S3chool

Sustainable Scientific Software School

Quality Tools for Research Software

Practical tools for ensuring code quality in 2026 with a bias on Python

Press Space for next page →

eosc | EVERSE

OSCARS Open Science Clusters' Action for Research & Society

Funded by the European Union

Lecture Overview

Duration: 60 minutes
Target Audience: Research software developers, PhD students, postdocs, researchers

Topics

1. What are Quality Tools ?
2. A few recommended ones
3. Hands-on Exercises

Learning Outcomes:

- Apply Python quality tools (linters, type checkers, security scanners)
- Configure tools for your projects
- Integrate tools into development workflow

Quality Tools ?

Why Quality Tools?

Benefits

- 🤖 **Automate** bug detection
- 📏 **Enforce** consistency
- 📖 **Improve** readability
- 🔒 **Detect** security issues
- ⚡ **Accelerate** development
- 🧪 **Support** testing practices

Categories

1. **Linters/Formatters** - Style & formatting
2. **Type Checkers** - Static type analysis
3. **Security Scanners** - Vulnerability detection
4. **Complexity Analyzers** - Code complexity metrics
5. **Documentation Checkers** - Doc quality

💡 Tools should integrate in your workflow to automate quality checks

Static vs Dynamic Analysis

Static Analysis

Analyze code without running it
=> Fast !

How It Works

- Read and parse source code
- Apply rules and patterns
- No execution needed
- Fast feedback loop

What It Checks

- Code style violations
- Type errors
- Security vulnerabilities
- Potential bugs

Dynamic Analysis

Analyze code while running it
=> Slower but more in depth

How It Works

- Execute the program
- Monitor behavior at runtime
- Test execution paths
- Measure performance

What It Checks

- Logic errors
- Runtime failures
- Performance bottlenecks
- Memory leaks
- Integration issues

Category 1: Linters and Formatters

What are Linters?

Tools that analyze code for:

- **Style violations** (PEP 8)
- **Potential bugs**
- **Code smells**
- **Best practice violations**

Popular Python Linters

- **Ruff** ⭐
- Flake8 (classic)
- Pylint (comprehensive)

What are Formatters?

Tools that automatically fix formatting:

- **Consistent style**
- **Readable code**
- **No manual formatting**

Popular Python Formatters

- **Ruff** ⭐
- Black (opinionated)
- autopep8 (PEP 8 focused)

💡 Ruff combines both linting and formatting in one fast tool!

Ruff: Ultra-Fast Python Linter & Formatter

What is Ruff?

- Written in **Rust** (10-100x faster)
- Replaces **Flake8, Black, isort…**
- Supports **700+ rules**
- Auto-fixes many issues

Installation

```
pip install ruff / pixi add ruff
```

or

> IDE integration (e.g. VScode extension)

Basic Usage

```
# Check for issues
ruff check .

# Auto-fix issues
ruff check --fix .

# Format code
ruff format .
```

Example configuration

`pyproject.toml` / `ruff.toml` (auto-read when running ruff)

```
[tool.ruff]
# Exclude common directories
exclude = [
    ".pixi",
    "__pycache__",
]

# Line length
line-length = 99

# Enable specific rule sets
select = [
    "E",   # pycodestyle errors
    "W",   # pycodestyle warnings
    "F",   # pyflakes errors
    "I",   # isort
    "B",   # flake8-bugbear
    "C4",  # flake8-comprehensions
]

[tool.ruff.rules]
D = true  # enable pydocstyle-like docstring checks

# Per-file ignores: override rules for notebooks
per-file-ignores = {
    "notebooks/*.py" = ["E501"]  # ignore line length in notebooks
}
```

Source: ruff.rs

Exercise : Try Ruff yourself (5’)

1. Go back to `pkoffee`
2. Add ruff to your pixi environment
3. run `ruff check .`

- see if there are issues
- understand them
- try and fix them with `ruff check . --fix` or manually

4. run `ruff format .`
5. (Optional) Install the ruff extension for your IDE
    - mess with a file (e.g. remove spaces in a function variables definition)
    - save
    - watch instant formatting

> 💡 Mess with files to see ruff in action (the pkoffee source files were already linted)

Category 2: Type Checkers

What is Type Checking?

Static analysis of type hints:

```python
def add(a: int, b: int) -> int:
    return a + b

# Type checker catches this:
result = add("hello", 5)  # Error!
```

Benefits

- Catch bugs before runtime
- Better IDE support
- Self-documenting code
- Safer refactoring

ty: Static Type Checker

What is ty?

- **Fast, Rust-based type checker** for Python (much faster than `mypy`).
- **Rule-based configuration**: set severity per type of check (`error`, `warn`, `ignore`).
- **Inline suppressions**: `# ty: ignore[...]` for fine-grained control.
- **CLI and IDE integration** (e.g. VSCode extension)

Installation

```
pip install ty
```

```
pixi add ty
```

or

> IDE integration (e.g. VScode extension)

Basic Usage

Check a directory:

```
ty check src/
```

Example configuration

`pyproject.toml` / `ty.toml` (auto-read when running ruff)

```
[tool.ty]
include = ["your_package/", "src/"]
exclude = ["tests/", "data/", "notebooks/"]

[tool.ty.rules]
# treat missing imports as errors
possibly-missing-import = "error"
# unused ignore comments are warnings
unused-ignore-comment = "warn"
```

Source: https://docs.astral.sh/ty

Exercise : try ty

In `pkoffee`

1. Install ty
2. Add config to `pyproject.toml`
3. Run `ty check`

Category 3: Security Scanners

Tools can detect:

- **Hardcoded secrets** (passwords, API keys)
- **SQL injection** vulnerabilities
- **Insecure functions** (eval, pickle)
- **Weak cryptography**
- **Path traversal** issues

Why It Matters

- Protect sensitive data
- Prevent security breaches
- Meet compliance requirements
- Build trust

Bandit: Security Scanner

Installation

```
pip install bandit
```

```
pixi add bandit
```

or

> Use with IDE integration.

Usage

```
# Scan a directory
bandit -r your_package/

# Generate detailed report in JSON format
bandit -r your_package/ -f json -o report.json

# Ignore specific tests
bandit -r . -s B101,B601

# Pass a confif
bandit -r . -c bandit.toml
```

Example of issues

- B105: Hardcoded password
- B301: Use of pickle (unsafe)
- B614: Unsafe use of pytorch load

Complete list

Source: PyCQA/bandit

Dependabot: Automated Dependency Updates

What is Dependabot?

- **GitHub-native** tool (free for all repos)
- Automatically **monitors dependencies**
- Creates **pull requests** for updates
- Detects **security vulnerabilities**
- Supports multiple ecosystems (Python, npm, Docker, etc.)

⚠️ pixi.toml not supported yet :(

Key Features

- Security alerts for known CVEs
- Version updates (major, minor, patch)
- Automatic PR creation
- Configurable update schedule
- Grouping related updates

Configuration

`.github/dependabot.yml`

```yaml
version: 2
updates:
  # Python dependencies
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
      day: "monday"
    open-pull-requests-limit: 5
    labels:
      - "dependencies"
      - "python"
    reviewers:
      - "your-team"

  # GitHub Actions
  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "monthly"
```

💡 Exercise: Enable Dependabot in your pkoffee repository security settings for auto vulnerability fixes!

Additional Useful Tools

pip-audit

Check for known vulnerabilities in dependencies.

- Works locally or in CI.
- Fails builds if vulnerable packages are present.

```
pip install pip-audit
```

```
pixi add pip-audit
```

```
pip-audit
```

interrogate

Measure docstring coverage

```
pip install interrogate
```

```
pixi add interrogate
```

```
interrogate -v your_package/
```

hadolint: Dockerfile Linting

Linter for Dockerfiles - checks best practices, security, and efficiency.

See their repository for installation instructions (depends on your system).

Usage:

```
# Lint a Dockerfile
hadolint Dockerfile

# Ignore specific rules
hadolint --ignore DL3008 Dockerfile
```

Common checks:

- Base image pinning
- Layer optimization
- Security best practices
- COPY vs ADD usage

Pre-commit Hooks

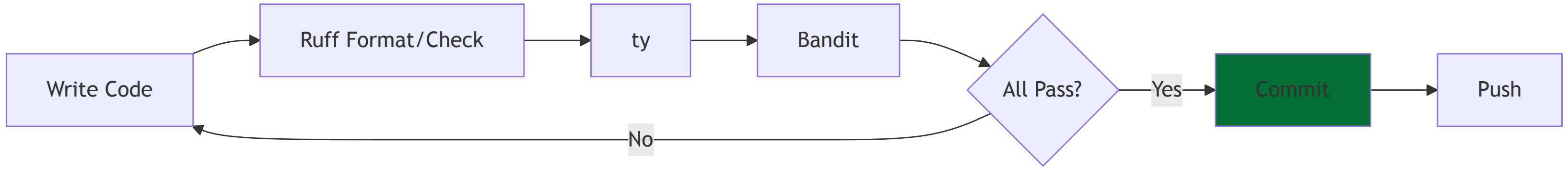
Run tools automatically before commits

```
pip install pre-commit
pre-commit install
```

Config example `.pre-commit-config.yaml`:

```
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v6.0.0
    hooks:
    -   id: trailing-whitespace
    -   id: check-added-large-files
    - id: actionlint
      name: Lint GitHub Actions workflow files
      description: Runs actionlint to lint GitHub Actions workflow files
      language: golang
      types: ["yaml"]
      files: ^\.github/workflows/
      entry: actionlint
      minimum_pre_commit_version: 3.0.0
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.1.0
    hooks:
      - id: ruff
      - id: ruff-format
  - repo: https://github.com/gitleaks/gitleaks
    rev: v8.24.2
    hooks:
      - id: gitleaks
```

Putting It All Together: Quality Workflow



Local Development

1. Write code
2. Format with `ruff format`
3. Lint with `ruff check --fix`
4. Type check with `ty`
5. Security scan with `bandit`
6. Commit if all pass

Automation Options

- **Pre-commit hooks** - Run before each commit
- **IDE integration** - Real-time feedback or fix
- **CI/CD** - Prevents from merging code not following repository standards or rules

💡 Start manually, then add automation as you get comfortable

Exercise

- add `ruff` check to your CI/CD
- add `interrogate` to your CI/CD

💡 Use GitHub actions marketplace

Exercise: Add badges to your README

⏱️ 15 minutes hands-on exercise

Tasks

1. **GitHub Actions Badge**:
   - Go to your repository **Actions** tab
   - Select your workflow
   - Click **...** -> **Create status badge**
   - Copy Markdown and add it to `README.md`
2. **Ruff / Quality Badges**:
   - Use Shields.io to create custom badges
   - Example: Linted with Ruff
3. **Interrogate (Opt)**:
   - Use the interrogate badge if you have it in your CI

Examples

GitHub Action status:

```
[![Deliver results](https://github.com/s3-school/pkoffee/actions/wo
```

Deliver results | failing

Ruff badge:

```
[![Ruff](https://img.shields.io/endpoint?url=https://raw.githubuser
```

Ruff

Custom badge:

```
![Quality](https://img.shields.io/badge/quality-A-brightgreen)
```

quality | A

Resources and Further Learning

Documentation

- Ruff Documentation
- mypy Documentation
- Bandit Documentation
- pre-commit

Tools & Guides

- Ruff Rules
- mypy Type Hints Cheat Sheet
- PEP 8 Style Guide
- Python Type Hints

EVERSE

- EVERSE Project
- RSQKit
- Quality Dimensions
- EVERSE TechRadar

Community

- EVERSE Network
- Research Software Engineers (RSE)
- Software Carpentry

Questions?

thomas.vuillaume@lapp.in2p3.fr

Thank you!