







# Introducing PYRA: A High-level Linter for Data Science Software

Greta Dolcetti<sup>1</sup>, Vincenzo Arceri<sup>2</sup> (✉), Antonella Mensi<sup>3</sup>, Enea Zaffanella<sup>2</sup>, Caterina Urban<sup>4</sup>, and Agostino Cortesi<sup>1</sup>

<sup>1</sup> Ca' Foscari University of Venice, Via Torino 155, 30170 Venice, Italy  
`{greta.dolcetti|cortesi}@unive.it`

<sup>2</sup> University of Parma, Parco Area delle Scienze 53/A, 43124 Parma, Italy  
`{vincenzo.arceri|enea.zaffanella}@unipr.it`

<sup>3</sup> University of Verona, Piazzale L. A. Scuro 10, 37134 Verona, Italy  
`antonella.mensi@univr.it`

<sup>4</sup> Inria & École Normale Supérieure | Université PSL, Paris, France  
`caterina.urban@inria.fr`

**Abstract.** We present PYRA, a static analysis tool that aims at detecting code smells in data science workflows. Our goal is to capture potential issues, focusing on misleading visualizations, challenges for reproducibility, as well as misleading, unreliable or unexpected results.

Link to the demo: <https://www.youtube.com/watch?v=D-AsyuhTsYo>

GitHub repository: <https://github.com/spangea/Pyra>

**Keywords:** Static analysis · Code smells · Data science · Jupyter Notebooks · Python

## 1 Introduction

In this demo, we present PYRA, a high-level linter for data science software. PYRA is a static analysis tool that helps developers identify potential issues in their data science code written in Python. PYRA focuses primarily on code smells, aiming at capturing anti-patterns that, although not raising a warning due to Python's inherent flexibility, can result in potential issues for the data science pipeline being implemented.

PYRA is inspired by the pervasiveness and versatility of data science software, which is often applied in interdisciplinary fields. Due to this nature, many projects [4,1,3] aim at easing and making the development of data science software more reliable: yet, they often require a huge effort by the users (such as manually annotating program variables) or they focus either on the data or general best practices for the code, but not on the combination of both. Conversely, PYRA is designed to be easy to use and integrates seamlessly with Python code, without requiring any additional annotations or modifications of the code. Moreover, our goal is to infer and reason about more abstract datatypes, potentially capturing a broader and less conventional set of code smells. Therefore we propose an easily extensible framework to help developers achieve correct results.

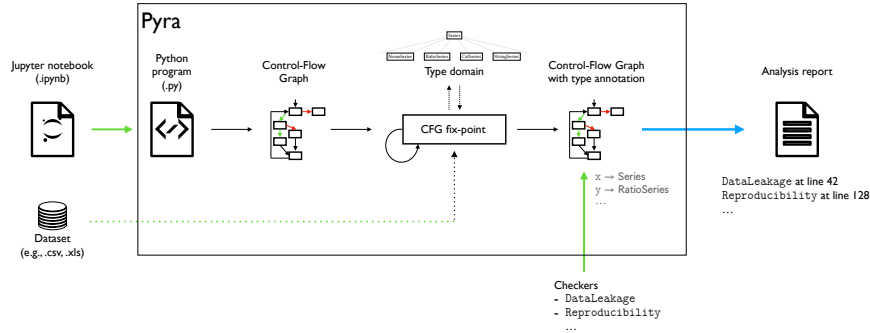


Fig. 1. PYRA high-level execution flow.

## 2 PYRA’s Architecture

The high-level execution flow of PYRA is reported in Fig. 2. PYRA takes as input a Jupyter Notebook and produces a report containing the detected code smells and the results of the analysis. First, the notebook is converted into a Python script; then, PYRA performs static analysis on the Control Flow Graph (CFG) extracted from the script, computing a fixpoint over the CFG to infer abstract type information for each variable at each program point; finally, a set of checkers is applied to the annotated CFG to detect code smells.<sup>5</sup>

The 55 abstract datatypes that PYRA can infer extend those described in [2]: some correspond to concrete datatypes (e.g., `List`, `DataFrame`, `Series`), others are more abstract (e.g., `Numeric`, which can represent either a `float` or an `int`), and some are specific to data science (e.g., encoders, scalers, standardized/normlized `Series`). Such information is exploited by the checkers to identify code smells in the input Python script. The rationale behind the checkers is intuitive: the static analysis computes and propagates types while maintaining the abstract type environment  $\Gamma$ ; whenever a procedure associated with one of the code smells is encountered, the analyzer uses the information in  $\Gamma$  to determine whether it might represent a code smell; if so, it raises a warning and provides the user with a description of the issue, its cause, and a suggestion about how to fix it. We grouped the code smells and their corresponding checkers into four categories:

- **Misleading visualizations:** issues that can compromise data interpretation due to inappropriate visualization choices, such as using line plots for categorical data or applying PCA for visualization when more suitable techniques like t-SNE could reveal clearer patterns.
- **Misleading results:** issues that can lead to incorrect/biased outcomes without raising exceptions. Examples include data leakage (e.g., pre-processing

<sup>5</sup> PYRA can optionally use the information about the concrete dataset used in the notebook, if provided, to improve the accuracy of the analysis.

before the train-test split), improper PCA usage, failure to remove duplicates, and poor handling of missing values.

- **Challenges for reproducibility:** issues that can hinder the reproducibility of data science pipelines, such as the omission of random seed (`'random_state'`) settings in operations involving randomness (e.g., train-test split).
- **General issues** such as high dimensionality (too many features vs. samples), which may lead to the curse of dimensionality, and assignment of the result of in-place operations that can cause unexpected behavior.

Currently, PYRA includes 16 different checkers for detecting code smells. In this demo, we highlight four of them, one per category, to demonstrate PYRA's core capabilities.

### 3 Demo

We demonstrate PYRA by analyzing the code shown in Fig. 2. The code represents a simple data science pipeline that reads a CSV file, drops duplicates, plots the data, scales it, splits it into training and testing sets, and fits a logistic regression model. The dataset contains three columns: 'Fruit' (categorical), 'Amount' (integer), and 'Label' (0 or 1). This code, although short, contains several issues that belongs to the four identified categories of code smells, specifically:

- The `drop_duplicates` method is called with `inplace=True`, which modifies the `DataFrame` in place and returns `None`. This can lead to confusion, as the variable `result` will be assigned `None`.
- The `plot` method is used to create a line plot with a categorical  $x$ -axis. This is inappropriate, as line plots are typically used for continuous data. A bar plot would be more suitable in this case.
- The `train_test_split` method is called without setting the `random_state` parameter, meaning the split will differ each time the code is run. This can result in non-reproducible outcomes.
- The data is scaled before the train-test split. This can cause data leakage, as the scaling parameters are computed using the entire dataset, including the test set. The scaling should be performed after the split to avoid this issue.

PYRA detects these issues and raises warnings, including the type of the code smell, its cause, and suggestions for fixing it.

### 4 Conclusion

In this demo, we presented PYRA, a high-level linter for data science software. PYRA is designed to help developers identify code smells in the data science software. By analyzing code and issuing warnings, PYRA supports the development of more robust and reliable data science pipelines. The demo showcased how PYRA can analyze a data science pipeline and identify several issues, offering valuable feedback and suggestions for improvement. The presented tool

```

In [1]: import pandas as pd
import matplotlib.pyplot as plt
from sklearn import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

df = pd.read_csv("data.csv")
# Columns: ['Fruit', 'Amount', 'Label']
result = df.drop_duplicates(inplace=True)

plt.plot(df["Fruit"], df["Amount"])

scaler = StandardScaler()
X_scaled = scaler.fit_transform(df[["Amount"]])

X_train, X_test, y_train, y_test =
    train_test_split(X_scaled, df["Label"])

model = LogisticRegression()
model.fit(X_train, y_train)

```

**Fig. 2.** Example of a pipeline with a plotting issue due to a categorical  $x$ -axis and an in-place DataFrame modification that results in a `None` return from `drop_duplicates`.

stands out from other static analysis tools or linters by specifically addressing the unique challenges and pitfalls of data science software, making it a valuable addition to the data science ecosystem. Moreover, PYRA can be easily integrated into existing IDEs as a plugin, providing real-time feedback to developers as they write their code. This feature makes the tool especially useful, given that our target audience includes not only experienced data scientists but also beginners and experts from other fields who may not be familiar with the peculiarities, routines and best practices of data science software.

**Acknowledgments.** Work partially supported by Bando di Ateneo 2024 per la Ricerca, funded by University of Parma (FIL\_2024\_PROGETTI\_B\_IOTTI - CUP D93C24001250005) and SERICS (PE00000014 - CUP H73C2200089001) project funded by PNRR NextGeneration EU.

## References

1. Bantilan, N.: pandera: Statistical data validation of pandas dataframes. In: Agarwal, M., Calloway, C., Niederhut, D., Shupe, D. (eds.) Proceedings of the 19th Python in Science Conference 2020 (SciPy 2020), Virtual Conference, July 6 - July 12, 2020. pp. 116–124. scipy.org (2020). <https://doi.org/10.25080/MAJORA-342D178E-010>
2. Dolcetti, G., Cortesi, A., Urban, C., Zaffanella, E.: Towards a high level linter for data science. In: Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2024, Co-located with SPLASH 2024. p. 18 – 25 (2024). <https://doi.org/10.1145/3689609.3689996>
3. Quaranta, L., Calefato, F., Lanubile, F.: Pynblint: a static analyzer for python jupyter notebooks. In: Crnkovic, I. (ed.) Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, CAIN 2022, Pittsburgh,

- Pennsylvania, May 16-17, 2022. pp. 48–49. ACM (2022). <https://doi.org/10.1145/3522664.3528612>
4. Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. pp. 683–710. Lecture Notes in Computer Science, Springer (2018). [https://doi.org/10.1007/978-3-319-89884-1\\_24](https://doi.org/10.1007/978-3-319-89884-1_24)