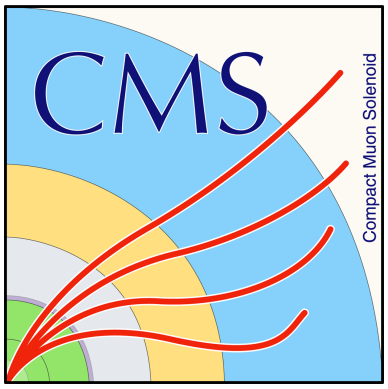

Implementing Python code quality checks in the CMSSW Continuous Integration infrastructure

September 3, 2024

Benedikt Tor Kristinsson
Aarhus University Department of Physics and Astronomy

Supervised by: Andrea Valenzuela Ramirez
Fellow at Core Software.
CMSSW
CERN

Acknowledgements

I would like to acknowledge and thank my supervisor, Andrea Valenzuela Ramirez, for her patience and guidance throughout this project. Her willingness to help me understand how to overcome obstacles and her steady support have been invaluable. She was a guiding hand at the start of the project that i could follow to reach the project goals. Additionally, I would like to recognize Malik Shahzad Muzaffar for his support during the project.

Contents

1 Introduction

CMSSW is a sophisticated software framework with an extensive codebase exceeding 5.5 million lines. The majority of this code is written in C++ (approximately 61%), with a significant portion in Python (about 28%), and Python's usage has been increasing over recent years[1]. CMSSW maintainers have successfully established a robust method for ensuring C++ code adheres to best practices through Pull Request (PR) testing. Building on this success, the current project focuses on enhancing the Continuous Integration (CI) checks for the Python codebase within CMSSW.
The project involves evaluating the existing Python code formatting and linting tools, such as Black[2] Pylint[3], Flake8[4], and Ruff[5]. The objective is to integrate the most suitable tool into the existing infrastructure to automatically detect errors in PRs, and suggest fixes. This initiative aims to enhance the quality and consistency of Python code.

This project presents an opportunity to gain hands-on experience with CI and Continuous Delivery while expanding knowledge in Python, bash, and GitHub.

2 Summer Student Activities

This section outlines the workflow and steps involved in the development of the Code Quality Check (CQC) software product. The project is divided into two stages to ensure a gradual and systematic approach.

By structuring the project into these stages, the development process aims to be systematic and adaptive, addressing challenges incrementally and ensuring a robust final product.

2.1 First Stage

Stage One is primarily focused on exploring and evaluating various code quality tools, such as Ruff, Black, and others, to determine which one best aligns with the CMSSW requirements. During this stage, the emphasis is on experimenting with these tools, understanding their functionalities, and assessing their suitability for integration into the CQC system. The primary goal is to identify the most appropriate tool for performing code quality checks rather than developing a complete software solution at this point.

During the first stage of the project, a considerable amount of time was dedicated to familiarizing with the CMSSW infrastructure and the tools. This period overlapped with lectures and activities for summer students at CERN, which added to the complexity of managing time effectively.

The initial phase of the workflow focused on understanding the CMSSW environment and how it integrates with CI practices. This involved becoming acquainted with the intricacies of the CMSSW framework, which included learning about both the `cms-sw/cmsbot`[6] and `cms-sw/cmssw`[7] repositories— the latter being the primary focus of the project.

Jenkins[8] is used as the CI server utilized within the CMSSW framework. Gaining proficiency with Jenkins involved understanding how it supports the CI pipeline by automating builds and tests. Another key aspect of this stage was studying the interaction between GitHub and Jenkins. We learned how GitHub webhooks are used to trigger Jenkins jobs and how these jobs are configured to respond to various GitHub events. This integration is vital for maintaining a smooth CI process and ensuring that CQCs and automated builds are effectively managed.

2.2 Second Stage

Stage Two builds upon the integration work completed in Stage One. After selecting the most suitable code quality tool and incorporating it into the existing infrastructure—specifically, the existing code-checks workflow that primarily focused on C++—the focus shifted to refining and enhancing the system. This stage involved testing, resolving any issues, and improving the functionality based on the feedback received.

During this stage was the mid-project presentation at the Core Software meeting on July 30, 2024[9]. During this presentation, the findings and progress made so far were shared, and feedback was gathered from members of the software group and other stakeholders. Their input was invaluable in guiding the further development of the CQC system, including suggestions for additional features and improvements for customization. The final adjustments to the project were discussed upon in the subsequent Core Software meeting[10].

3 Documentation

This section outlines the key resources and configurations related to the CQC project. It covers tool setups, integration steps, and the CI workflow documentation, ensuring clarity and accessibility for future use.

3.1 Selecting Tools for CQC Software

The selection of tools for Python CQC project involved evaluating various options for formatting and linting. The following tools have been considered: Black, Flake8, Pylint, and Ruff. Each tool has its own strengths and weaknesses, and their suitability for the project will be discussed below.

3.1.1 Understanding Patches and Diffs

When working with code, it's common to make modifications or improvements to existing files. These changes are often represented as **patches**. A patch is essentially a set of differences between the original version of a file and the modified version. To clearly see what has changed, developers use a **diff**, such as `git diff`, which highlights the specific lines of code that have been added, removed, or altered. The diff is essential for reviewing code changes, this feature can also be found in some Linting and Formatting tools, which is an wanted quality for this project.

3.1.2 Tool Evaluation

The evaluation of tools was a critical part of the process, involving a more detailed and dynamic discussion with my supervisor, where specific properties of each tool were closely examined.

To clarify the roles of the tools considered, it's important to differentiate between linters and formatters:

Linters are tools that analyze code to identify programming errors, potential bugs, stylistic issues, and deviations from coding standards. They help maintain code quality by enforcing best practices.

Formatters automatically adjust the layout of code, ensuring consistent style and formatting according to a predefined guide (e.g., PEP 8).

Some tools provide both linting and formatting capabilities, offering a comprehensive solution for maintaining code quality.

During my evaluation, I classified the tools as follows:

3.1.3 Linters:

3.1.3.1 Flake8 focuses on linting, checking for style violations, potential bugs, and code complexity, based on PEP 8.

- **Purpose**: Linter that checks code for compliance with coding standards.
- **Development Activity**: Actively maintained with contributions from the open-source community.
- **Installation Steps**:

```
pip install flake8
```

- **Usage**:

```
flake8 path/to/file.py
```

- **Reporting Diff**: Not available; provides linting results without direct formatting capabilities.
- **Configuration**: Can be customized via `.flake8` file. Example:

```
[flake8]
ignore = E251, W503
```

- **Pros**: Provides detailed linting information, customizable.
- **Cons**: Does not format code, only provides linting warnings.

3.1.3.2 Pylint is a robust linter that provides detailed feedback on code quality, enforcing coding standards and suggesting refactoring opportunities.

- **Purpose**: Static code analyzer that checks code quality and compliance with standards.
- **Development Activity**: Actively maintained with a large user base.
- **Installation Steps**:

```
pip install pylint
```

- **Usage**:

```
pylint path/to/file.py
```

- **Reporting Diff**: Not available; focuses on linting and code analysis.
- **Configuration**: Customizable via '.pylintrc' file. Example:

```
[MASTER]
disable = C0114, C0116
```

- **Pros**: Comprehensive analysis, highly configurable.
- **Cons**: Can be verbose, no direct formatting capabilities.

3.1.4 Formatters:

3.1.4.1 Black is dedicated to formatting, automatically reformatting Python code to ensure consistency according to PEP 8.

- **Purpose**: Code formatter that enforces PEP 8 standards.
- **Development Activity**: Actively maintained by the Python Software Foundation with regular updates.

- **Installation Steps**:

```
pip install black
```

- **Usage**:

```
black path/to/file.py
```

- **Reporting Diff**:

```
black path/to/file.py --diff
```

- **Configuration**: Limited configuration options via ‘pyproject.toml‘. Example:

```
[tool.black]
line-length = 88
skip-string-normalization = true
target-version = ['py38']
```

- **Pros**: Consistent formatting, widely adopted.
- **Cons**: Limited flexibility in formatting options.

3.1.5 Both:

3.1.5.1 Ruff serves as both a linter and a formatter, combining the functionalities of enforcing coding standards and formatting code consistently. The final tool was selected based on its ability to meet the specific needs of the CMSSW project, effectively balancing both linting and formatting requirements[11].

- **Purpose**: Linter and formatter that combines functionality of multiple tools (e.g., Black, Flake8).
- **Development Activity**: Actively developed with significant contributions, used by major projects.
- **Installation Steps**:

```
pip install ruff
```

- **Usage**:

```
ruff check path/to/file.py
ruff format path/to/file.py
```

- **Reporting Diff**:

```
ruff check --diff path/to/file.py
```

- **Configuration**: Configurable via ‘pyproject.toml‘ or ‘ruff.toml‘. Example:

```
[tool.ruff]
ignore = ["E501"]
```

- **Pros**: Combines linting and formatting, very fast, configurable.
- **Cons**: Newer tool faster then black. it has limited configuration options as Black but includes a few more. It also has linting and formatting on the same tool.

3.1.6 Conclusion on Tool Selection

After evaluating the tools, Ruff is chosen as the primary tool for the CQC project due to its ability to perform both linting and formatting efficiently see. Ruff's performance benefits, combined with its flexibility and speed, make it a suitable choice for handling the large code-base of CMSSW, see 1.

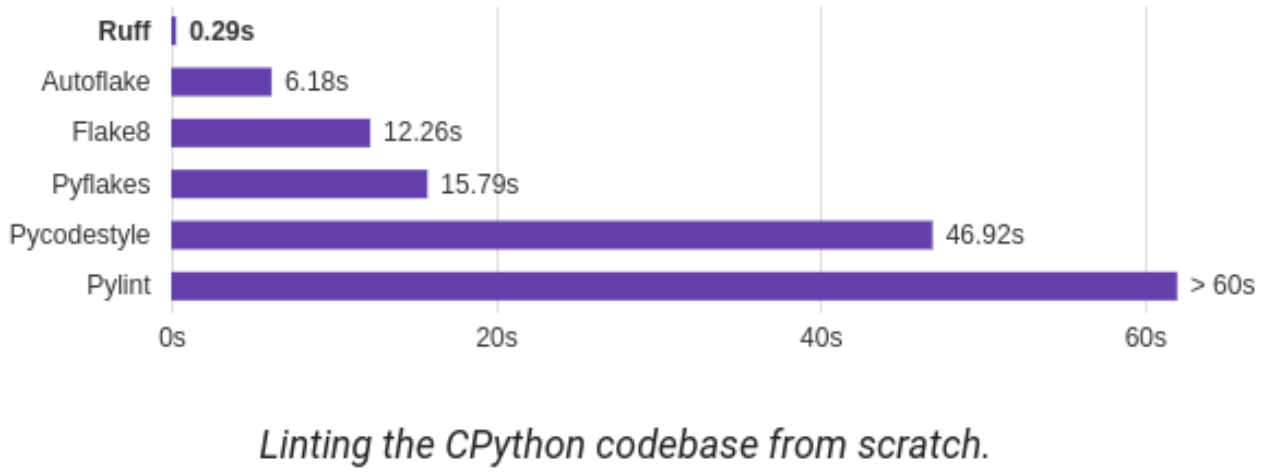

Figure 1: This figure is from the Ruff homepage [5] and it displays how much faster Ruff is then some other linters at linting the CPython codebase from scratch.

When we ran Ruff on the `cmssw/HLTrigger/Configuration/python/` it took 448.7s for linting and formatting whereas for black it took 593.812s, only for formatting.

3.2 Usage of Ruff for CQC

This subsection focuses on the integration of Ruff with the CQC project, providing information on setup and usage.

3.2.1 Configuration Guidelines

Ruff can be configured using a '.toml' file, which allows you to specify the same settings and rules for code linting and formatting as for Black with a few more options. This configuration file is typically named 'pyproject.toml' and is placed at the root of your project directory. For more detailed information on configuring Ruff, refer to the Ruff documentation [12]. The configuration specifics were also demonstrated in the Ruff-configuration section (3.1.5.1).

Here is an example of how an project directory might be structured, including the configuration file:

```
your_project/
├── pyproject.toml              % Configuration file for Ruff and other tools
├── your_module/
│   ├── __init__.py             % Module initialization
│   ├── your_code.py            % Your main code files
├── tests/
│   ├── test_your_code.py       % Unit tests for your code
└── ...                         % Other files and directories
```

In this example:

- `pyproject.toml` contains the configuration settings for Ruff. We can specify various options such as which linting rules to enforce, formatting styles, and other settings.
- `your_module/` is where our Python code is located. It includes an `__init__.py` file to mark the directory as a package and other Python scripts.
- `tests/` holds the test files. It is good practice to keep the tests separate from the main code [13].

4 Script Documentation

4.1 CI Workflow Documentation - run-pr-code-checks

4.1.1 Purpose

`run-pr-code-checks` is a bash script that's part of an automated CI process for a project using Jenkins. It's used to retrieve information about a specific PR from a GitHub repository and checks out the code. See appendix B.

- **Initialization:**
  - Sets up environment variables and paths based on input parameters such as PR number, repository, and build details.
- **Pull Request Handling:**
  - Retrieves the latest commit status for the PR from GitHub.
  - Updates the PR status to "pending" to indicate that checks are in progress.
- **Code Checkout and Build:**
  - Checks out the PR code and applies any specified tool configurations.
  - Merges the PR code and performs a build to check for errors.
  - Generates logs and reports if issues are detected.
- **Code Checks and Formatting:**
  - Runs code checks and formatting tasks using tools like `scram` and Python scripts.
  - Handles errors and applies fixes if needed.
- **Reporting and Notifications:**
  - Uploads build logs, check results, and code fixes to a server.
  - Posts comments on the GitHub PR with links to the logs and instructions for applying fixes.

4.1.2 Closed Loop Transparency

The closed loop implemented in `run-pr-code-checks` refers to the continuous feedback and adjustment mechanism in the CI workflow. This closed loop is designed to automatically update and refine the status and results of the PR process. Importantly, this feedback mechanism is transparent to the tool itself.

So in practice, this means that the tool operates independently of the feedback loop internal details. The script functions according to its defined tasks, such as checking out code, building, and running tests, without needing to be managed or be aware of the ongoing adjustments made by the feedback loop.

This transparency allows the tool to focus on its core functionality while the closed-loop system ensures that the overall process remains dynamic and responsive to changes.

4.2 run-python-format.py

4.2.1 Purpose

`run-python-format.py` automates the process of formatting and linting Python files by reading a list of files from an input file, constructing their absolute paths using a CMSSW base directory, and running `PFA.py` on them. See appendix C.

Suggestion: This script will be eventually integrated as part of SCRAM (https://github.com/cms-sw/SCRAM), the current management tool in CMSSW

4.2.2 Script Overview

- **Imports**:
  - `os`: For file path operations and executing system commands.
  - `argparse`: For parsing command-line arguments.
- **Main Functionality**:
  - `-inputfile`: Path to a file containing a list of files to process.
  - `-cmsswbase`: The path to the CMSSW base directory.
  - The script:
    1. Parses command-line arguments to get the paths for the input file and the CMSSW base directory.
    2. Checks if the input file exists; exits with an error message if it doesn't.
    3. Reads the list of file paths from the input file, stripping any whitespace.
    4. Constructs full file paths by joining each file path with the CMSSW base directory.
    5. Opens a file named `python-linting.txt` to record linting results.
    6. If the list of files is empty, writes a message to `python-linting.txt` indicating no files to check and exits.
    7. For each file, checks if the file exists, and if so, runs `ruff check` for linting, appending the results to `python-linting.txt`.
    8. Tracks whether all files passed the linting checks and records the result in `python-linting.txt`.
    9. If all files pass linting, runs `PFA.py` with the list of files as arguments.

4.2.3 Example Usage

```
python run-python-format.py --inputfile path/to/inputfile.txt --cmsswbase /path/to/cmssw
```

4.3 PFA.py

4.3.1 Purpose

`PFA.py` is a Python script designed to perform code quality checks on Python files by formatting them using the `ruff` tool. See appendix D.

4.3.2 Script Overview

- **Imports**:
  - `os`: For file path manipulation and executing system commands.
  - `argparse`: For handling command-line arguments.
- **Functions**:
  - `find_file_path(file)`: Returns the absolute path of the specified file.
  - `find_python_files(directory)`: Recursively searches for Python files in the specified directory.
  - `CodeQualityChecks(python_files)`: Formats the provided list of Python files using `ruff`.
- **Main Functionality**:
  - `paths`: A list of directories or file paths to process.
  - `-cmsswbase`: The path to the CMSSW base directory.
  - The script checks whether each path is a directory or file, gathers Python files, and formats these files using `ruff`.

4.3.3 Example Usage

```
python PFA.py path/to/dir_or_file1
```

5 Results

5.1 Product

The structure of the software can be seen in B, C, and D respectively in that order. The QCQ effectively formats, and gives linting suggestions as well as displaying it on the PR, effectively implementing the Python code quality checks in the CMSSW CI infrastructure. An example on how it can look can be seen on 2, where it is displayed how the interface will look like once a PR is created.

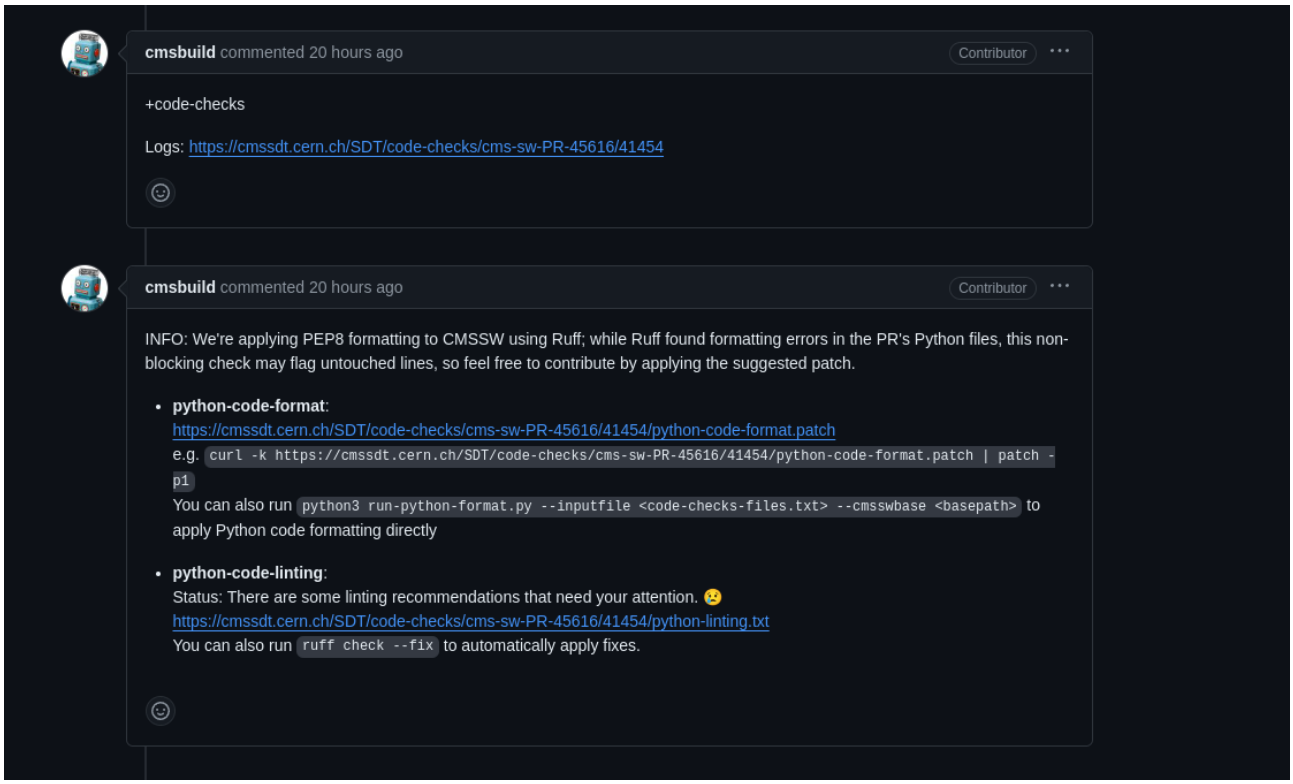

Figure 2: This figure shows how the final product will display CQC on the github PR[14]

5.2 Future steps

Given the time constraints, we were unable to implement all the suggested changes during the initial development phase. Moving forward, future add-ons could be to incorporate the feedback gathered from the meetings into future versions of the CQC.

6 Appendix

A Tool table

Table 1: Comparison of Python Code Quality Tools

Feature	Ruff	Black	Flake8	Pylint
Primary Function	Linter, formatter, and code quality checker	Code formatter	Linter	Linter and code quality checker
Language Support	Python	Python	Python	Python
Performance	High speed, particularly in large codebases	Fast	Moderate	Slower due to more comprehensive checks
Compliance with PEP8	Yes	Yes	Yes	Yes
Customizability	High, with many configurable rules	Low, with a focus on enforcing a consistent style	High, with plugins available	High, with many options for configuring checks
Error Detection	Detects a wide range of errors, including logic and style issues	Focuses on formatting only	Detects common style and logic errors	Comprehensive error detection, including coding standards and potential bugs
Integration	Easy to integrate into CI/CD pipelines	Easy to integrate	Easy to integrate, widely used	Can be integrated, but may require more configuration
Popularity	Increasingly popular, especially for large codebases	Very popular	Very popular	Popular, but considered more thorough and strict

B run-pr-code-checks

```
#!/bin/bash -ex
CMS_BOT_DIR=$(dirname $0)
case $CMS_BOT_DIR in /*) ;; *) CMS_BOT_DIR=$(pwd)/${CMS_BOT_DIR} ;; esac
export JENKINS_PREFIX_STR=$(echo "${JENKINS_URL}" | sed 's|jenkins/*$||;s|.*/||')
PULL_REQUEST=$1
export USER_CODE_CHECKS=$2
BUILD_NUMBER=$3
DRY_RUN=$4
REPOSITORY=$5
CODE_FORMAT=$6
CODE_CHECKS=$7
CMSSW_TOOL_CONF=$8
APPLY_PATCH=$9
if [ "${APPLY_PATCH}" != "true" ] ; then APPLY_PATCH=false; fi
MULTIPLE_FILES_CHANGES=true
if [ "${CONTEXT_PREFIX}" = "" ] ; then CONTEXT_PREFIX="cms"; fi
if [ "$DRY_RUN" != "true" ] ; then DRY_RUN=false ; fi

source ${CMS_BOT_DIR}/common/github_reports.sh
PR_COMMIT="$(wget -q -O- https://api.github.com/repos/cms-sw/cmssw/pulls/${PULL_REQUEST} | grep 'api.github.com/repos/cms-sw/cmssw/statuses/' | grep '"href"' | tail -1 | sed 's|.*/||;s|".*||')"
$DRY_RUN || mark_commit_status_pr -r "cms-sw/cmssw" -c "${PR_COMMIT}" -C "${CONTEXT_PREFIX}/code-checks" -s "pending" -u "${BUILD_URL}" -d "Running"

#disable script build rules to avoid copying newly added files in PATH
sed -i -e '/scripts+scripts/d' $CMSSW_BASE/config/BuildFile.xml

if [ "${CMSSW_TOOL_CONF}" != "" ] ; then
  mv $CMSSW_BASE/config/toolbox/${SCRAM_ARCH}/tools/selected $CMSSW_BASE/old-tools
  cp -r ${CMSSW_TOOL_CONF}/tools/selected $CMSSW_BASE/config/toolbox/${SCRAM_ARCH}/tools/selected
  if [ -e $CMSSW_BASE/old-tools/cmssw.xml ] ; then
    cp $CMSSW_BASE/old-tools/cmssw.xml $CMSSW_BASE/config/toolbox/${SCRAM_ARCH}/tools/selected/
  fi
  scram setup      > /dev/null 2>&1
  rm -rf $CMSSW_BASE/old-tools $CMSSW_BASE/config/SCRAM $CMSSW_BASE/external
  cp -r $(scram tool tag coral CORAL_BASE)/config/SCRAM $CMSSW_BASE/config/SCRAM
  scram b -r echo_CXX
  eval `scram run -sh` > /dev/null 2>&1
fi

if [ "X${REPOSITORY}" = "X" ] ; then REPOSITORY="cms-sw/cmssw" ;fi
REPO_USER=$(echo ${REPOSITORY} | sed 's|/.*||')
UP_URL="https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}"
if [ "$BUILD_NUMBER" = "" ] ; then BUILD_NUMBER=$(date +%s); fi
NUM_PROC=$(nproc)
if [ $NUM_PROC = "0" ] ; then NUM_PROC=1; fi
cd $CMSSW_BASE
UP_DIR="${CMSSW_BASE}/upload"
rm -rf ${UP_DIR}
mkdir ${UP_DIR}

if python3 -c 'import yaml' ; then
  sed -i -e 's|/usr/bin/env python$|/usr/bin/env python3|' config/SCRAM/fix-code-checks-yaml.py
fi
```

```
touch ${UP_DIR}/all-changed-files.txt
git cms-init --upstream-only
source $CMS_BOT_DIR/jenkins-artifacts
pushd $CMSSW_BASE/src
  if ! git cms-merge-topic -u ${REPO_USER}:${PULL_REQUEST} >${UP_DIR}/cms-checkout-topic.log 2>&1 ; then
    echo '--code-checks' > ${UP_DIR}/code-checks.md
    echo -e "\nLogs: $UP_URL" >> ${UP_DIR}/code-checks.md
    echo -e '\nERROR: Unable to merge PR.' >> ${UP_DIR}/code-checks.md
    echo -e "\nSee log ${UP_URL}/cms-checkout-topic.log" >> ${UP_DIR}/code-checks.md
    if ! $DRY_RUN ; then
      send_jenkins_artifacts ${UP_DIR}/ pr-code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}
      ${CMS_BOT_DIR}/comment-gh-pr.py -r ${REPOSITORY} -p $PULL_REQUEST -R ${UP_DIR}/code-checks.md
    fi
    exit 0
  fi
  ERR=false
  for f in $(curl -s -L https://patch-diff.githubusercontent.com/raw/${REPOSITORY}/pull/${PULL_REQUEST}.patch | grep '^diff --git ' | grep '^diff --git ' | sed 's|.* a/||;s|  +b/| |' | tr ' ' '\n' | sort | uniq) ; do
    [ ! -e $f ] || echo "$f" >> ${UP_DIR}/all-changed-files.txt
  done
  grep -v '^[^/]*/[^/]*/test/' ${UP_DIR}/all-changed-files.txt > ${UP_DIR}/code-checks-files.txt          || true
  grep -v '^[^/]*/[^/]*/data/' ${UP_DIR}/code-checks-files.txt > ${UP_DIR}/filename-code-checks-files.txt || true
  $CMS_BOT_DIR/cms-filename-checks.py ${UP_DIR}/filename-code-checks-files.txt $CMSSW_RELEASE_BASE/src > ${UP_DIR}/invalid-filenames.txt || true
  echo "Changed files:"
  cat ${UP_DIR}/code-checks-files.txt
  echo ""
  scram build -r echo_CXX > ${UP_DIR}/code-checks.log 2>&1 || ERR=true
  if $ERR ; then
    echo '--code-checks' > ${UP_DIR}/code-checks.md
    echo -e "\nLogs: $UP_URL" >> ${UP_DIR}/code-checks.md
    echo -e '\nERROR: Build errors found during clang-tidy run.' >> ${UP_DIR}/code-checks.md
    echo '```' >> ${UP_DIR}/code-checks.md
    grep 'SCRAM error: ' ${UP_DIR}/code-checks.log >> ${UP_DIR}/code-checks.md || true
    grep 'Parse Error ' -A 1 ${UP_DIR}/code-checks.log >> ${UP_DIR}/code-checks.md || true
    echo '```' >> ${UP_DIR}/code-checks.md
    if ! $DRY_RUN ; then
      send_jenkins_artifacts ${UP_DIR}/ pr-code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}
      ${CMS_BOT_DIR}/comment-gh-pr.py -r ${REPOSITORY} -p $PULL_REQUEST -R ${UP_DIR}/code-checks.md
    fi
    exit 0
  fi
  $CMS_BOT_DIR/pr-checks/check-pr-files.py -d -r ${REPO_USER}/cmssw ${PULL_REQUEST} > ${UP_DIR}/invalid_files.txt || true
  touch ${UP_DIR}/duplicate-data.txt
  if [ -d ${CMSSW_RELEASE_BASE}/external/${SCRAM_ARCH}/data ] ; then
    for f in $(cat ${UP_DIR}/all-changed-files.txt) ; do
      if [ -f ${CMSSW_RELEASE_BASE}/external/${SCRAM_ARCH}/data/$f -a -f ${CMSSW_BASE}/src/$f ] ; then
        echo $f >> ${UP_DIR}/duplicate-data.txt
      fi
    done
  fi
popd
```

```
#If we have any non-tests changed files
COMMIT_CHG=0
touch ${UP_DIR}/code-checks.patch
if $CODE_CHECKS ; then
  if [ -s ${UP_DIR}/code-checks-files.txt ] ; then
    ERR=false
    scram build -k -j $NUM_PROC code-checks USER_CODE_CHECKS_FILE="${UP_DIR}/code-checks-files.txt" > ${UP_DIR}/code-checks.log 2>&1 || ERR=true

    if $ERR ; then
      echo '-code-checks' > ${UP_DIR}/code-checks.md
      echo -e "\nLogs: $UP_URL" >> ${UP_DIR}/code-checks.md
      echo -e '\nERROR: Build errors found during clang-tidy run.' >> ${UP_DIR}/code-checks.md
      echo '```' >> ${UP_DIR}/code-checks.md
      grep -A 3 ': error: \|gmake: \*\*\*' ${UP_DIR}/code-checks.log | tail -24 | sed "s|$CMSSW_BASE/src/||" >> ${UP_DIR}/code-checks.md || true
      echo '```' >> ${UP_DIR}/code-checks.md
      if ! $DRY_RUN ; then
        send_jenkins_artifacts ${UP_DIR}/ pr-code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}
        ${CMS_BOT_DIR}/comment-gh-pr.py -r ${REPOSITORY} -p $PULL_REQUEST -R ${UP_DIR}/code-checks.md
      fi
      #exit 0
      exit 0
    fi
    if [ -e ${CMSSW_BASE}/tmp/${SCRAM_ARCH}/code-checks-logs ] ; then
      mv ${CMSSW_BASE}/tmp/${SCRAM_ARCH}/code-checks-logs ${UP_DIR}/
    fi
    pushd $CMSSW_BASE/src
      git diff > ${UP_DIR}/code-checks.patch
      if [ -s ${UP_DIR}/code-checks.patch ] ; then
        git commit -a -m 'auto applied code-checks changes'
        let COMMIT_CHG=${COMMIT_CHG}+1
      fi
    popd
  fi
fi

touch ${UP_DIR}/code-format.patch
if $CODE_FORMAT ; then
  if [ -f ${CMSSW_BASE}/src/.clang-format ] ; then
    grep -v '\.inc$' ${UP_DIR}/all-changed-files.txt > ${UP_DIR}/code-format-files.txt || true
    if [ -s ${UP_DIR}/code-format-files.txt ] ; then
      scram build -k -j $NUM_PROC code-format USER_CODE_FORMAT_FILE="${UP_DIR}/code-format-files.txt" > ${UP_DIR}/code-format.log 2>&1
      pushd $CMSSW_BASE/src
        git diff > ${UP_DIR}/code-format.patch
        cat ${UP_DIR}/code-format.patch
        if [ -s ${UP_DIR}/code-format.patch ] ; then
          git commit -a -m 'auto applied code-formats changes'
          let COMMIT_CHG=${COMMIT_CHG}+1
        fi
      popd
    fi
  fi
fi
```

```
echo "We will now run formatting, and then linting checks"

python3 ../cms-bot/run-python-format.py --inputfile "${UP_DIR}/code-checks-files.txt" --cmsswbase "${CMSSW_BASE}/src"

pushd $CMSSW_BASE/src
  git diff > ${UP_DIR}/python-code-format.patch || true
 popd

cp "python-linting.txt" "${UP_DIR}/python-linting.txt" || true

if [ ! -f "python-linting.txt" ]; then
  echo "Error: python-linting.txt was not created."
fi

if ${MULTIPLE_FILES_CHANGES} ; then
    ${CMS_BOT_DIR}/github_scripts/simultaneous_files_modifications_by_PRs.py ${PULL_REQUEST} > \
       ${UP_DIR}/multiple_files_changes.txt || true
fi

RES="+code-checks"
HOW_TO_RUN=""
if [ ${COMMIT_CHG} -gt 0 ] ; then
  if $APPLY_PATCH ; then
    pushd $CMSSW_BASE/src
      if [ ${COMMIT_CHG} -gt 1 ] ; then
        git reset --soft HEAD~${COMMIT_CHG}
        git commit -a -m 'auto applied code-checks/format changes'
      fi
      curl -s https://api.github.com/repos/${REPO_USER}/cmssw/pulls/${PULL_REQUEST}  > pr.json
      export CMSBOT_PYTHON_CMD=$(which python3 >/dev/null 2>&1 && echo python3 || echo python)
      TEST_BRANCH=$(${CMSBOT_PYTHON_CMD} -c "import json,sys;obj=json.load(open('pr.json'));print(obj['head']['ref'])")
      TEST_REPO=$(${CMSBOT_PYTHON_CMD} -c "import json,sys;obj=json.load(open('pr.json'));print(obj['head']['repo']['full_name'])")
      if ! $DRY_RUN ; then
        git push  git@github.com:${TEST_REPO} HEAD:${TEST_BRANCH}
      fi
    popd
    exit 0
  fi
```

```
  RES="-code-checks"
  HOW_TO_RUN="\n\nCode check has found code style and quality issues which could be resolved by applying following patch(s)"
  if [ -s ${UP_DIR}/code-checks.patch ] ; then
    HOW_TO_RUN="${HOW_TO_RUN}\n\n- **code-checks**:"
    HOW_TO_RUN="${HOW_TO_RUN}\nhttps://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/code-checks.patch"
    HOW_TO_RUN="${HOW_TO_RUN}\ne.g. \`curl -k https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/code-checks.patch | patch -p1\`"
    HOW_TO_RUN="${HOW_TO_RUN}\nYou can also run \`scram build code-checks\` to apply code checks directly"
  fi

  if [ -s ${UP_DIR}/code-format.patch ] ; then
    HOW_TO_RUN="${HOW_TO_RUN}\n\n- **code-format**:"
    HOW_TO_RUN="${HOW_TO_RUN}\nhttps://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/code-format.patch"
    HOW_TO_RUN="${HOW_TO_RUN}\ne.g. \`curl -k https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/code-format.patch | patch -p1\`"
    HOW_TO_RUN="${HOW_TO_RUN}\nYou can also run \`scram build code-format\` to apply code format directly"
  fi

  if [ -s ${UP_DIR}/python-code-format.patch ] ; then
    HOW_TO_RUN="${HOW_TO_RUN}\n\n- **python-code-format**:"
    HOW_TO_RUN="${HOW_TO_RUN}\nhttps://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/python-code-format.patch"
    HOW_TO_RUN="${HOW_TO_RUN}\ne.g. \`curl -k https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/python-code-format.patch | patch -p1\`"
    HOW_TO_RUN="${HOW_TO_RUN}\nYou can also run \`python3 PFA.py <file> or python3 run-python-format.py --inputfile <code-checks-files.txt> --cmsswbase <basepath>\` to apply Python code formatting directly"
  fi
fi



MSG="\n\nLogs: $UP_URL"
if [ -s ${UP_DIR}/invalid-filenames.txt ] ; then
    RES="-code-checks"
    MSG="${MSG}\n\n- Error: Found path names starting with digits: ${UP_URL}/invalid-filenames.txt"
fi
if [ -s ${UP_DIR}/duplicate-data.txt ] ; then
    RES="-code-checks"
    MSG="${MSG}\n\n- Error: Found data files available in both cmssw and cms-data. Please update these data files directly in cms-data repository: ${UP_URL}/duplicate-data.txt"
fi
if [ -s ${UP_DIR}/invalid_files.txt ] ; then
    MSG="${MSG}\n\n- Found files with invalid states:\n$(cat ${UP_DIR}/invalid_files.txt | sed 's|^\(\s*\)|\1  - |')"
fi

if [ -s ${UP_DIR}/multiple_files_changes.txt ] ; then
    MSG="${MSG}\n\n- There are other open Pull requests which might conflict with changes you have proposed:"
    MSG="${MSG}\n$(cat ${UP_DIR}/multiple_files_changes.txt | grep -v '^ *$'  | sed 's|^\(\s*\)|\1  - |')"
fi
```

```
MSG="${RES}${MSG}"
echo -e "${MSG}${HOW_TO_RUN}" > ${UP_DIR}/code-checks.md
if ! $DRY_RUN ; then
  send_jenkins_artifacts ${UP_DIR}/ pr-code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}
  eval `scram unset -sh`
  ${CMS_BOT_DIR}/comment-gh-pr.py -r ${REPOSITORY} -p $PULL_REQUEST -R ${UP_DIR}/code-checks.md
fi

if [ -s ${UP_DIR}/python-code-format.patch ] ; then
  echo "INFO: We're applying PEP8 formatting to CMSSW using Ruff; while Ruff found formatting errors in the PR's Python files, this non-blocking check may flag untouched lines, so feel free to contribute by applying the suggested patch."
>> ${UP_DIR}/python-code-checks.md

  echo -e "\n\n- **python-code-format**:" >> ${UP_DIR}/python-code-checks.md
  echo -e "https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/python-code-format.patch" >> ${UP_DIR}/python-code-checks.md
  echo -e "e.g. \`curl -k https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/python-code-format.patch | patch -p1\`" >> ${UP_DIR}/python-code-checks.md
  echo -e "You can also run \`python3 run-python-format.py --inputfile <code-checks-files.txt> --cmsswbase <basepath>\` to apply Python code formatting directly" >> ${UP_DIR}/python-code-checks.md

  echo -e "\n\n- **python-code-linting**:" >> ${UP_DIR}/python-code-checks.md
  file="python-linting.txt"
  content=$(<"$file")
  if [ "$content" == "All checks passed!" ]; then
    echo -e "Status: All linting checks passed!  \xF0\x9F\x98\x8A " >> ${UP_DIR}/python-code-checks.md
  else
    echo -e "Status: There are some linting recommendations that need your attention. \xF0\x9F\x98\xA2" >> ${UP_DIR}/python-code-checks.md
    echo -e "https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/python-linting.txt" >> ${UP_DIR}/python-code-checks.md
    echo -e "You can also run \`ruff check --fix\` to automatically apply fixes." >> ${UP_DIR}/python-code-checks.md
  fi
  if ! $DRY_RUN ; then
    ${CMS_BOT_DIR}/comment-gh-pr.py -r ${REPOSITORY} -p $PULL_REQUEST -R ${UP_DIR}/python-code-checks.md
  fi
else
  echo "INFO: We're applying PEP8 formatting to CMSSW using Ruff; while Ruff found formatting errors in the PR's Python files, this non-blocking check may flag untouched lines, so feel free to contribute by applying the suggested patch."
>> ${UP_DIR}/python-code-checks.md
  echo -e "\n\n- **python-code-linting**:" >> ${UP_DIR}/python-code-checks.md
    file="python-linting.txt"
    content=$(<"$file")
    if [ "$content" == "All checks passed!" ]; then
      echo -e "Status: All linting checks passed!  \xF0\x9F\x98\x8A " >> ${UP_DIR}/python-code-checks.md
    else
      echo -e "Status: There are some linting recommendations that need your attention. \xF0\x9F\x98\xA2" >> ${UP_DIR}/python-code-checks.md
      echo -e "https://cmssdt.cern.ch/SDT/${JENKINS_PREFIX_STR}code-checks/${REPO_USER}-PR-${PULL_REQUEST}/${BUILD_NUMBER}/python-linting.txt" >> ${UP_DIR}/python-code-checks.md
      echo -e "You can also run \`ruff check --fix\` to automatically apply fixes." >> ${UP_DIR}/python-code-checks.md
      ${CMS_BOT_DIR}/comment-gh-pr.py -r ${REPOSITORY} -p $PULL_REQUEST -R ${UP_DIR}/python-code-checks.md
    fi
fi
                                                                                                   283,2          Bot
```

C run-python-format.py

```python
#!/usr/bin/python3
import os
import argparse
def main():
    parser = argparse.ArgumentParser(description="Run Python formatting and linting.")
    parser.add_argument(
        "--inputfile",
        required=True,
        help="Path to the file containing the list of files to process.",
    )
    parser.add_argument(
        "--cmsswbase",
        required=True,
        help="Path to the CMSSW base directory.",
    )
    args = parser.parse_args()

    input_file = args.inputfile
    cmssw_base = args.cmsswbase

    if not os.path.isfile(input_file):
        print("Error: {} does not exist.".format(input_file))
        return
    try:
        with open(input_file, "r") as file:
            files_list = [
                os.path.join(cmssw_base, line.strip()) for line in file if line.strip()
            ]
    except IOError as e:
        print("Error reading {}: {}".format(input_file, e))
        return
    with open("python-linting.txt", "w") as linting_output:
        if not files_list:
            linting_output.write("No files to check.\n")
            print("No files to check. Exiting.")
            return

        all_checks_passed = True
        for file in files_list:
            if os.path.isfile(file):
                check_command = "ruff check {} >> python-linting.txt".format(file)
                result = os.system(check_command)
                if result != 0:
                    all_checks_passed = False

        if all_checks_passed:
            linting_output.write("All checks passed!\n")

    print("Python linting completed. Check 'python-linting.txt' for details.")
    pfa_command = (
        "python3 ../cms-bot/PFA.py "
        + " ".join(files_list)
    )

    format_result = os.system(pfa_command)

    if format_result == 0:
        print("Successfully formatted files.")
    else:
        print("An error occurred while running PFA.py. Exit code: {}".format(format_result))

if __name__ == "__main__":
```

D PFA.py

```python
#!/usr/bin/python3
import os
import argparse
def find_file_path(file):
    if not os.path.isfile(file):
        return None
    return os.path.realpath(file)
def find_python_files(directory):
    py_files = []
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(".py"):
                py_files.append(os.path.join(root, file))
    return py_files
def CodeQualityChecks(python_files):
    if python_files:
        for file in python_files:
            file_path = find_file_path(file)
            if not file_path:
                print("File {} not found or invalid path.".format(file))
                continue

            format_command = "ruff format {}".format(file_path)
            print("Executing command: {}".format(format_command))  # Debug print
            os.system(format_command)
def main():
    parser = argparse.ArgumentParser(
        description="Linting and formatting Python files in directories or file paths."
    )
    parser.add_argument(
        "paths",
        nargs="+",
        help="List of directories or file paths to process"
    )
    args = parser.parse_args()

    all_python_files = []
    for path in args.paths:
        if os.path.isdir(path):
            all_python_files.extend(find_python_files(path))
        elif os.path.isfile(path):
            all_python_files.append(path)
        else:
            print("Error: {} is not a valid file or directory.".format(path))
            return
    CodeQualityChecks(all_python_files)
if __name__ == "__main__":
    main()
```

References

[1] CMS Collaboration. *CMS Software (CMSSW)*. Accessed: 2024-08-06. 2024. URL: https://github.com/cms-sw/cmssw.

[2] The Black Team. *Black Documentation*. Accessed: 2024-08-12. 2024. URL: https://black.readthedocs.io/en/stable/.

[3] The Pylint Team. *Pylint Documentation*. Accessed: 2024-08-12. 2024. URL: https://pylint.readthedocs.io/en/stable/.

[4] The Flake8 Team. *Flake8 Documentation*. Accessed: 2024-08-12. 2024. URL: https://flake8.pycqa.org/en/latest/.

[5] Astral. *Ruff Documentation*. Accessed: 2024-08-12. 2024. URL: https://docs.astral.sh/ruff/.

[6] CERN. *cms-bot*. Accessed: 2024-08-21. 2024. URL: https://github.com/cms-sw/cms-bot.

[7] CERN. *cms-sw*. Accessed: 2024-08-21. 2024. URL: https://github.com/cms-sw/cmssw.

[8] CERN. *CMSSDT Jenkins Documentation*. Accessed: 2024-08-12. 2024. URL: https://cmssdt.cern.ch/jenkins/.

[9] CERN. *CMSSW Continuous Integration and Code Quality Workshop*. Accessed: 2024-08-12. 2024. URL: https://indico.cern.ch/event/1442038/.

[10] CERN. *Core Software Meeting*. Accessed: 2024-08-21. 2024. URL: https://indico.cern.ch/event/1442038/.

[11] CERN. *CMSSW Continuous Integration Infrastructure Documentation*. Accessed: 2024-08-12. 2024. URL: https://codimd.web.cern.ch/xSQXD-oWTj-SipkhtjJHJg?edit.

[12] Astral. *Ruff Configuration*. Accessed: 2024-08-12. 2024. URL: https://docs.astral.sh/ruff/configuration/.

[13] CERN. *Test folder placement CMSSW*. Accessed: 2024-08-21. 2024. URL: https://github.com/cms-sw/cmssw/tree/master/CalibCalorimetry/EcalCorrectionModules.

[14] B.T.Kristinsson. *Github PR*. Accessed: 2024-08-21. 2024. URL: https://github.com/cms-sw/cmssw/pull/45616.