METASNAKE

Modern Python

- Ruff
- Precommit
- Walrus
- Match
- Dataclasses
- Async
- PyPi

Ruff: Fast Python Linter

- **What is Ruff?**
  - A fast, highly configurable linter and formatter for Python.
  - Built in Rust for speed, focusing on linting, formatting, and static analysis.
- **Key Features**:
  - Fastest Python linter due to Rust implementation.
  - Supports over 500 rules, covering best practices, PEP8, unused imports, and more.
  - Can be used alongside other tools like **flake8** and **black**.
- **Example Usage**:

```
uv add --dev ruff
uv run ruff check
```

Basic Linter Usage

- **Check all files in the current directory**:

```
uv run ruff check .
```

- **Check a specific file**:

```
uv run ruff check path/to/file.py
```

- **Example Output**:
  - Ruff will display violations, including line numbers, rule IDs, and descriptions of the issue.

Auto Fixing

- Ruff can automatically fix certain linting issues:

```
uv run ruff check . --fix
```

- Hint: Run tests before and after auto-fixing to ensure no functionality is broken.

```
uv run pytest
```

- Check diff using `git`:

```
git diff
```

Basic Formatter Usage

- Ruff can format code using the `format` option:

```
uv run ruff format
```

- Run your tests again!

Excluding Files/Directories

- **Exclude Specific Files/Directories**:
  - Use the `--exclude` flag to ignore specific paths:

```
uv run ruff check . --exclude tests/
```

Configuration

- Ruff can be configured using a `pyproject.toml` file for persistent project settings. You can also put these in `ruff.toml`. Example configuration:

```
[tool.ruff]
line-length = 60
```

- Running `uv run ruff format` will format the code according to the configuration.
- Running `uv run ruff check` will complain about long lines if we add rule `E501` to the configuration.

```
[tool.ruff.lint]
extend-select = ["E501"]
```

- Can use:
  - `select` to enable specific rules.
  - `ignore` to disable specific rules.
  - `extend-select` to add to the selected rules.
- `--select ALL` will run all rules.
- Prefer `select` over `extend-select` for clarity.
- Start with a minimal set of rules and add more as needed.

More Configuration

I like single quotes, so I can add a rule to enforce that:

```
[tool.ruff.format]
quote-style = "single"
indent-style = "space"
```

Rules

Ruff supports over 800 rules, which can be enabled or disabled based on your project's needs. Theses are reimplemented in Rust for optimal performance.

Use the rule code prefix to enable groups.

Some categories include:

- Pyflakes (F): Detects common issues like undefined variables.
- pycodestyle (E, W): Enforces PEP8 style guide.
- mccabe (C90): Detects complex code blocks. (turn on `C901`)
- isort (I): Enforces import sorting.
- pep8-naming (N): Enforces PEP8 naming conventions.
- pydocstyle (D): Checks docstrings. (turn on `D100`, `D101`, `D102`, `D103`, `D104`, `D105`, `D106`, `D107`)
- pyupgrade (UP): Upgrades syntax to newer versions. (on by default)
- flake8-2020 (YTT): Misuse of `sys.version` and `sys.version_info`
- flake8-annotations (ANN): Checks for missing type annotations.
- flake8-async (ASYNC): Checks for incorrect usage of async.
- flake8-bandit (S): Checks for common security issues.
- flake8-blind-except (BLE): Checks for broad `except` clauses.
- flake8-boolean-trap (FBT): Checks for common boolean traps.
- flake8-bugbear (B): Checks for common bugs and design problems.
- flake8-builtins (A): Checks for shadowing built-in names.
- flake8-commas (COM): Checks for missing or extra commas.
- flake8-copyright (CPY): Checks for missing copyright.
- flake8-comprehensions (C4): Checks for inefficient list comprehensions.
- flake8-datetimez (DTZ): Checks for incorrect usage of `datetime`.
- flake8-debugger (T10): Checks for debugger statements.
- flake8-django (DJ): Checks for common Django issues.
- flake8-errmsg (EM): No f-strings, `.format()`, in error messages.
- flake8-executable (EXE): Checks for executable scripts.
- … more flake-8
- eradicate (ERA): Checks for commented-out code.
- pandas-vet (PD): Checks for common pandas issues.
- pygrep-hooks (PGH): Checks for common issues.
- Pylint (PL): Checks for common issues.
- tryceratops (TRY): Checks for missing exception handling.
- flynt (FLY): Replace .join suggestion.
- Numpy (NPY): Checks for common NumPy issues.
- FastAPI (FAST): Checks for FastAPI-specific issues.
- Airflow (AIR): Checks for Apache Airflow-specific issues.
- Perflint (PERF): Checks for performance issues.
- refurb (FURB): Checks for refactoring opportunities.
- pydoclint (DOC): Checks for missing docstrings.
- Ruff specific rules (RUF): `RUF001`…

Running Specific Rules

Use the command line option `--select` to run specific rules:

```
uv run ruff check --select D
```

Checking Numpy Docstrings

Add this configuration to check for missing docstrings in NumPy-style:

```
[tool.ruff.lint]
extend-select = ["D"]

[tool.ruff.lint.pydocstyle]
convention = "numpy"
```

Ignoring Errors

Add `# noqa` to the end of a line to ignore a specific error:

```
@pytest.mark.matt
def test_matt(): # noqa
    assert 'matt' == 'matt'
```

You can also add specific rules to ignore. `# noqa: E501` will ignore the line length rule.

You can add `# ruff: noqa` to ignore all rules in a file or `# ruff: noqa: E501` to ignore a specific rule for a file.

Fixing noqa Issues

`RUF100` *What it does*

Checks for noqa directives that are no longer applicable.

Why is this bad?

A noqa directive that no longer matches any diagnostic violations is likely included by mistake, and should be removed to avoid confusion.

Use:

```
uv run ruff check --extend-select RUF100 --fix
```

To detect erroneous `noqa` comments and remove them.

Working with New Rules

To just focus on a new rule, you can tell Ruff to add `# noqa` to all lines that violate that rule. This can be useful to focus on fixing new issues without being distracted by old ones.

```
uv run ruff check --select F --add-noqa
```

What is Pre-Commit?

- **Pre-Commit** is a framework for managing and maintaining pre-commit hooks.
- Hooks are scripts that run automatically at key points in your Git workflow.
  - Example: Before committing code changes.
- **Why Use Pre-Commit?**
  - Helps catch common issues like linting errors or missing files **before** they are committed.
  - Enforces consistent code quality across all team members.

Note

Git also uses the term *pre-commit* for hooks that run before a commit is made. Pre-Commit is a (Python) tool that helps manage these hooks.

Setting Up Pre-Commit

1. **Install Pre-Commit**:

```
uv add --dev pre-commit
```

2. **Create a Configuration File** (`.pre-commit-config.yaml`):
   - The file contains a list of hooks to run.

```yaml
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v2.3.0
  hooks:
    - id: check-yaml
    - id: end-of-file-fixer
    - id: trailing-whitespace
- repo: https://github.com/astral-sh/ruff-pre-commit
# Repo Version
  rev: v0.6.9
  hooks:
    # Run the linter.
    - id: ruff
      types_or: [ python, pyi, jupyter ]
      args: [ --fix ]
    # Run the formatter.
    - id: ruff-format
      types_or: [ python, pyi, jupyter ]
```

3. **Install the Hooks**:
   - Run the following command to install the pre-commit hooks. It will create a `.git/hooks/pre-commit` file.

```
uv run pre-commit install
```

4. **Run the Hooks**:
   - Run the hooks on all files using:

```
uv run pre-commit run --all-files
```

```
- Can run individual hooks using:
```

```
uv run pre-commit run <hook-id> --all-files
```

Pre-Commit Hooks

The pre-commit tool includes a variety of pre-built hooks for common tasks, but there are also a bunch of other repos with hooks available.

https://pre-commit.com/hooks.html

More Notes

You should check in the `.pre-commit-config.yaml` file into your repository.

Every user who clones the repository will need to run `uv run pre-commit install` to set up the hooks.

Pre-Commit Hooks

There are pre-built hooks available for common tasks like:

- `check-yaml`: Validates YAML files.
- `end-of-file-fixer`: Ensures files end with a newline.
- `trailing-whitespace`: Removes trailing whitespace from files.

Using With Git

When you commit changes, the pre-commit hooks will run automatically. If any hook fails, the commit will be aborted.

Note that some of the hooks may modify the files. So subsequent commits may work.

You might want to run `git diff` to see the changes made by the hooks.

Hints

- Turn off hooks temporarily using `--no-verify` flag with `git commit`.
- Validate your configuration file using `uv run pre-commit validate-config .pre-commit-config.yaml`
- Can exclude files from running hooks using `.pre-commit-config.yaml`:

```yaml
hooks:
  - id: trailing-whitespace
    exclude: (tests|docs)/.*
```

Walrus Operator (:=)

The **Walrus Operator** (introduced in Python 3.8) allows you to assign a value to a variable as part of an expression.

This means you can both compute and assign values in a single, concise line.

Expression vs Statement

- **Expression**: A piece of code that produces a value, like `2 + 2`.
- **Statement**: A line of code that performs an action, like `name = "Alice"` or `def greet(): ...`.

A statement doesn't produce a value, so you can't use it within an expression. However, the walrus operator allows you to assign a value within an expression.

Common Use Cases

- **In Conditionals**: You can assign a value inside a conditional statement to avoid recomputation.

  Before:

  ```
  n = len(data)
  if n > 10:
      print(f"Data length is {n}")
  ```

  With walrus:

  ```
  if (n := len(data)) > 10:
      print(f"Data length is {n}")
  ```

- **In Loops**: The walrus operator is useful when you want to reuse a value that is computed within a loop condition.

  Before:

  ```
  line = file.readline()
  while line != '':
      print(line)
      line = file.readline()
  ```

  With walrus:

  ```
  while (line := file.readline()) != '':
      print(line)
  ```

- **In Regular Expressions**: You can use the walrus operator to avoid repeating the same expression multiple times.

  Before:

  ```
  match = re.search(r"\d+", text)
  if match:
      print(match.group())
  ```

  With walrus:

  ```
  if (match := re.search(r"\d+", text)):
      print(match.group())
  ```

In Pandas

You can use the walrus operator with a parameter in a function call. See `data` below:

```
import plotly.express as px
# sampling because it takes too long with 40,000 points
fig = px.scatter_3d(data:=pd.DataFrame(X_pca, columns=[f'PC{i}' for i in range(1, X_pca.shape[1]+1)]).assign(**auto,
        cluster=labels)
```

```
    .sample(2_000, random_state=42),
                x='PC1', y='PC2', z='PC3', color='cluster', hover_data=data.columns,
                color_continuous_scale='viridis')
fig.update_layout(
    width=800,
    height=600,
    title='3D PCA Scatter Plot'
)
fig.update_traces(marker=dict(size=3))

fig.show()
```

Another Example

```
    def fetch_many_wrapper(result, count=20_000):
        """
        In an effort to speed up queries, this wrapper
        fetches count objects at a time. Otherwise our
        implementation has sqlalchemy fetching 1 row
        at a time (~30% slower).
        """
        done = False
        while not done:
            items = result.fetchmany(count)
            done = len(items) == 0
            if not done:
                yield from items
```

Walrus operator (:=) version

```
    def fetch_many_wrapper(result, count=20_000):
        """
        In an effort to speed up queries, this wrapper
        fetches count objects at a time. Otherwise our
        implementation has sqlalchemy fetching 1 row
        at a time (~30% slower).
        """
        while len(items := result.fetchmany(count)):
            yield from items
```

The Match Statement in Python

- Introduced in Python 3.10
- Provides structural pattern matching
- Enhances readability and reduces boilerplate code

Basic Syntax

- Uses `match` keyword followed by an expression
- Contains one or more `case` clauses

```
match value:
    case pattern1:
```

```
        # code for pattern1
    case pattern2:
        # code for pattern2
    case _:
        # default case
```

Simple Value Matching

- Match against literal values

Before:

```
def describe_number(x):
    if x == 0:
        print("Zero")
    elif x == 1:
        print("One")
    else:
        print("Something else")
```

With match:

```
def describe_number(x):
    match x:
        case 0:
            print("Zero")
        case 1:
            print("One")
        case _:
            print("Something else")
```

```
>>> describe_number(0), describe_number(1), describe_number(2)
```

```
Zero
One
Something else

(None, None, None)
```

Matching Sequences

- Match against lists or tuples

Before:

```
def process_point(point):
    if len(point) == 2:
        x, y = point
        print(f"2D point: ({x}, {y})")
    elif len(point) == 3:
        x, y, z = point
        print(f"3D point: ({x}, {y}, {z})")
    else:
        print("Not a valid point")
```

With match:

```
def process_point(point):
    match point:
        case (x, y):
            print(f"2D point: ({x}, {y})")
        case (x, y, z):
            print(f"3D point: ({x}, {y}, {z})")
        case _:
            print("Not a valid point")
```

```
>>> process_point((1, 2)), process_point((1, 2, 3)), process_point((1, 2, 3, 4))
```

```
2D point: (1, 2)
3D point: (1, 2, 3)
Not a valid point

(None, None, None)
```

Matching with Guards

- Add conditions to case patterns

```
def classify_number(x):
    match x:
        case n if n < 0:
            print("Negative number")
        case n if n % 2 == 0:
            print("Even number")
        case n if n % 2 != 0:
            print("Odd number")
```

```
>>> classify_number(-1), classify_number(2), classify_number(3)
```

```
Negative number
Even number
Odd number

(None, None, None)
```

Matching Objects

- Match against object attributes

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def process_shape(shape):
    match shape:
        case Point(x=0, y=0):
            print("Point at origin")
        case Point(x=x, y=y):
            print(f"Point at ({x}, {y})")
        case _:
            print("Not a point")
```

```
>>> p = Point(0, 0)
>>> p2 = Point(1, 2)
>>> process_shape(p), process_shape(p2)
```

```
Point at origin
Point at (1, 2)

(None, None)
```

OR Patterns

- Match multiple patterns in a single case

```
def describe_type(value):
    match value:
        case str() | bytes():
            print("String-like object")
        case int() | float():
            print("Numeric type")
        case list() | tuple() | set():
            print("Sequence type")
        case _:
            print("Unknown type")
```

```
>>> describe_type("hello"), describe_type(1), describe_type([1, 2, 3])
```

```
String-like object
Numeric type
Sequence type

(None, None, None)
```

Capturing Matched Values

- Assign matched values to variables

```
def process_command(command):
    match command.split():
        case ["quit"]:
            print("Exiting program")
        case ["create", filename]:
            print(f"Creating file: {filename}")
        case ["delete", *files]:
            print(f"Deleting files: {files}")
        case _:
            print("Unknown command")
```

```
>>> process_command("quit"), process_command("create file.txt"), process_command("delete file1 file2 file3")
```

```
Exiting program
Creating file: file.txt
Deleting files: ['file1', 'file2', 'file3']

(None, None, None)
```

Example from sk-stepwise

Old code:

```python
def clean_int_params(self, params: dict[str, PARAM]) -> dict[str, PARAM]:
    int_vals = ['max_depth', 'reg_alpha']
    return {k: int(v) if k in int_vals else v for k, v in params.items()}
```

New code:

```python
def clean_int_params(self, params: dict[str, PARAM]) -> dict[str, PARAM]:
    match params:
        case {'max_depth': max_depth, **rest} if isinstance(max_depth, (float, str)):
            params['max_depth'] = int(max_depth)
        case {'reg_alpha': reg_alpha, **rest} if isinstance(reg_alpha, (float, str)):
            params['reg_alpha'] = int(reg_alpha)
    return params
```

Benefits of Match Statements

- More readable and concise code
- Reduces the need for multiple `if-elif` chains
- Powerful pattern matching capabilities
- Easier to maintain and extend

Considerations

- Only available in Python 3.10 and later
- May require a learning curve for developers new to pattern matching
- Not a replacement for all conditional logic, but a powerful tool when used appropriately

Dataclasses in Python

- Introduced in Python 3.7, the `dataclass` decorator simplifies class creation by automatically generating special methods like `__init__()`, `__repr__()`, `__eq__()`, etc.
- Provides a cleaner and more readable way to create data-centric classes, especially when dealing with many attributes.

Why Use Dataclasses?

- **Reduces Boilerplate Code**: You no longer need to write repetitive `__init__` and `__repr__` methods.
- **Built-in Methods**: Common methods like `__eq__` are automatically implemented.
- **Field Management**: Offers features like default values, default factories, and field metadata for easy management of attributes.

Basic Example

- The `dataclass` decorator automatically generates the `__init__` and `__repr__` methods for the `Point` class.

```python
from dataclasses import dataclass

@dataclass
class Point:
```

```
    x: float
    y: float

>>> p1 = Point(3.0, 4.0)
>>> print(p1)

Point(x=3.0, y=4.0)
```

Features of Dataclasses

Default Values and Factories

- You can define default values for attributes, or use `field()` for more complex default behavior.

```
from dataclasses import dataclass, field

@dataclass
class Car:
    make: str
    model: str
    year: int = 2023
    options: list[str] = field(default_factory=list)
```

- `year` has a default value of `2023`.
- `options` creates a list in the constructor if not provided (rather than using a mutable default argument).

Immutable Dataclasses

- Use `frozen=True` to make dataclass instances immutable (like named tuples).

```
@dataclass(frozen=True)
class Coordinates:
    latitude: float
    longitude: float
```

- Attempting to modify `latitude` or `longitude` will raise an exception.

`__post_init__` Method

- The `__post_init__` method is a special method in dataclasses that runs automatically after the `__init__` method.
- It is useful for performing additional initialization or validation after the dataclass has been created.
- Example:

```
from dataclasses import dataclass

@dataclass
class Rectangle:
    width: float
    height: float
    area: float = 0.0

    def __post_init__(self):
        if self.width <= 0 or self.height <= 0:
            raise ValueError("Width and height must be positive values.")
        self.area = self.width * self.height
```

```
rect = Rectangle(3.0, 4.0)
print(rect.area)  # Output: 12.0
```

- In this example, `__post_init__` is used to calculate the `area` after the object is initialized and to validate that both `width` and `height` are positive.
- `__post_init__` is ideal for situations where you need to derive values, perform checks, or modify attributes based on the initial input values.

Conclusion

- Dataclasses offer a powerful, concise way to create classes focused on storing data.
- They are especially useful for applications involving configuration, serialization, or domain models with little logic.
- They improve readability and reduce boilerplate code, making your codebase easier to manage and maintain.

Example from sk-stepwise

Old code:

```
class StepwiseHyperoptOptimizer(BaseEstimator, MetaEstimatorMixin):
    def __init__(
        self,
        model: _Fitable,
        param_space_sequence: list[dict[str, PARAM | SymbolTable]],
        max_evals_per_step: int = 100,
        cv: int = 5,
        scoring: str
        | Callable[[ArrayLike, ArrayLike], float] = 'neg_mean_squared_error',
        random_state: int = 42,
    ) -> None:
        self.model = model

        self.param_space_sequence = param_space_sequence
        self.max_evals_per_step = max_evals_per_step
        self.cv = cv
        self.scoring = scoring
        self.random_state = random_state
        self.best_params_: dict[str, PARAM] = {}
        self.best_score_ = None
```

New code:

```
from dataclasses import dataclass, field

@dataclass
class StepwiseHyperoptOptimizer(BaseEstimator, MetaEstimatorMixin):
    model: _Fitable
    param_space_sequence: list[dict[str, PARAM | SymbolTable]]
    max_evals_per_step: int = 100
    cv: int = 5
    scoring: str | Callable[[ArrayLike, ArrayLike], float] = 'neg_mean_squared_error'
    random_state: int = 42
    best_params_: dict[str, PARAM] = field(default_factory=dict)
    best_score_: float = None
```

Asynchronous Programming in Python

- Asynchronous programming allows a program to handle multiple tasks at once, without waiting for each task to complete before moving to the next.
- In Python, `asyncio` is the core library used for writing asynchronous code.
- Useful for I/O-bound tasks like making HTTP requests, reading/writing files, or handling multiple network connections.

Synchronous vs. Asynchronous

- **Synchronous Code**: Tasks are executed one at a time, blocking further execution until each completes.
- **Asynchronous Code**: Tasks are executed concurrently, allowing the program to move on without waiting for each task to complete.

Basic Async Example

- Using `async` and `await` keywords to define and call asynchronous functions.

```
import asyncio

async def greet():
    print("Hello!")
    await asyncio.sleep(1)
    print("World!")

asyncio.run(greet())
```

- `async def greet()` defines an asynchronous function.
- `await asyncio.sleep(1)` pauses the function, allowing other tasks to run concurrently.
- `asyncio.run(greet())` runs the event loop to execute the coroutine.

Coroutines and Event Loop

- **Coroutines**: Functions defined with `async def` are called coroutines. They are the building blocks of asynchronous programming.
- **Event Loop**: The event loop manages and executes coroutines. It coordinates tasks and resumes them when they're ready to continue.

Running Multiple Tasks Concurrently

- Use `asyncio.gather()` to run multiple tasks concurrently.

```
import asyncio

async def task_one():
    await asyncio.sleep(1)
    print("Task One Complete")

async def task_two():
    await asyncio.sleep(2)
    print("Task Two Complete")

async def main():
    await asyncio.gather(task_one(), task_two())
```

```
asyncio.run(main())
```

- `asyncio.gather()` runs `task_one` and `task_two` concurrently, resulting in faster overall execution.

Using `await` to Pause Execution

- The `await` keyword pauses the coroutine until the awaited task completes.
- It is important to use `await` only on functions that are declared as `async` or return awaitable objects.

Asynchronous I/O Operations

- Asynchronous programming is particularly effective for I/O-bound tasks.
- Example of making HTTP requests asynchronously:

```
import asyncio
import aiohttp

async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    url = "https://example.com"
    data = await fetch_data(url)
    print(data)

asyncio.run(main())
```

- `aiohttp` is an asynchronous library for making HTTP requests.
- `async with` ensures that resources are cleaned up properly after use.

Common Use Cases for Async

- **Web Scraping**: Making multiple HTTP requests concurrently.
- **Database Queries**: Performing multiple non-blocking queries.
- **Real-time Applications**: Handling multiple WebSocket connections or chat clients concurrently.

Benefits and Considerations

- **Benefits**:
  - Improves performance in I/O-bound tasks.
  - Provides a more responsive experience in applications like web servers.
- **Considerations**:
  - Asynchronous code can be harder to read and debug compared to synchronous code.
  - Not suitable for CPU-bound tasks—use multithreading (Python 3.13) or multiprocessing for CPU-intensive workloads.

README.md

A README should include the following sections:

- **Title**: The name of the project.
- **Description**: A brief overview of the project's purpose.
- **Installation**: Instructions on how to install the project.
- **Usage**: Examples of how to use the project.
- **Contributing**: Guidelines for contributing to the project.
- **License**: Information about the project's license.

AI Help

Use AI to write boilerplate code for your README.

Building and Publishing Python Packages on PyPI

Remove section from `pyproject.toml`:

```
build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

Run `uv build` to build a source distribution package and a wheel.

Run `uv build --sdist` to build a source distribution package.

Validate that package can be installed and imported:

```
uv run --with sk-stepwise -- python -c "import sk_stepwise"
```

Creating PyPI Account

- register for an account

https://test.pypi.org/account/register/

- create tokens

https://test.pypi.org/manage/account/#api-tokens

jump through hoops for two factor authentication, get token. Through in /.pypirc

```
[testpypi]
  username = __token__
  password = pypi-REMOVED
```

Repeat for pypi

Uploading to PyPI

upload to test pypi

```
uv add --dev twine
uv run twine upload --repository testpypi dist/*
```

Updating the Package

- Update the version number in `pyproject.toml`
- Run `uv build` to create a new distribution package
- Run `uv run twine upload dist/*` to upload the new version to PyPI