complexipy: A Deep Dive into Code Readability

A Hands-On Workshop for PyCon Colombia 2025

Led by @rohaquinlop

About the Speaker

Robin Hafid Quintero Lopez (@rohaquinlop)

Software Engineer at GenLogs

- Rust Enthusiast
- Creator & maintainer of **complexipy**
- Contributed to `Rust`, `terraform-aws-gitlab-runner` and other open-source projects
- Speaker at local meetups
- Passionate about developer experience, performance, and readable code
- `C` - `C++` - `Python` - `Rust` - `Nix` - `Gleam` | `FP`

Workshop Overview

This workshop provides a comprehensive exploration of code complexity, transitioning from traditional metrics to the modern, more intuitive concept of Cognitive Complexity.

Learning Objectives

Upon completion, you will be able to:

- **Articulate** the difference between Cyclomatic and Cognitive Complexity
- **Analyze** Python code for cognitive complexity using the **complexipy** CLI and library
- **Implement** automated complexity checks in a CI/CD pipeline using GitHub Actions
- **Develop** strategies for refactoring high-complexity code to improve readability
- **Generate** and interpret complexity reports to guide code quality improvements

Prerequisites

- ✓ **Basic Python Knowledge**: Familiarity with Python syntax and data structures
- ✓ **Laptop with Python**: Python 3.9+ installed
- ✓ **uv (Optional)**: Have uv installed
- ✓ **Code Editor**: Your preferred code editor (e.g., VS Code)
- ✓ **Git**: Required to clone the workshop repo
- ✓ **GitHub Account**: For the GitHub Actions section

Workshop Structure

- **Module 1**: Foundations of Code Complexity
- **Module 2**: Introducing **complexipy**
- **Module 3**: Workflow Integration
- **Module 4**: Practical Refactoring & Conclusion

Module 1

Foundations of Code Complexity

1.1: The Business Case for Readable Code

Why does code complexity matter?

- **Reduced Maintenance Costs**: Complex code is expensive to maintain
- **Faster Onboarding**: New team members can understand code faster
- **Fewer Bugs**: Simpler code has fewer edge cases and failure modes
- **Better Collaboration**: Teams can work together more effectively

1.2: Beyond Cyclomatic Complexity

Limitations of Cyclomatic Complexity

- **Created 1976**: Useful for test coverage, not readability.
- **Path-focused**: Counts execution paths, ignores comprehension.
- **False alarms**: Penalizes obvious patterns, misses subtle ones.
- **Ignores modern Python**: `try/except`, async, lambdas, pattern matching.
- **Scale issues**: Totals grow with lines of code, not true complexity.

> **Takeaway**: Cyclomatic Complexity remains useful for test coverage planning, but it is an unreliable proxy for readability or maintainability.

1.3: Introduction to Cognitive Complexity

Created by G. Ann Campbell, primary author of the Cognitive Complexity metric

Why Cognitive Complexity?

A score for the mental effort required to read code.

- **Understands modern Python** (exceptions, lambdas, async, etc.)
- **Scales up smoothly** from line → function → module → app
- **Feels right** – numbers match our gut sense of readability

1.3: Introduction to Cognitive Complexity

How It's Scored

1. Skip simple language shortcuts.
2. +1 for every control-flow break (`if` , loops, `&&` , `||`).
3. +1 for each extra level of nesting.

Increment Types

- **Nesting** – inside another control-flow block.
- **Structural** – starts a new block.
- **Fundamental** – breaks flow without starting a block (`break` , `continue`).
- **Hybrid** – shifts nesting for what follows (`else` , `finally`).

> **Bottom line**: Mirrors how we read code and flags genuinely hard-to-read sections.

Code Sample Analysis

```
def sumOfPrimes(max: int) -> int:  # +1
  total = 0
  for i in range(1, max+1):        # +1
    should_add = True
    for j in range(2, i):          # +1
      if i%j == 0:                 # +1
        should_add = False

    if should_add:                 # +1
      total += i

  return total
```

Cyclomatic Complexity: 5

```
def getWords(number: int) -> str:     # +1
  match number:
    case 1:                           # +1
      return "one"
    case 2:                           # +1
      return "a couple"
    case 3:                           # +1
      return "a few"
    case 4:                           # +1
      return "some more values"
    case _:
      return "lots!"
```

Cyclomatic Complexity: 5

Code Sample Analysis

```python
def sumOfPrimes(max: int) -> int:
  total = 0
  for i in range(1, max + 1):# +1
    should_add = True
    for j in range(2, i):    # +2 (+1 itself, +1 nesting)
      if i % j == 0:         # +3 (+1 itself, +2 nesting)
        should_add = False

    if should_add:           # +2 (+1 itself, +1 nesting)
      total += i

  return total
```

Cognitive Complexity: 8

```python
def getWords(number: int) -> str:
  match number:
    case 1:
      return "one"
    case 2:
      return "a couple"
    case 3:
      return "a few"
    case 4:
      return "some more values"
    case _:
      return "lots!"
```

Cognitive Complexity: 0

Module 2: Introducing **complexipy**

2.1: **complexipy** - A Modern Solution

Project Goals & Architecture

Why complexipy?

- **Performance**: Rust-based
- **Accuracy**: Implements the Cognitive Complexity specification
- **Usability**: Simple CLI and Python library
- **Integration**: Works with existing tools and workflows

Architecture

- **Core Engine**: Written in Rust for performance
- **Python Bindings**: Easy integration with Python projects
- **CLI Interface**: Command-line analysis tool
- **Library API**: Programmatic access for custom tools

2.2: Command-Line Interface Deep Dive

Basic Usage

```
# Analyze a specific file
complexipy path/to/file.py

# Analyze specific directory
complexipy path/to/directory

# Ignore complexity threshold and show all functions
complexipy path/to/file.py -i

# Output results to a CSV file
complexipy path/to/directory -c

# Show only files exceeding maximum complexity
complexipy path/to/directory -d low

# Sort results in descending order
complexipy path/to/directory -s desc
```

2.3: Hands-On Lab: First Complexity Audit

Your Tasks

Setup

Go to https://github.com/rohaquinlop/complexipy-workshop and fork it.

```
git clone https://github.com/$your_user/complexipy-workshop.git
cd complexipy-workshop
```

If you have installed uv

```
uv venv
source .venv/bin/activate
uv sync --frozen
```

If not

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

2.3: Hands-On Lab: First Complexity Audit

Task 1: Install complexipy

```
uv add complexipy      # If you're using uv
pip install complexipy # If you're using pip
```

Task 2: Basic Analysis

```
complexipy .
```

Module 3: Integrating **complexipy** into Your Workflow

3.1: Visual Studio Code Integration

The **complexipy** VS Code Extension

Features

- **Real-time Analysis**: See complexity as you type
- **Visual Indicators**: Color-coded complexity levels

Installation & Setup

1. Search: `complexipy` at VS Code marketplace
2. Install

3.2: Automating Quality with GitHub Actions

The `complexipy-action`

```yaml
name: Check Code Complexity
on: [push, pull_request]

jobs:
  complexity:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: complexipy
      uses: rohaquinlop/complexipy-action@v2
      with:
        paths: src
```

Benefits

- **Automated Quality Gates**: Block PRs with high complexity
- **Team Awareness**: Everyone sees complexity trends
- **Historical Tracking**: Monitor complexity over time

3.3: Hands-On Lab: Development Workflow Integration

Lab Objectives

Part 1: VS Code Extension

1. Install the complexipy extension
2. Open a Python file with complex functions
3. Observe real-time complexity indicators
4. Try the quick-fix suggestions

Part 2: GitHub Actions Setup

1. Add the complexipy-action workflow
2. Push the changes and create a PR to the original repo
3. Observe the CI check failure

Module 4: Practical Refactoring & Conclusion

4.1: From Analysis to Action

Live Refactoring Session

Common Refactoring Patterns

Strategies for Reducing Complexity

1. Guard Clauses

- Return early for invalid conditions
- Reduces nesting levels
- Makes the happy path clearer

2. Extract Methods

- Break complex functions into smaller ones
- Each method has a single responsibility
- Improves readability and testability

3. Simplify Conditionals

- Use helper methods for complex boolean logic
- Replace nested ifs with early returns
- Consider using data structures instead of conditionals

4. Reduce Nesting

- Flatten nested structures where possible
- Consider alternative control flow patterns

4.2: The Future of **complexipy**

Roadmap & Future Features

- **Language Support**: Extending beyond Python
- **IDE Integration**: More editor plugins

Open Discussion

Questions to Consider

- How will you integrate complexity analysis into your workflow?
- How can complexity analysis improve your code review process?
- What challenges do you anticipate in adoption?

4.3: Q&A and Wrap-up

Key Takeaways

What We Covered

- **Cognitive vs Cyclomatic Complexity**: Understanding the difference
- **complexipy Tool**: CLI, library, and integrations
- **Workflow Integration**: VS Code and CI/CD automation
- **Practical Refactoring**: Real techniques for reducing complexity

Next Steps

- **Install complexipy** and start analyzing your code
- **Set up VS Code extension** for real-time feedback
- **Share knowledge** with your team

Resources for Continued Learning

- complexipy Documentation
- Cognitive Complexity Whitepaper

Thank You!

Questions & Discussion

Resources available at: github.com/rohaquinlop/complexipy-workshop

Slides available at: github.com/rohaquinlop/pycon-col-2025-slides