type annotated and non-annotated program should only differ in their type annotations. Untyped Python code (2.1) does not use type annotations. Type annotations are denoted with `:` for variables and function arguments and `->` for return values (2.2).

```
1 def f(i, j):
2
3     sum = i + j
4     return sum % 2 == 0
```

Figure 2.1: Untyped Python function

```
1 def f(i: int, j: int) ->
      int:
2     sum: int = i + j
3     return sum % 2 == 0
```

Figure 2.2: Typed Python function

2.3.2 Python type checking

As of 2024, there are three widely adopted tools for Python type checking: Mypy, Pyright, PyCharm. There are also general linters and other code quality tools that type check, but these three are the most prevalent.

Mypy by Python foundation is a common tool to run in Continuous Integration environments to do type checking across tests. It has a command line interface and can be utilized through Language Servers in code editors.

Microsoft's Visual Studio Code is the most popular editor for Python programming [18]. It utilizes Pyright for type checking by default, in-line type errors.

```
1 def f() -> bool:
2     num = 0
3     return num
```

Figure 2.3: Inferred type error

PyCharm is a commercial Python IDE developed by JetBrains s.r.o. It is the second most popular Python editor and the most popular full IDE. PyCharm has its own integrated type checker. PyCharm, being an IDE, is rarely utilized outside of the developer's machine.

It is common for developers to develop code with VS Code or PyCharm and then run `mypy` in a continuous integration environment, providing a kind double checking with slightly different prioritisations affecting what situations each checker considers errors.

2.3.3 Type inference

Mypy, Pyright and PyCharm all support *type inference* [8, 11, 14]. This means that they can *infer* the type of a variable from the values assigned to it. The tools can also infer function return values from return values. Type inference enables some benefits from type checking even when code is minimally typed, and is commonly used for editor features such as autocompletion and detecting type errors before runtime.

Mypy 1.13.0 returns the following when checking the source code in 2.3:

```
type_inference_example.py:3: error:
Incompatible return value type (got "int", expected "bool") [return-value]
```

The error indicates that on line 3 the type checking rule `[return-value]` was violated. It also indicates that the value that is being returned is type `int` indicating that `num`'s type was correctly inferred.

3 Research on type annotations

3.1 Researching type annotations

The most prevalent research method to investigate if existing type annotations in Python code are correct is to sample a dataset of source code and then run type checking against it [3, 12, 19, 22]. This creates data on type coverage and type errors. To get data on defect detection fixed defects are sampled. By adding correct type annotations to the defective source code before the fix, researchers can detect if a type checker could have detected the defect.

3.2 Usage of type annotations

Annotations often stay unchanged for an extended period after being added [3]. Projects can be categorized into three patterns: regular annotation, type sprints, and occasional usage. In a dataset of 9655 Python projects the used pattern correlated with the number of contributors, with regular annotation averaging the highest contributor count of 62, type sprints 45, and occasional use projects averaging 25 contributors. A motivation of type annotations helping coordination between higher numbers of active contributors was hypothesized to explain this phenomenon.

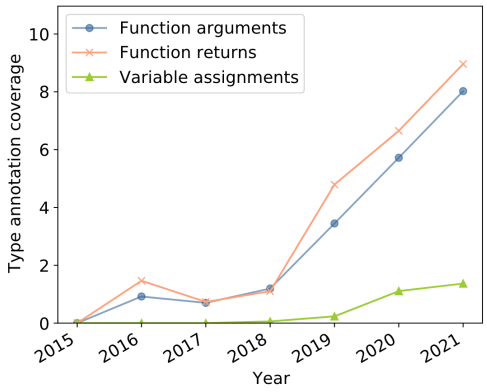

Figure 3.1: Python type annotation evolution from 2015 to 2021 [3]

From the introduction of Python type annotations in 2015 they reached around 8% function argument and return coverage by 2021 in an analysis of 9655 Python projects [3].

Using a single large Python project picked for representativeness (TensorFlow) [12], the ratio of functions that have type annotations rose from 5% in 2019 to 39% in 2022. This increase seems consistent with the general rate in 3.1. The trend of adoption for type annotations has been increasing, but it should be noted that the proportion of typed code was not very high by 2022.

3.2.1 Incorrect type annotations

A 2020 study of Python type annotations usage patterns [19] found that even though type annotations and type checking are being adopted more widely, only 15% of the 2678 codebases analysed successfully type checked with Mypy. A study of 9655 most popular Python projects on GitHub in 2022 found that 78% of commits that had type annotations contained type errors [3].

Some errors that Mypy and Pytype find are false positives [19]. False positives can be caused by differences in type checking setup to the one originally used by the project. Different type checkers, type checker versions, and configurations affect what type patterns are raised as errors, and which errors are detected.

3.2.2 Empirical benefits

Making use of types and type checking can allow development time detection of various program defects. In a retroactive analysis it was found that up to 11% of a sample of patched defects could have been found at development time with types [10]. Another analysis [22] found that in a sample of 40 type-related defects 35% could be detected without type annotations and 73% with annotations.

Both experienced and inexperienced programmers make significant amounts of type-related mistakes when working with Python [10]. These consist of: using variables before initialising them, null safety errors, and re-using variables with values of different types. This implies that both experienced and inexperienced programmers can derive benefits from types, since type checking helps fix these mistakes.

Pydantic is a widely adopted data validation library for Python that enables runtime data validation using type annotations [16]. There is no academic research available on its

effectiveness or popularity. Common use cases include web services where incoming data has to be validated anyway, type annotations enabling a simple way to write schemas for the data.

4 Why use typed Python

4.1 Why use Python

Python's popularity can be attributed to several factors:

- Python's design goals have led to a programming language that is simple to read, quick to write, and easy to learn. Effectively no popular statically typed languages have a similar design focus on developer experience.
- The ease of learning Python leads to it being a popular first programming language to learn, which leads it to have a large developer base.
- Useful packages are widely available. There is excellent support for various use cases, including scientific computing, machine learning, web development, desktop software development, and automation.

4.2 Why type annotate Python

Research on Python Type annotations focuses on how they are used, and their benefits and adoption patterns. It is rare to analyse why they are adopted.

In practice developers using type annotations are not aiming for completely typed Python programs. As can be seen from the amount of type errors present in most Python projects in the wild [3, 19], developers are utilizing the fact that a gradual type system enables gradually typing projects.

Figure 4.1 describes Python and Typed Python relative correctness and conciseness properties compared to other programming languages. Here correctness describes how much the programming language supports developers in writing correct programs. Conciseness refers to the amount of required source code to achieve a certain function.

The figure is based on a study of various programming languages' performance on Rosetta Code tasks [15]. They do not generalize to generic programs, but they are indicative and the relations are consistent with other research [20]. The Typed Python indicator is hypothetical.

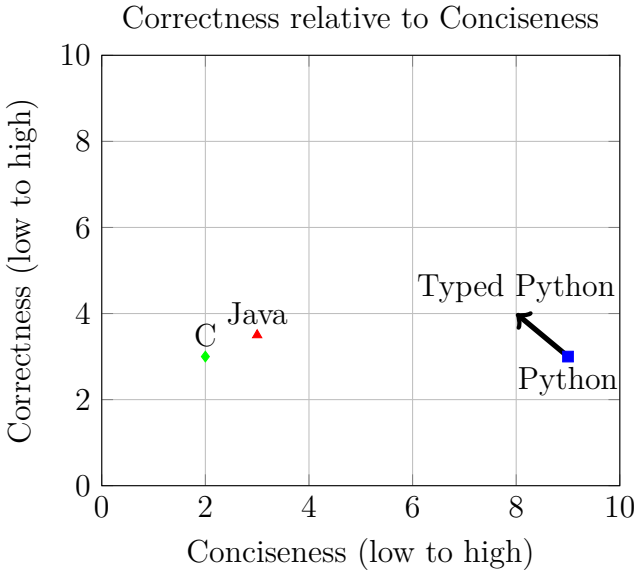

Figure 4.1: A comparison of programming languages on conciseness and correctness.

5 Conclusions

Python is widely adopted for various reasons, including ease of use, ease of learning, and the wide availability of packages. In comparison to other programming languages it lands at a useful compromise on readability, expressivity and conciseness.

Utilising type annotations types makes Python into a gradually typed programming language, with increased readability, self-documentation and correctness, with costs in additional syntactic noise and additional development effort. This has led to type annotations being adopted more widely [9, 10]. Type checking has been made easier by the popularity of editors that do it by default. This assists developers in finding defects and utilising type information better.

Typed Python allows developers to utilize Python's core strengths while improving on shortcomings that have historically widely applied to dynamic scripting languages.

Python projects should adopt type annotations when they want to spend development efforts on improving the maintainability, correctness, and static analysis of their code base. Intentions on coverage, specifically which internal and public APIs should end up covered, should be set [9]. Some time should be spent on estimating where correctness and analysis improvements provide the most benefit. Often this would be parts of the software where type annotating is relatively easy, and which often cause bugs. When type annotating proves difficult it can be an indicator that the program structure is complicated, and could benefit from refactoring.

Promising avenues for future research include: comparing type checkers, especially PyCharm and Pyright, optimal adoption of type annotations (including instructions for developers), and comparing type annotations to other quality assurance tools such as automated testing to determine where correctness investments provide the most benefit.

Bibliography

[1] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek. "On the Impact of Programming Languages on Code Quality: A Reproduction Study". en. In: *ACM Transactions on Programming Languages and Systems*, 41(4) (2019). DOI: 10.1145/3340571.

[2] L. Cardelli. "Typeful Programming". In: *Formal Description of Programming Concepts, IFIP State of the Art Reports Series*. Springer-Verlag, 1989. ISBN: 978-3-540-53961-2. URL: http://lucacardelli.name/Papers/TypefulProg.A4.pdf.

[3] L. Di Grazia and M. Pradel. "The evolution of type annotations in python: an empirical study". en. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore: ACM, 2022, pp. 209–220. DOI: 10.1145/3540250.3549114.

[4] N. Diallo, W. Ghardallou, and A. Mili. "Correctness and Relative Correctness". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 591–594. DOI: 10.1109/ICSE.2015.200.

[5] M. Gabbrielli and S. Martini. "Programming Languages: Principles and Paradigms". en. In: Undergraduate Topics in Computer Science. London: Springer London, 2010, pp. 197–262. DOI: 10.1007/978-1-84882-914-5.

[6] Z. Gao, C. Bird, and E. T. Barr. "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 758–769. DOI: 10.1109/ICSE.2017.75.

[7] Guido van Rossum, Jukka Lehtosalo, Łukasz Langa. *PEP 484 – Type Hints*. en. Sept. 2014. URL: https://peps.python.org/pep-0484/ (visited on 10/22/2024).

[8] JetBrains s.r.o. *Type hinting in PyCharm | PyCharm*. en-US. URL: https://www.jetbrains.com/help/pycharm/type-hinting-in-product.html (visited on 12/06/2024).

[9] W. Jin, D. Zhong, Z. Ding, M. Fan, and T. Liu. "Where to Start: Studying Type Annotation Practices in Python". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 529–541. DOI: 10.1109/ASE51524.2021.9678947.

[10] F. Khan, B. Chen, D. Varro, and S. McIntosh. "An Empirical Study of Type-Related Defects in Python Projects". In: *IEEE Transactions on Software Engineering*, 48(8) (2022), pp. 3145–3158. DOI: 10.1109/TSE.2021.3082068.

[11] J. Lehtosalo and mypy contributors. *Type inference and type annotations - mypy 1.13.0 documentation*. URL: https://mypy.readthedocs.io/en/stable/type_inference_and_annotations.html (visited on 12/13/2024).

[12] X. Lin, B. Hua, Y. Wang, and Z. Pan. "Towards a Large-Scale Empirical Study of Python Static Type Annotations". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 414–425. DOI: 10.1109/SANER56733.2023.00046.

[13] Z. Manna and A. Pnueli. "Axiomatic approach to total correctness of programs". en. In: *Acta Informatica*, 3(3) (1974), pp. 243–263. DOI: 10.1007/BF00288637.

[14] Microsoft. *Type Inference*. URL: https://microsoft.github.io/pyright/#/type-inference (visited on 12/13/2024).

[15] S. Nanz and C. A. Furia. "A Comparative Study of Programming Languages in Rosetta Code". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 778–788. DOI: 10.1109/ICSE.2015.90.

[16] pydantic.dev. *Welcome to Pydantic - Pydantic*. en. URL: https://docs.pydantic.dev/latest/ (visited on 10/30/2024).

[17] Python Software Foundation. *typing — Support for type hints*. en. URL: https://docs.python.org/3/library/typing.html (visited on 10/30/2024).

[18] Python Software Foundation, JetBrains s.r.o. *Python Developers Survey 2023 Results*. en. URL: https://lp.jetbrains.com/python-developers-survey-2023/ (visited on 12/06/2024).

[19] I. Rak-amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby. "Python 3 types in the wild: a tale of two type systems". en. In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. Virtual USA: ACM, 2020, pp. 57–70. ISBN: 978-1-4503-8175-8. DOI: 10.1145/3426422.3426981.

[20] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. "A Large Scale Study of Programming Languages and Code Quality in Github". en. In: (2014). DOI: 10.1145/2635868.2635922.

[21] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. “Refined Criteria for Gradual Typing”. en. In: *LIPIcs, Volume 32, SNAPL 2015*, 32 (2015), pp. 274–293. DOI: 10.4230/LIPICS.SNAPL.2015.274.

[22] W. Xu, L. Chen, C. Su, Y. Guo, Y. Li, Y. Zhou, and B. Xu. “How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python”. In: *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 2023, pp. 242–253. DOI: 10.1109/ICPC58990.2023.00039.

Appendix A Use of AI

ChatGPT 4-o was used to generate LaTeX source code for figures 2.1, 2.2, 4.1 and table 2.1.

ChatGPT 4-o, Overleaf's Writeful integration, and Grammarly were used for proofreading the final draft of the thesis.

ChatGPT 4-o and Claude 3.5 Haiku was used to spell check the extended abstract in Finnish.