Chasten Your Python Program: Configurable Program Analysis and Linting with XPath

Gregory M. Kapfhammer

July 26, 2025

Access resources during the talk!

Scan the QR code to follow along

What is chasten? Why did we build it?

- Configurable **program analysis** and **linting** with XPath expressions
  - Avoid a unique performance anti-pattern?
  - Confirm the use of a new coding style?
  - Nervous about writing custom AST visitors?
  - Need configuration and data storage of results?

Chasten helps you automatically detect patterns in Python programs

Developers

- Project-specific checks
- Avoid code anti-patterns
- Facilitate code reviews

Students

- Explore different code style
- Avoid performance problems
- Confirm project criteria

Researchers

- Count code patterns
- Measure code quality
- Easily share results

Educators

- Give early feedback on code style
- Enforce assignment requirements
- Support use on laptops and in CI

Example: students and educators using `chasten` for a Python project

- Students may struggle to write **efficient** and **readable** Python code
- Manual review by instructors is **time-consuming** and **error-prone**
- Regex is **brittle** and AST-based tools are **hard to prototype**

Project Goal: `chasten` enables **scalable** and **structure-aware** feedback that effectively **supports** both **instructors** and **students**

Take a Step Back: Before diving into the implementation of `chasten`, it's worth surveying the landscape of **linting** and **checking**

Many Trade-Offs: Different tools with varying implementation, features, performance, and extensibility! Which one(s) to pick?

Building a source code analyzer! What are the options and trade-offs?

Regular Expressions

- Easy to write and try out
- Often brittle and confusing

Ruff

- Fast and easy to use
- No extension mechanism

Pylint and Flake8

- Extensible with plugins
- Must have AST knowledge

Treesitter and Ast-Grep

- Configurable with patterns
- Less support for tool building

Wow, `pyastgrep` offers a novel way query a program's AST! Is XPath sufficient? Can this tool support all envisioned use cases? How?

Wait, what is an abstract syntax tree?

Python Source Code

```python
def calculate_sum(x, y):
  """Add two numbers."""
  return x + y
```

Abstract Syntax Tree

```
Module(
  body=[
    FunctionDef(
      name='calculate_sum',
      args=...,
      body=[
        Return(
          value=BinOp(
            left=Name(id='x', ...),
            op=Add(),
            right=Name(id='y', ...)))],
      ...)],
  ...)
```

Understanding the AST

- Tree representation of code
- Nodes are syntax elements
- Great for program analysis
- Independent of code style

AST Analysis Challenges

- Complex structure for code
- Brittle regular expressions
- False positives and negatives
- Need easy way to query
- Avoid bespoke solutions
- Adopt XPath-like queries

Scanning code with `pyastgrep`

Define a Python file with functions

```python
def too_many_args(a, b, c, d, e, f):
def another_function(x, y):
def a_third_function(p, q, r, s, t, u, v):
```

Find functions with more than 5 arguments

```
pyastgrep '//FunctionDef[count(args/args) > 5]' example.py
```

Results from running the query with `pyastgrep`

```
example.py:1:1:def too_many_args(a, b, c, d, e, f):
example.py:7:1:def a_third_function(p, q, r, s, t, u, v):
```

Make the connection by comparing the `pyastgrep` and `chasten` tools

`pyastgrep`

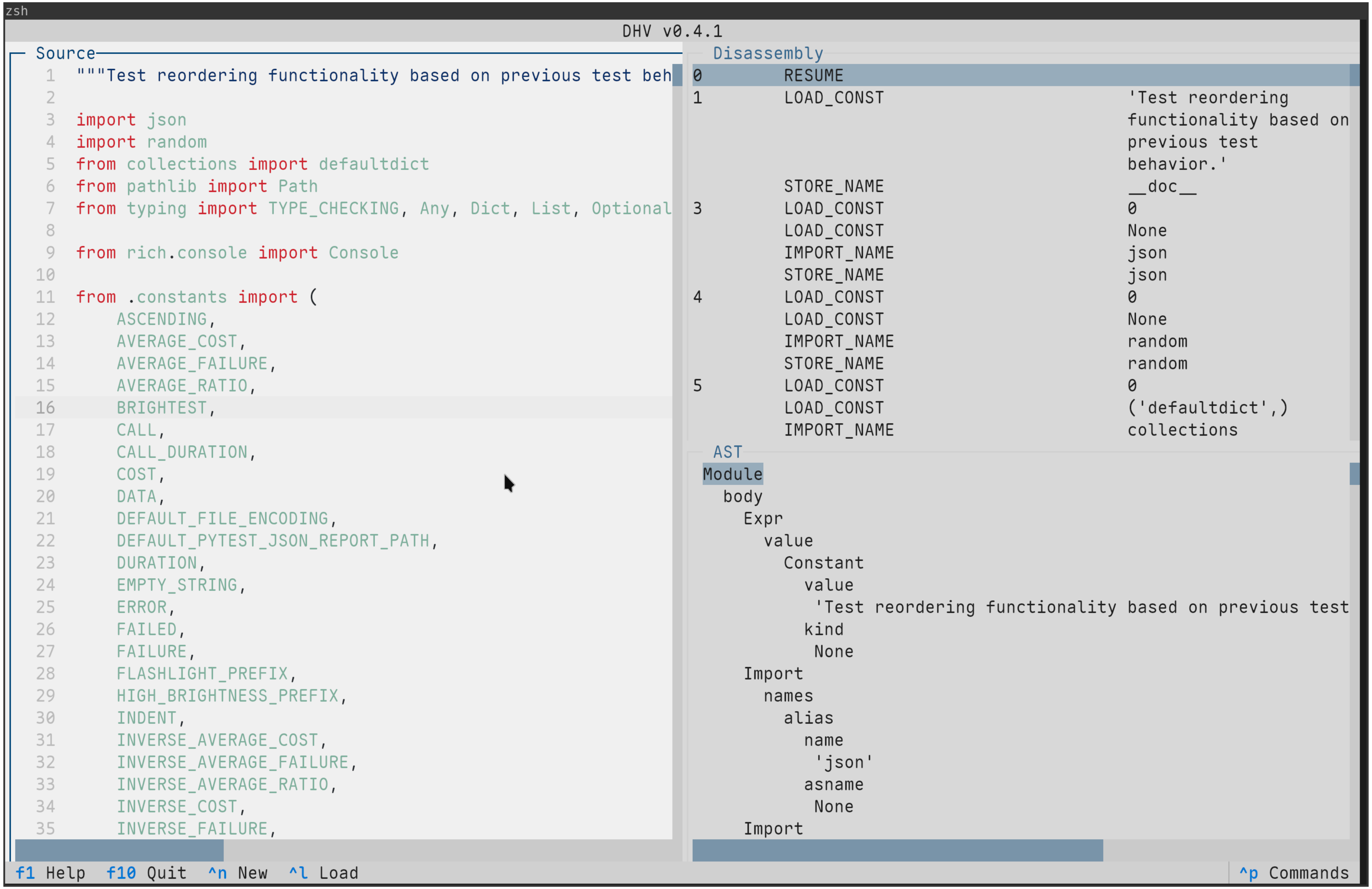
- Interactive AST search tool
- Ad-hoc queries from the CLI
- Uses raw XPath expressions
- `grep`-like console output

`chasten`

- Built using `pyastgrep`'s API
- Runs checks from a YAML file
- Saves results in JSON, CSV, DB
- View results with `datasette`

Key Idea: `chasten` uses `pyastgrep`'s powerful search to build a configurable, project-oriented linter. Developers, researchers, students, and instructors can "`chasten`" Python projects and save the results!

Use dhv to explore a Python AST!



Quick recap of referenced projects

Click these links to preview documentation for referenced tools!

- Python `ast` module: Python's abstract syntax tree module
- Pylint: A popular static code analyzer for Python
- Flake8: An extensible wrapper around PyFlakes, pycodestyle, and McCabe
- Ruff: An extremely fast Python linter and code formatter, written in Rust
- Tree-sitter: A parser generator tool and incremental parsing library
- Ast-grep: A CLI tool for searching and rewriting code with ASTs
- Pyastgrep: A tool for searching Python code with XPath expressions
- Dhv: A comprehensive TUI for Python code exploration built with Textual
- Datasette: A SQL-based tool for exploring and publishing data to the web

Next Steps: Use case for Python project analysis with `chasten`

Avoid time complexity of $O(n^2)$

```python
# O(n) is acceptable
seen = set()
for item in items:
    if item in seen:
        return True
    seen.add(item)
```

```python
# O(n²) is not okay
for i in range(len(items)):
    for j in range(len(items)):
        if i != j
         and items[i] == items[j]:
            return True
```

- **Goal**: Automatically scan the source code that students submit to confirm that there are no inefficient looping constructs
- **Challenge**: Linters like Ruff and Pylint don’t have rules to detect nested control structures that either are or are not acceptable
- **Build**: An extensible tool allowing instructors to scan for arbitrary code patterns without detailed AST knowledge

Chasten to the rescue!

- Uses XPath to search Python’s AST
- Rules written in simple YAML
- Structure-first, not just style
- Outputs to JSON, CSV, or SQLite

Result: Instructors define checks once and use Chasten to easily apply them at scale across all student submissions

```yaml
- name: "nested-loops"
  code: "PERF001"
  pattern: "//For[descendant::For]"
  description: "Detects doubly nested for-loops that are often O(n²)"
```

Let's run chasten!

Install the Tool

```
pipx install chasten  # Install Chasten in venv
pipx list             # Confirm installation
chasten --help        # View available commands
```

Run Chasten

```
chasten analyze time-complexity-lab \
        --config chasten-configuration \
        --search-path time-complexity-lab \
        --save-directory time-complexity-results \
        --save
```

- Save results to a **JSON file** and produce **console output**
- Configure the **return code** for different **detection goals**

Results from running `chasten`

Nested loop analysis

Check ID	Check Name	File	Matches
PERF001	nested-loops	analyze.py	1
PERF001	nested-loops	display.py	7
PERF001	nested-loops	main.py	0

Check ID → A unique short rule code (e.g., `PERF001`)

Check Name → The rule name that matched (e.g., `nested-loops`)

File → The Python file that the tool scanned (e.g., `analyze.py`)

Matches → Number of times the pattern was detected in that file (e.g., `1` match)

Exploring a bespoke AST visitor

```python
import ast
import json
import os
import sys

class ForVisitor(ast.NodeVisitor):
    """
    An AST visitor that detects doubly-nested for loops.
    """
    def __init__(self, filepath):
        self.filepath = filepath
        self.nested_for_loops = []
        self._for_depth = 0

    def visit_For(self, node):
        """
        Visit a for-loop node in the AST.
        """
        self._for_depth += 1
        if self._for_depth > 1:
```

What role should generative AI play in program analysis and `chasten`?

- The prior program was automatically generated by Gemini 2.5 Pro with `gemini-cli`. And, it works! Impressive!
- Similar programs can also be generated by GPT4.1 or Claude Sonnet 4 with `open-code`. Again, really nice!
    - `npx https://github.com/google-gemini/gemini-cli`
    - `npx opencode-ai@latest`
- Or, use these tools to generate `chasten` configurations!

Limitations and future directions

- **Limitations of the current version of `chasten`**
  - Doesn’t handle style, formatting, or type inference
  - Not optimized for fast use in continuous integration
  - Pattern matches through XPath on Python’s AST
- **Empirical study of `chasten`’s effectiveness and influence**
  - Frequency of false positives or false negatives?
  - How do students respond to the tool’s feedback?
  - Differences in scores with varied feedback types?

Chasten your Python program!

- Help developers, researchers, students, and educators
- Write declarative rules for AST-based code checks
- Focus on bespoke code structure patterns in Python
- Automated grading aligned with learning outcomes
- Generate data-rich insights into your code patterns

- **Try out Chasten and contribute to its development!**
  - GitHub: https://github.com/AstuteSource/chasten
  - PyPI: https://pypi.org/project/chasten/