Python-by-Contract Dataset

Jiyang Zhang
jiyang.zhang@utexas.edu
The University of Texas at Austin
Austin, USA

Marko Ristin
rist@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

Phillip Schanely
pschanely@gmail.com
Independent researcher
New York, USA

Hans Wernher van de Venn
vhns@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

Milos Gligoric
gligoric@utexas.edu
The University of Texas at Austin
Austin, USA

ABSTRACT

Design-by-contract as a programming technique is becoming popular in Python community as various tools have been developed for automatically testing the code based on the contracts. However, there is no sufficiently large and representative Python code base with contracts to evaluate these different testing tools. We present Python-by-contract dataset containing 514 Python functions annotated with contracts using icontract library. We show that our Python-by-contract dataset can be easily used by existing testing tools that take advantage of contracts. The demo video can be found at https://youtu.be/08wZN-xh6mY.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

Design by contract, automatic testing tools, dataset

1 INTRODUCTION

Python is now one of the most popular programming languages in the world due to its simple syntax, extensive support modules and active community. Nevertheless, Python's relatively dynamic nature is likely to be one of the factors that contribute to it being deemed less suitable for the backbone of software systems. Design-by-contract [16] which requires developers to write precise and verifiable specifications for software components, is one of the techniques able to improve code robustness. Though interest in contracts for Python dates back to early 2000s [27], fully

functional libraries for Python contracts such as dpcontracts [12], icontract [20] and deal [25] sprout out only in about last five years. The library support for writing contracts in Python was lacking, and changed only recently with libraries such as icontract [20]. Consequently, there are a few tools to help developers confirm (or refute) code contracts in Python, e.g., CrossHair [22] and icontract-hypothesis [19]. While various datasets capture Python programs with bugs [14, 24, 28], to the best of our knowledge, there exist no dataset of Python programs annotated with contracts that can be used to evaluate or benchmark these Python-specific tools due to little Python code with contracts in open-source projects.

In this paper, we present the Python-by-contract dataset - a novel collection of Python programs annotated with contracts. The programs solve problems spanning a wide range of computer science domains, from simple string manipulation to file operations. The dataset contains both correct implementations and ones with bugs that violate a contract. Besides, we carefully curate the bugs so that the correct and the incorrect programs have only minimal differences, which makes it easier to confirm the presence of a genuine bugs and debug problems in testing tools. Furthermore, we show that our Python-by-contract dataset can be used to evaluate and help the development of existing testing tools. Our Python-by-contract dataset is publicly available at https://github.com/mristin/python-by-contract-corpus.

The contribution of this work is thus three-fold: 1) We manually write Python solutions to exercises to construct the Python-by-contract dataset. 2) We explicitly annotate the Python functions and classes with contracts. 3) We carefully curate the bugs detected during developing to make tracing downstream defects easier and include them in the dataset.

2 DATASET CONSTRUCTION

In this section, we first introduce the collection of programming exercises that underlie our Python programs. Then we describe our development of the Python programs and contracts, as well as how we remold the bugs recorded during the development into the *incorrect programs*.

2.1 Data Source

As the Python programs in our dataset serve as the benchmark for different testing tools, they should not involve complicated dependencies. It should be easy for users to reason about them and trace the potential bugs. Due to the recency of the library support for contracts in Python, publicly available programs with contracts

are scarce, complex and involve many dependencies, thus making them unsuitable as a dataset. We resort to manually writing Python programs as solutions to two public collection of programming exercises: Advent of Code 2020 [26] (**AoC 2020**) and the exercises for the lecture "Introduction to Programming" at ETH Zurich in Fall 2019 [9] (**ETHZ Eprog 2019**).

AoC 2020. Advent of Code is an annual set of computer programming challenges that follow an Advent calendar since 2015. The programming puzzles include a variety of skill sets and skill levels and can be solved using any programming language [26].

ETHZ Eprog 2019. "Introduction to Programming" course at ETH Zurich aims to teach students to systematically develop simple programs in Java. The exercises cover a wide scope of programming concepts, such as basic data structures, iterative and recursive algorithms, file operations, etc. [9].

These two sources are chosen because they are publicly available, cover a wide enough spectrum of problems, and sufficient amount of alternative solutions can be easily obtained for comparison. More importantly, the solutions do not have complicated dependencies and thus are easy for tracing of bugs.

2.2 Selection of ETHZ Eprog 2019 Exercises

Not all problems from ETHZ Eprog 2019 were suitable for our dataset. We evaluated the problems and selected only the most relevant ones according to the following steps. We exclude:

(1) Non-programming exercises: exercises which do not require writing code are removed, such as exercises that introduce students to version control systems.
(2) Exercises specific to Java: since we want to build a Python dataset, exercises focusing on Java-specific topics such as Java-specific constructors are removed.
(3) Trivial exercises: problems considered too trivial are removed, such as those about basic input/output operations.
(4) Technology-specific exercises: this includes, e.g., exercises about the graphical user interfaces (GUIs). Although contracts are useful in the GUI programming, we consider GUI programming to be too technology-specific and thus outside the scope of the dataset.

Additionally, some exercise statements are simplified to make for a more pointed code. We explicitly mark the corresponding changes in the description of the problem in Python files.

We provide the complete list of removed and simplified exercises with the rationale on our dataset website[1].

2.3 Dataset Construction Process

All the programs in our dataset are written by 2 experienced developers with 5-10 years of Python programming experience. Each Python file is the solution to one programming exercise with functions annotated with contracts using `icontract` [20] library. Following the common practice in industry, both the code and the contracts were written by the same developer.

Figure 1 demonstrates the data construction process. For each exercise, we iteratively develop the solution and the contracts
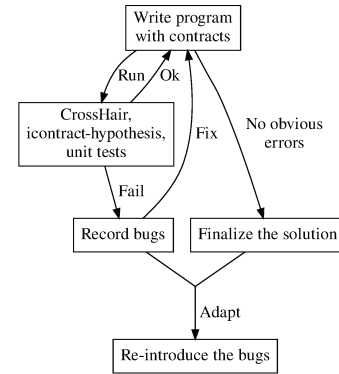

Figure 1: The flow of our data construction process.

with the help of two testing tools, `cross-hair` and `icontract-hypothesis`, as well as manually written unit tests. Specifically, we first write the solution as Python functions or classes together with the contracts for them. We follow general software engineering advice on writing contracts: we fully specify the preconditions, while we specify the postconditions and class invariants based on best effort. Then we check the correctness of our Python code either by the automatic testing tools or the unit tests. The solutions that violate the contracts or fail to pass the tests are recorded as bugs. The developer keeps modifying them to get the final correct solutions which pass all the checks and contain no obvious errors. The bugs are further adapted by re-introducing them to the final correct solution to obtain a minimally different incorrect programs. Note that although the contracts may have mistakes (i.e., they may not correctly encode the problem description); our expected use cases for this dataset do not depend on full contract correctness.

2.4 Incorrect Programs

As we develop the solutions to the exercises, we record bugs detected by the automatic testing tools or manually written unit tests. The buggy programs captured *during* the development often blur the cause of errors and diverge substantially from the final correct solution. This is suboptimal since we need to clearly distinguish between expected, collateral bugs, and the possible errors of testing tools we use. To provide a better and more precise testbed, we convert the recorded bugs into minimal changes of final correct solutions. Specifically, we first manually inspect each recorded bug to ensure that it is not caused by a defect in the tools. Then we re-introduce the bug into the final correct solution such that the buggy program is curated to be minimally different to the final correct solution. This makes it easier to confirm the presence of a genuine bug and provides succinct test cases for the testing tools. The resulting curated bugs are kept in the dataset as the *incorrect programs*. We provide an example to illustrate the process of curating a recorded bug to an incorrect program in Figure 2.

Although we are able to re-introduce most of the recorded bugs, some cases are not suitable for re-introduction. This is either because the recorded bugs are not informative enough or because the final correct solution ends up diverging too much from the buggy one to be re-introduced in a meaningful manner. Namely,

---

[1]https://python-by-contract-corpus.readthedocs.io/en/latest/correct_programs/ethz_eprog_2019/details.html

a)
```
# ERROR:
# bus_id and min_time should be reversed.
wait_time = bus_id % min_time
return min_time + wait_time
```

b)
```
# CORRECT:
missed_last_bus_by = min_time % bus_id
if missed_last_bus_by == 0:
    return min_time
else:
    return (
        min_time - missed_last_bus_by +
        bus_id
    )
```

c)
```
# ERROR:
# bus_id and min_time should be reversed.
missed_last_bus_by = bus_id % min_time
if missed_last_bus_by == 0:
    return min_time
else:
    return (
        min_time - missed_last_bus_by +
        bus_id
    )
```

Figure 2: An example how a) the recorded error `wrong_mod.py` from AoC 2020, Day 13 is combined with the b) final solution to produce a minimally different c) incorrect program. Evidently, the renaming of the variable (`wait_time` to `missed_last_bus_by`) as well as the non-zero check (`if missed_last_bus_by == 0`) have been added after the recorded error, but replicated in the incorrect program. This makes the incorrect program minimally different from the final solution, while we keep the relevant bug.

Table 1: Statistics of the Python-by-contract dataset. Func. represents Python function; Pre-C. represents precondition; Post-C. represents postcondition; Inv. represents class invariant.

# Files	LOC	# Class	# Func.	# Pre-C.	# Post-C.	# Inv.
55	4,796	114	514	246	269	9

Table 2: Distribution of preconditions and postconditions.

	Univ. Q.	Bound	Pattern	Misc.
Precondition	36	49	27	134
Postcondition	44	10	3	212

the initial and the final solutions pursue different directions as we had to completely change the approach and re-model the problem with different abstractions. In other cases, the bugs result from the under-specification of the exercise itself. We consequently ignore such cases though they represent valid bugs.

3 PYTHON-BY-CONTRACT DATASET

Our dataset contains 1) Python solutions to the AoC 2020 annotated with contracts; 2) Python solutions to the ETHZ Eprog 2019 annotated with contracts; 3) Python incorrect programs with minimal difference to the solutions to AoC 2020 and ETHZ Eprog 2019.

Table 1 shows the statistics of the correct programs in the Python-by-contract dataset. Since a precondition is given as a set of conjunctions, the total number of preconditions is calculated as the sum of all the conjunctions. Our dataset consists of 55 Python files as the solutions to the exercises and 59 Python files as the incorrect programs. There are more incorrect than correct programs since for each exercise we find zero, one or multiple bugs during the development.

The correct programs are broken down into manageable chunks, so a file on average contains 9.3 functions. For a large fraction of the functions we could write the contracts. Among them, 37.7% are annotated with at least one precondition or postcondition. The remaining functions dealt with general inputs, so no preconditions were necessary, or no postconditions could be defined with meaningful effort or sufficient readability. We specified very few class invariants because the natural solutions did not usually employ mutable classes.

We classify the preconditions and postconditions in the correct programs into four categories and show the distribution in Table 2. Univ. Q. means the condition involves universal quantifiers ("*for all*"). Bound means the condition specifies the boundaries on a value. Pattern means the condition defines a matching of a regular expression pattern. The remaining are regarded as miscellaneous. As shown in Table 2, our dataset covers different kinds of preconditions and postconditions evenly. This contrasts previous datasets, where the basic checks (e.g., non-null check) dominate [23]. Moreover, the complexity shifts from the preconditions (with a substantial mass in "Bound" and "Pattern") to postconditions (with much less respective mass). This is expected as the functions and the underlying problems in our dataset are general. Restricting the inputs of a general function is often easier than checking the validity of the result of the function.

4 USE CASES

While different uses of our dataset are possible, we demonstrate its utility by examining how it is used to evaluate and aid the development of two testing tools, CrossHair [22] and icontract-hypothesis [19].

4.1 Use Case: CrossHair

CrossHair [22] is a concolic [10] testing tool. It uses a constraint solver to confirm or refute properties for symbolic inputs over concrete execution paths. Unlike most concolic execution tools that analyze binary executables, CrossHair models the Python language itself. CrossHair natively checks contracts in a variety of formats, including the contract format used in our dataset: icontract.

Application. We ran the "crosshair check" command both over the solution files and the files with recorded bugs. This revealed bugs in both CrossHair and the dataset, and provided valuable insight into CrossHair's performance.

Insights. The Python-by-contract dataset helps improve the stability of CrossHair. In particular, some contracts triggered fatal errors in CrossHair. We diagnosed and fixed two CrossHair bugs before completing a fully successful check of the solutions. The two bugs pertained to regular expression matching on sliced strings and directive parsing for some multi-line strings.

After applying the fixes above, we checked the correct solution files. Though one might expect to find no failures, `CrossHair` found 23 counterexamples. With extended timeout settings, `Cross-Hair` is able to find 7 additional counterexamples. The counterexamples largely pointed to omitted preconditions, rather than bugs in the solution itself. That said, in many cases the problems are under-specified, and a reasonable fix can have been made to either the contracts or the code underneath. Finally, the dataset includes 59 Python files that contain known bugs, most of which are revealed by contracts. Since `CrossHair` is presently able to find counterexamples in some correct solutions, we limit our analysis to incorrect programs for which the corresponding correct solution passes `CrossHair`'s checks: 35 of those 59. `CrossHair` successfully found counterexamples in 26 out of those 35.

4.2 Use Case: `icontract-hypothesis`

The `icontract-hypothesis` [19] is a testing tool that infers strategies for random generation of function's input. The generated input is supplied to the function under test, while the function's post-conditions serve as a test oracle checking the correctness. The `hypothesis` [15] library is used to generate the data given the inferred strategies. At current version (1.1.7), the strategies are inferred based on a basic set of patterns matched against the preconditions (lower/upper bounds and regular expressions). The generated data is further filtered by unmatched preconditions (i.e., reduced through rejection sampling).

Application. We manually selected function points (i.e., functions or methods of classes) for which the tool can infer a feasible generation strategy for the input. Out of total 514 function points in the dataset, we are able to cover 67 points for which `icontract-hypothesis` can directly be applied. We can test 10 further points by wrapping them in more restrictive preconditions, since their original preconditions were too permissive and resulted in valid, but practically inexecutable strategies (e.g., functions on strings with a length argument for which the strategy generates excessively large numbers). The remaining points can not be tested as the inferred strategies are computationally prohibitive since the generated data are almost constantly rejected.

Insights. We are positively surprised that such a small testing surface results in a high code coverage. We can thus cover 76% of code statements (with negligible standard deviation over 10 runs, <1%). As it turned out, simple preconditions were enough to narrow down the generation strategies such that consequent rejection sampling by the further complex preconditions did not incur too high computational cost. It is important to note that we heavily apply the recipe for deduplicating the preconditions by refactoring them into separate classes [21] in the dataset instead of writing conditions with `all(.)` quantifiers. This generalizes well throughout the programs, and largely explains the simplicity of the preconditions. Hence, as long as this recipe can be applied to keep the preconditions simple, we expect other code bases to achieve similar levels of code coverage with `icontract-hypothesis`.

5 RELATED WORK

Contracts. Thinking about program correctness in terms of contracts goes back to seminal works of Sir Hoare [11], Floyd [8] and

Naur [17]. Eventually, the theory flowed into practice of writing the contracts in the code to be checked either statically, at compile-time, or at runtime. This approach found wide popularity with Eiffel language [16], followed by Java [13], C# [7] and others.

Contracts in Python. Libraries for writing contracts for Python like dpcontracts [12], `icontract` [20] and deal [25] appear in the recent five years. Because of few libraries for python contracts, there is also only a nascent community around tools for ensuring correctness of practical Python programs based on contracts such as `CrossHair` and `icontract-hypothesis`.

Datasets of programs with contracts in Python. Because of the recency of the library support, there is impractically little Python code with contracts in the wild that can be collected by researchers. While various datasets capture programs with bugs, such as BugsInPy [28], BugSwarm [24], QuixBugs [14], they contain no contracts. This makes them thus impractical for tool developers which need insights into kinds of contracts used in practical programs as well as an appropriate testbed (see below).

Corpora of programs with contracts in other languages. In contrast to Python, there are many large program datasets with contracts in other languages. Notably, `EiffelBase`, the Eiffel's standard library, has been meticulously written contracts-first [4]. Since contracts are a core feature of the Eiffel language, solutions to school exercises usually contain contracts similar to our dataset [2]. Others looked into datasets to assess how programmers write contracts in practice. Chalin [3] analyzed a dataset of 85 Eiffel projects and 8M lines of code (LOC). Estler *et al.* [6] combed a suite of projects in Eiffel, C# and Java with high contract usage totaling 260M LOC. Shiller et al. [23] compiled a dataset of 90 C# programs listed on OpenHub of around 3.5M LOC, while Dietrich et al. [5] studied 200 most popular Java projects in Maven Central, and Casalnuovo *et al.* [1] looked into 100 most popular C/C++ projects on GitHub. Nie et al. [18] proposed a framework, Deuterium, for implementing Java methods as executable contracts and created a new benchmark for evaluating the executable contracts. Though our dataset is smaller in scope and volume, we see it as a valuable family member of the aforementioned corpora in other languages and hope it to be a first step towards this goal for Python community. As the design-by-contract gains more traction in the community, we hope that larger and more complex code bases with contracts will emerge.

6 CONCLUSION

In this paper, we present the Python-by-contract dataset containing both correct Python programs and curated incorrect Python programs with contracts. The Python programs in the dataset cover a wide variety of programming concepts with different types of contracts. We show that it can be utilized by researchers and practitioners for developing and evaluating correctness and testing tools. Our dataset is the first step in this direction, and we plan to keep expanding it in the future.

ACKNOWLEDGMENTS

REFERENCES

[1] Casey Casalnuovo, Premkumar T. Devanbu, Abílio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert Use in GitHub Projects. In *International Conference on Software Engineering*. 755–766. https://doi.org/10.1109/ICSE.2015.88

[2] ETH Zurich Chair of Software Engineering. 2015. Introduction to Programming 2015. http://se.inf.ethz.ch/courses/2015b_fall/eprog/english_index.html#downloads. [Online; accessed 15-August-2022].

[3] Patrice Chalin. 2006. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*. 100–113.

[4] Eiffel Community. 2022. The EiffelBase library 22.05. https://www.eiffel.org/doc/solutions/EiffelBase. [Online; accessed 20-July-2022].

[5] Jens Dietrich, David J. Pearce, Kamil Jezek, and Premek Brada. 2017. Contracts in the Wild: A Study of Java Programs. In *European Conference on Object-Oriented Programming*. 9:1–9:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.9

[6] H. Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. 2014. Contracts in Practice. In *FM 2014: Formal Methods*. Springer International Publishing, Cham, 230–246. https://doi.org/10.1007/978-3-319-06410-9_17

[7] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. 2010. Embedded Contract Languages. In *Symposium on Applied Computing*. 2103–2110. https://doi.org/10.1145/1774088.1774531

[8] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf

[9] ETH Zurich Laboratory for Software Technology. 2019. Einführung in die Programmierung Herbst 2019. https://www.lst.inf.ethz.ch/education/archive/Fall2019/einfuehrung-in-die-programmierung-i--252-0027-.html.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *Conference on Programming Language Design and Implementation*, 213–223. https://doi.org/10.1145/1065010.1065036

[11] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

[12] Rob King. 2015. dpcontracts - An implementation of contracts for Python. https://github.com/deadpixi/contracts. [Online; accessed 20-July-2022].

[13] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming* 55, 1 (2005), 185–208. https://doi.org/10.1016/j.scico.2004.05.015

[14] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 55–56. https://doi.org/10.1145/3135932.3135941

[15] David R. MacIver, Zac Hatfield-Dodds, and many other contributors. 2019. Hypothesis: A new approach to property-based testing. https://doi.org/10.21105/joss.01891 [Online; accessed 31-August-2022].

[16] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[17] Peter Naur. 1966. Proof of algorithms by general snapshots. *BIT Numerical Mathematics* 6 (1966), 310–316.

[18] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying execution of imperative generators and declarative specifications. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–26.

[19] Marko Ristin. 2017. icontract-hypothesis - Combine contracts and automatic testing. https://github.com/mristin/icontract-hypothesis. [Online; accessed 20-July-2022].

[20] Marko Ristin. 2018. icontract - Design-by-contract in Python3 with informative violation messages and inheritance. https://github.com/Parquery/icontract. [Online; accessed 20-July-2022].

[21] Marko Ristin. 2021. icontract documentation - Recipes. https://icontract.readthedocs.io/en/latest/recipes.html#encapsulation-of-immutable-types. [Online; accessed 20-July-2022].

[22] Phillip Schanely. 2017. CrossHair - An analysis tool for Python that blurs the line between testing and type systems. https://github.com/pschanely/CrossHair. [Online; accessed 20-July-2022].

[23] Todd W Schiller, Kellen Donohue, Forrest Coward, and Michael D Ernst. 2014. Case studies and tools for contract specifications. In *International Conference on Software Engineering*. 596–607. https://doi.org/10.1145/2568225.2568285

[24] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *International Conference on Software Engineering*. 339–349. https://doi.org/10.1109/ICSE.2019.00048

[25] Nikita Gram Voronov. 2018. deal - Design by contract for Python. https://github.com/life4/deal. [Online; accessed 20-July-2022].

[26] Eric Wastl. 2020. Advent of Code 2020. https://adventofcode.com/2020/about. [Online; accessed 21-July-2022].

[27] Terrence Way. 2003. PEP 316 - Programming by Contract for Python. https://peps.python.org/pep-0316/. [Online; accessed 20-July-2022].

[28] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1556–1560. https://doi.org/10.1145/3368089.3417943