

The Evolution of Type Annotations in Python: An Empirical Study

Luca Di Grazia
University of Stuttgart
Germany

luca.di-grazia@iste.uni-stuttgart.de

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp.209-220

発表者：坂本

動的型付け言語

| 動的型付け言語

- 変数, フィールド, 戻り値に型を指定しない
- PythonやJavaScript
- 自由度が高いため人気の言語

| 問題点

- 保守作業の手間
- IDEのサポートが限定される

| 型アノテーション

- Pythonではバージョン3.5でサポート
- 型をチェックするツールの登場

Pythonでのアノテーションの例

(a) Commit 1.

```
def f(x: float, y):  
    sum = x + y  
    if (sum % 2) == 0:  
        return True
```

(b) Commit 2.

```
def f(x: int, y) -> bool:  
    sum: int = x + y  
    if (sum % 2) == 0:  
        return True
```

(c) Commit 3.

```
def f(x: int, y) -> Optional[bool]:  
    sum: int = x + y  
    if (sum % 2) == 0:  
        return True
```

| 2つの変数の合計が偶数かを確認する関数

- 3回のコミットにおける進化の例

| Commit1

- 変数xはfloatで型付けされているが、変数yはされていない

| Commit2

- xの型がintへ変更、戻り値はboolで型付け

| Commit3

- 戻り値の型をOptional[bool]へ変更

目的 & RQ

| 目的

- Pythonにアノテーションを追加することがメリットになるのかを再確認
- タイプチェッカーが有用であるかの理解

| RQ

- RQ1：エコシステムレベルで，アノテーションの利用はどのように進化しているか
- RQ2：プロジェクトレベルで，アノテーションの利用はどのように進化しているか
- RQ3：個々のアノテーションはどのように進化していくか
- RQ4：型エラーとアノテーションはどのような関係か

型アノテーションの抽出

| 定義1

- アノテーション : $t_{ann} = (t, n, k, l)$
 - t : 型
 - n : アノテーション付き要素の名前
 - k : アノテーション付き要素の種類
 - l : アノテーションの位置

(a) Commit 1.

```
def f(x: float, y):  
    sum = x + y  
    if (sum % 2) == 0:  
        return True
```

t : "float"
 n : "x"
 k : "argument type"
 l : "line 1 of the corresponding file"

| 定義2

- アノテーションカバレッジ
 - コードベースBが与えられたときのプログラム要素のカバレッジ

$$cov_{ann} = \frac{|\{(t, n, k, l) \in B \mid k = k_{target}\}|}{|\{\text{All program elements of kind } k_{target}\}|}$$

(b) Commit 2.

```
def f(x: int, y) -> bool:  
    sum: int = x + y  
    if (sum % 2) == 0:  
        return True
```

引数のカバレッジ : 50%
戻り値のカバレッジ : 100%

型アノテーションの変化の抽出

| 定義3

- アノテーション変更 : $t = (t_{old}, t_{new}, c, d)$
 - t_{old} : コード変更前のアノテーション
 - t_{new} : コード変更後のアノテーション
 - c : コード変更の種類
 - d : コード変更の日付

(b) Commit 2.

```
def f(x: int, y) -> bool:  
    sum: int = x + y  
    if (sum % 2) == 0:  
        return True
```

変数 x の更新
戻り値へのアノテーション
変数 sum へのアノテーション

| 定義4

- アノテーション履歴 : $[t_1, \dots, t_m \text{ change change}]$

データ収集アルゴリズム

| 定義4

- 一連のコミットを繰り返してアノテーションの変更点を抽出

アノテーションを抽出

アノテーション変更のペアがコミット内の
ハンクの行範囲に収まっているかを判定

変更前にのみ存在するアノテーション
変更後にのみ存在するアノテーション
を特定

Algorithm 1 Extract type annotation changes from commits.

Input: Sequence C of commits

Output: Set T of type annotation changes (old and new type pairs)

1: $T \leftarrow \emptyset$

2: **for** commit c **in** C **do**

3: $T_{old} \leftarrow$ type annotations in code before commit c

4: $T_{new} \leftarrow$ type annotations in code after commit c

5: $d \leftarrow$ date of c

6: $T' \leftarrow \emptyset$

7: **for** $(t_{old}, t_{new}) \in T_{old} \times T_{new}$ **do**

8: **if** $\exists$ hunk h in c where t_{old} in $oldLineRange(h)$

9: **and** t_{new} in $newLineRange(h)$ **then**

10: **if** t_{old} and t_{new} have same kind k

11: **and** same name n **then**

12: Add $(t_{old}, t_{new}, \text{"update"}, d)$ to T'

13: $T' \leftarrow ensureSingleMatch(T')$

14: **for** t_{new} not yet added to T' **do**

15: Add $(_, t_{new}, \text{"insert"}, d)$ to T'

16: **for** t_{old} not yet added to T' **do**

17: Add $(t_{old}, _, \text{"remove"}, d)$ to T'

18: $T \leftarrow T \cup T'$

データセット

| タイプチェッカー(pyre)の使用

- タイプエラーの種類
 - 依存関係の欠落によるエラー
 - 実際の型のエラー

| データセット

- 全てのPythonプロジェクトのリストを取得
- 作成日ごとに10グループへ分類
- → グループごとに上位1000件, 合計1万件
- 9,655個のリポジトリ, 1,123,393 件のコミット, 1,414,936件のアノテーションの変更

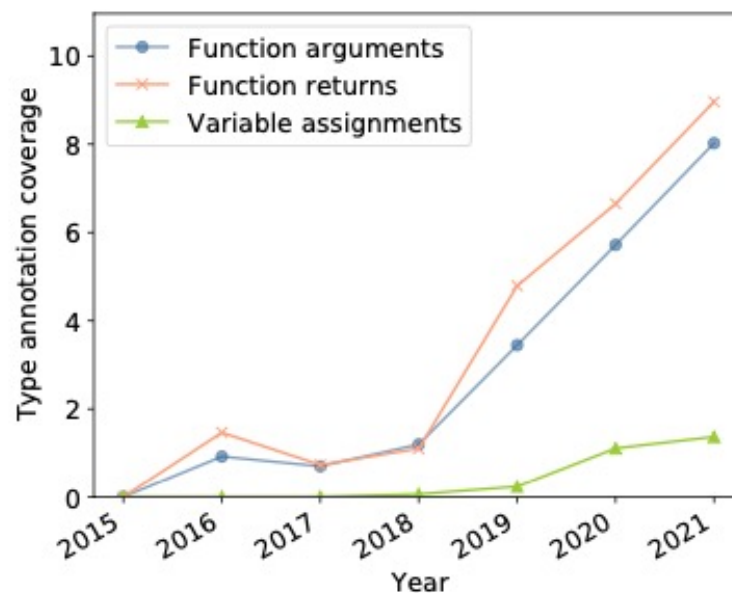
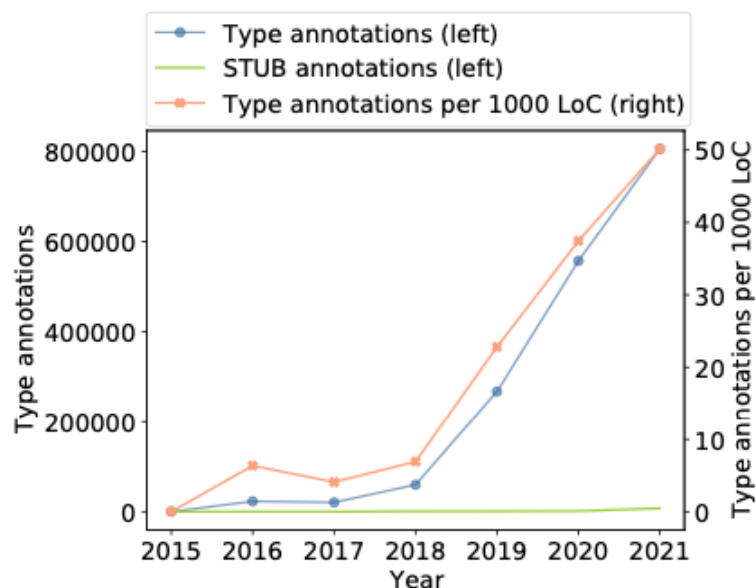
RQ1(エコシステムレベルでのアノテーションの進化)

| タイプアノテーションの総数の進化

- アノテーションの使用数は2017年以降、着実に増加

| アノテーションカバレッジの進化

- アノテーションを使用しているプロジェクトは7%(668件)
- 変数のアノテーションにはあまり目が向けられていない



RQ2(プロジェクトレベルでのアノテーションの進化)

| 3つのカテゴリへ分類

- 定期的なアノテーションの使用
- タイプスプリント
- 時々の使用

Algorithm 2 Determine project-level evolution pattern.

Input: Project P

Output: Evolution pattern of P

Divide P into 10 time steps of equal number of commits

```
1:  $P.annotations \leftarrow$  Number of annotations for each time step
2:  $P.slope \leftarrow$  Annotation evolution slope for each time step
3: if  $\max(P.annotations) < 15$  or  $P.slope.count(0.0) > 8$  then
4:   return "Occasional use"
5: else if  $P.slope.count(0.0) \geq 4$  then
6:   return "Type sprints"
7: else if  $Average(P.slope) \geq 0$  then
8:   return "Regular annotators"
9: else
10:  return "Other"
```

アルゴリズム2でラベル付け

プロジェクトがどのパターンに
当てはまるか判定

- 定期的なアノテーションの使用：44.4%
- タイプスプリント：28.1%
- 時々の使用：25.4%

RQ4(型エラーとアノテーション)

| 分析パート

- どれだけの型エラーが存在するかを計算
- 型エラーの数とアノテーションの間に相関があるか

| どれだけの型エラーが存在するか

- 各プロジェクトに静的解析を実行. 依存しない型エラーをカウント
 - 分析対象の78.3%に少なくとも1つの型エラー
平均で1000行当たり6個のエラー

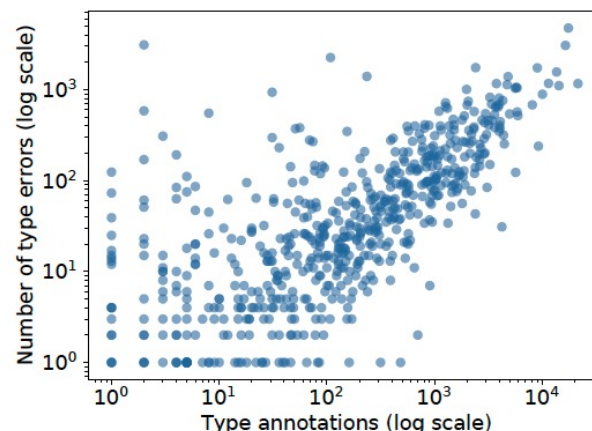


型チェックが開発ルーチンとして一般的ではない可能性

RQ4(型エラーとアノテーション)

| 型エラーの数とアノテーションの相関

- アノテーションの使用で、静的解析で型エラーを検出することができる
- pyre (タイプチェッカー)を使用



- 各プロジェクトの型エラーの数とアノテーションの数の相関
 - ピアソン係数：0.704



エラーを検出可能なプロジェクトはアノテーションが多い
型チェックツールの使用が必要

まとめ

| 型アノテーションは普及しつつある

- しかしほとんどのPythonコードでアノテーションが使用されていない

| 型アノテーションを使用することで型エラーを静的に発見できる

所感

| アノテーションを使用しているプロジェクトが極端に少ないのは意外だった