

Towards a Large-Scale Empirical Study of Python Static Type Annotations

Xinrong Lin, Baojian Hua, Yang Wang, Zhizhong Pan

Proceedings of the 30th International Conference on Software Analysis, Evolution and
Reengineering (SANER '23), pp. 414-425

Pythonと静的型付け

Python

- データサイエンスの主要言語の一つ
- 型宣言を必要としない動的型付け言語
 - 柔軟な開発が可能に

動的型付け言語における問題

型関連の問題を引き起こし、ネガティブな影響を与える可能性
開発の生産性、コードの利用や品質の低下 など[4][5][6][7]

→ 静的検査が可能な型アノテーションの導入

- [4] L. A. Meyerovich and A. S. Rabkin, “Empirical analysis of programming language adoption,” in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 1–18.
- [5] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A large scale study of programming languages and code quality in github,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Hong Kong China: ACM, Nov. 2014, pp. 155–165.
- [6] F. Khan, B. Chen, D. Varro, and S. McIntosh, “An empirical study of type-related defects in python projects,” IEEE Transactions on Software Engineering, pp. 1–1, 2021.
- [7] Z. Chen, Y. Li, B. Chen, W. Ma, L. Chen, and B. Xu, “An empirical study on dynamic typing related practices in python systems,” in Proceedings of the 28th International Conference on Program Comprehension. Seoul Republic of Korea: ACM, Jul. 2020, pp. 83–93.

静的型アノテーション

静的型アノテーション

	引数の型	戻り値の型
	<code>x: int</code>	<code>-> str:</code>
ローカル変数の型	<code>z: int</code>	<code>= x + y</code>
	<code>return z</code>	
	<code># Error: Incompatible return value type</code>	

型アノテーションの例

- Pythonでは、PEP 484^[8] を用いた記法が導入されている
 - typingモジュールを用いて、より高度なアノテーションを実現
 - プログラムの実行に影響を与えない
- **型関連の欠陥の検出が可能に**

静的型アノテーションや検査ツールの利用に関して不明

[8] <https://peps.python.org/pep-0484/>

研究目的

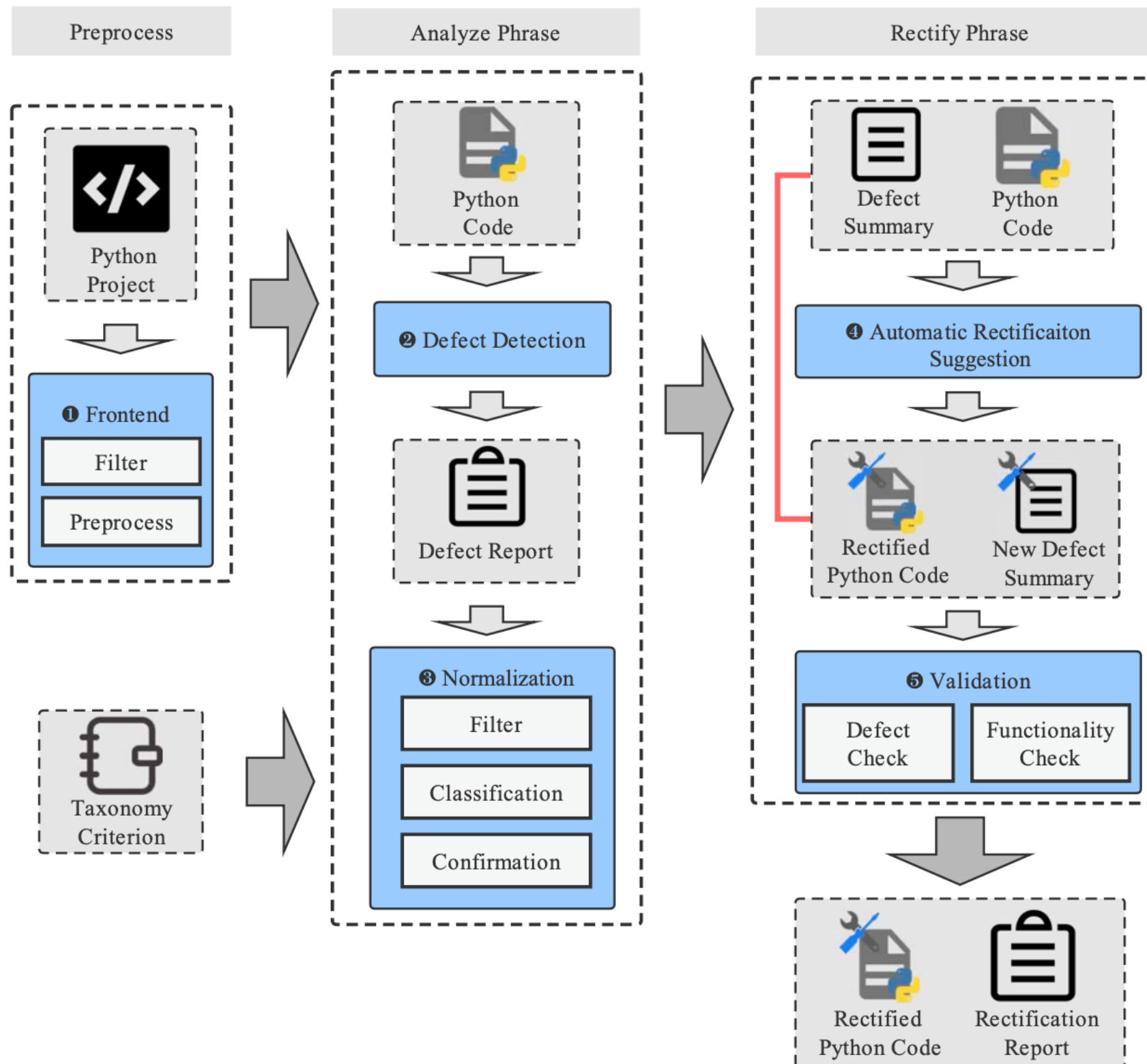
課題

- 公式や権威ある検査基準やツールの欠落
- 静的型アノテーションに関する統一された欠陥分類の欠落
- 静的型アノテーションに関する欠陥の自動修正

目的

- 静的型アノテーションや欠陥などに関する理解と自動修正に関する検討

PYSCAN : 全体像

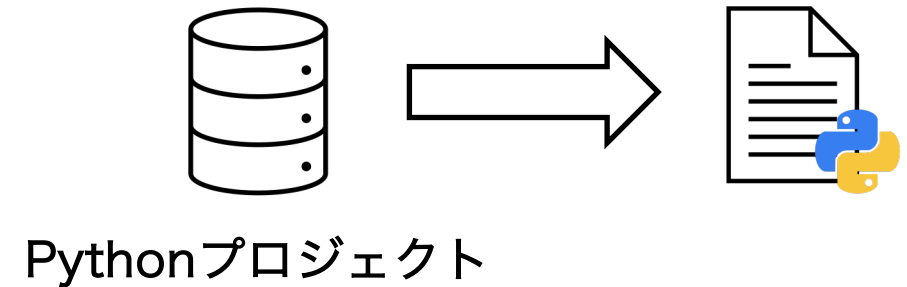


PYSCAN：各モジュールの役割

① Frontend

静的型アノテーションを利用しているpython
ファイルを抽出しリスト化

- ドキュメントなどは除外

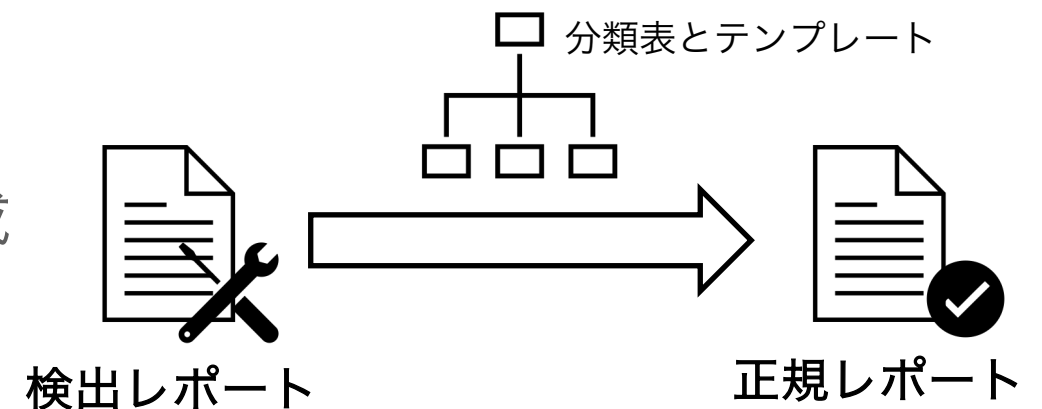


② Defect Detection & ③ Normalization

複数の検査ツールを用いて欠陥を検出し
分類表とテンプレート※に沿ってレポートを生成

- ツール：Mypy, Pytype, Pyre, Pyright

※ 同じ欠陥でもツールごとにレポートの様式が異なるため

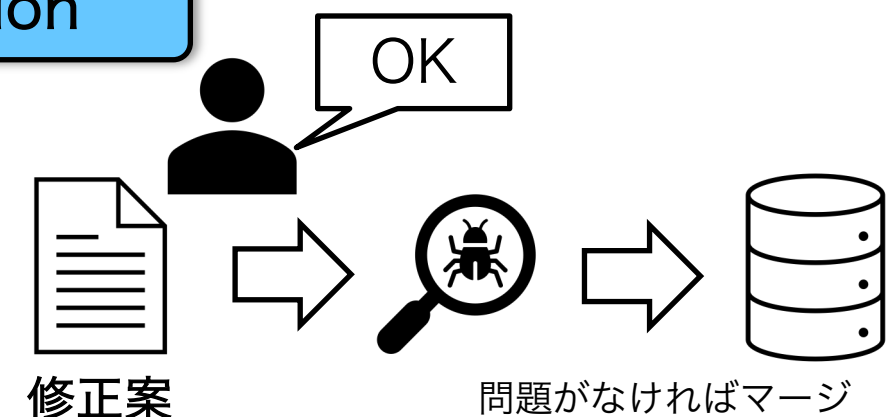


④ Automatic Rectification Suggestion & ⑤ Validation

欠陥を含むファイルと正規レポートから修正案を生成
開発者が承認すれば検証を行い，問題がなければ統合

- 欠陥が修正されているか？機能的変更がないか？

※ おそらく未実装のため，具体的な方法が不明



Research Question

RQ1

- 型アノテーションに関連する欠陥の分類は？

RQ2

- PYSCANはどのくらい欠陥を検出できるのか？

RQ3

- 型アノテーションに関連する欠陥の進化はどうか？

RQ4

- 修正提案はどのくらい利用できそうか？

RQ5

- 調査結果は実務者にとって役立つものか？

調査対象

Project	TSV Score	Files	LoC (K)	Stars	Watchers	Contributors
Cpython[33]	49.93	2,045	910	39.2k	1,395	1,749
Ansible[34]	47.41	1,560	253	49.2k	2,009	6,427
Core[35]	38.20	8,780	1,511	44.6k	1,350	2,791
FastAPI[36]	36.54	800	58	34.1k	527	240
Scikit-learn[37]	35.26	915	350	46.6k	2,256	2,300
Django[38]	34.62	2,751	447	58.7k	2,332	2,548
TensorFlow[39]	32.64	2,894	1,102	70.7k	2,936	820
Pandas[40]	31.28	1,444	576	30.5k	1,119	2,731
Flask[41]	28.96	70	18	56.1k	2,239	726
Transformers[42]	28.00	1,813	798	48.9k	756	943
Faceswap[43]	27.89	231	71	37.8k	1,505	87
Scrapy[44]	26.61	338	55	41.2k	1,829	483
Requests[40]	23.98	35	11	45.7k	1,393	694

より規模が大きいプロジェクトを選定

1. 自動化 手動の型検査は時間がかかりエラーを含みやすい
2. スケーラビリティ さまざまなドメインのプロジェクトに適用可能

RQ1：型アノテーションに関連する欠陥の分類は？

欠陥分類表

Taxonomy	Inconsistent Type Annotation	Insufficient Safety Check	Incorrect Redefinition / Overload	Wrong Type Annotation
Patterns	Inconsistent Return Type	Insufficient Checks for None	Multiple Definitions	Undefined or Invalid Type
	Inconsistent Parameter Type	Insufficient Checks for Existence of Members	No Applicable Overload or Redefinition	Illegal Annotation Target
	Inconsistent Variable Type	Insufficient Checks for Operator Support	-	-
	Inconsistent Attribute Type	-	-	-

他のパターンを検索しようとしたがクローズドアクセスとなっていた

Inconsistent Type Annotation

アノテーション対象と型の不一致

- 例：戻り値の不一致のパターン

```
def _get_account(account_identifier: str) ->  
    IcloudAccount:  
    if account_identifier is None:  
        return None
```

型アノテーション `IcloudAccount`:
×
実際の戻り値 `return None`

- 型アノテーションでは、IcloudAccountと独自の型を指定
- しかし、戻り値はNoneを取りうることもある

→ `_get_account()`がNoneを返すことを想定していないとエラー

- IcloudAccountのメソッドなど参照するとプログラムが止まる

Insufficient Safety Check

予期していない型を防止するためのチェックが不足

- 例：Noneチェック

```
netloc_re = re.compile(...)
match = netloc_re.match(parts[1]) マッチング
auth = match.group(1) マッチした文字列の参照
```

- 検索文字列がヒットしなかった場合を考慮していない

- 例：データ操作

```
def get_name_with_age(name: str, age: int):
    name_with_age=name+" is this old: "+age
    return name_with_age
```

str型とint型が混在

- フォーマット関数などを利用するべき

RQ2 : PYSCANはどのくらい欠陥を検出できるのか？

ツールの検出評価

各ツールが検出した4つのパターンの欠陥が正しいか
人手で分析

Tool	ITA		ISC		IRO		WTA		Total				
	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Precision	Recall	F1
Mypy	6	3	7	3	2	2	0	0	15	8	53.33%	7.27%	12.80%
Pyre	0	0	0	0	2	1	160	15	162	16	9.88%	14.55%	11.76%
Pyright	76	37	97	45	0	0	0	0	173	82	47.40%	74.55%	57.95%
Pytype	8	3	33	0	0	0	9	1	50	4	8.00%	3.64%	5.00%
Total	90	43	137	48	4	3	169	16	400	110	/	/	/

- Mypyの場合：

- $Precision = \frac{8}{15} \times 100 = 53.33 (\%)$

- $Recall = \frac{8}{110} \times 100 = 7.27 (\%)$

Pyrightが最も良く検出できるツールであることがわかった

まとめ

自動修正に向けて静的型アノテーションと関連する欠陥について大規模調査を実施

- 13のPythonプロジェクト対象に分析
- PYSCANの提案

型アノテーションにまつわる4つの欠陥パターン

- Inconsistent Type Annotation
- Insufficient Safety Check
- Incorrect Redefinition/Overload
- Wrong Type Annotation

所感

提案手法のPYSCANの自動修正の部分がどのように実現させるのかが記述されていなくてRQ4が解釈できなかった

- 修正対象コードと検出レポートを入力にすることしか書いていない

保守のしやすさの観点から型アノテーションの利用は個人的にも推奨したいです

- 言わずもがなコメントも