Engineer

Ruff

🖌 Linter et formateur rapides pour Python

Ce que c’est

Ruff est un programme unique en ligne de commande, écrit en Rust, qui vérifie le code source Python selon un catalogue de règles de lint et réorganise sa mise en forme pour lui donner un style cohérent. Il est soutenu par Astral, l’entreprise derrière uv et ty, et deux de ses sous-commandes assurent le travail quotidien : `ruff check` est le linter et `ruff format`, le formateur. Il annonce une vitesse de 10 à 100 fois supérieure à celle de Flake8 et Black, conserve un cache pour ne pas analyser de nouveau les fichiers inchangés et peut remplacer Flake8 avec des dizaines de ses extensions, Black, isort, pydocstyle, pyupgrade et autoflake.

Le linter comprend plus de 900 règles, dont beaucoup sont des réécritures natives d’extensions Flake8 populaires. Chaque règle possède un code composé d’un court préfixe et de trois chiffres, et le préfixe indique son origine : F401, un import inutilisé, vient de Pyflakes, et E501, une ligne trop longue, de pycodestyle. Une configuration sélectionne les règles par ces codes, individuellement ou par préfixe : `select = ["E", "F"]` active donc toutes celles dont le code commence par E ou F, et le sélecteur spécial `ALL` les active toutes. Beaucoup de règles proposent une correction, classée comme sûre, censée préserver le comportement à l’exécution, ou comme non sûre, susceptible de le modifier ; `ruff check --fix` n’applique que les corrections sûres, sauf si l’on ajoute aussi `--unsafe-fixes`.

Le formateur est conçu pour remplacer Black directement. Dans des bases de code comme Django et Zulip, plus de 99,9 % des lignes produites sont identiques, mais la documentation indique clairement que les deux outils ne sont pas destinés à être utilisés en alternance sur le même code. Comme Black, il ignore les zones délimitées par `# fmt: off` et `# fmt: on`.

Les deux parties lisent la même configuration, placée sous `[tool.ruff]` dans `pyproject.toml`, ou dans un fichier `ruff.toml` ou `.ruff.toml`. Chaque fichier Python est vérifié selon le fichier de configuration le plus proche en remontant les répertoires, et ce fichier est utilisé en entier : Ruff ne fusionne rien depuis les répertoires parents, sauf si le fichier le plus proche désigne son parent avec `extend`. Les options de ligne de commande prennent le pas sur les réglages de tous les fichiers. Sans réglage explicite, les lignes peuvent atteindre 88 caractères, comme dans Black, et la version de Python supposée par les règles provient du champ `requires-python` du projet si aucune configuration n'en indique une.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que Ruff ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une seule équipe transforme une idée en un produit auquel vos clients peuvent se connecter, sans le faire passer entre trois prestataires.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Application GIS : revenus multipliés par 10 — 10x croissance du chiffre d'affaires portée par le produit cartographique

Application GIS : revenus multipliés par 10 Contactez-nous →

Pourquoi c'est important

La vitesse détermine où une vérification peut s'exécuter. Un linter assez lent pour interrompre une personne est repoussé vers l'intégration continue, où il rend son verdict après que l'auteur est passé à autre chose. Un outil qui répond vite peut s'exécuter dans l'éditeur et dans un hook de commit, pendant que la modification est encore fraîche.

Le regroupement évite les désaccords entre outils. Un projet qui vérifie le code avec Flake8 et ses extensions, trie les imports avec isort et formate avec Black doit maintenir la cohérence entre plusieurs configurations et versions épinglées. Ruff propose une configuration, une version à épingler et une numérotation des règles, et publie la liste des règles de lint qui entrent en conflit avec son propre formateur, afin de pouvoir les accorder délibérément.

Ses codes de sortie en font un contrôle bloquant sans programme intermédiaire. `ruff check` se termine avec 0 s'il ne reste aucune violation ou si elles ont toutes été corrigées, avec 1 s'il en reste, et avec 2 si la configuration ou la ligne de commande est invalide ou si Ruff lui-même échoue, comme ESLint et Prettier.

`ruff format --check` se termine avec 1 si un fichier devrait être reformaté. Une règle est soit sélectionnée, soit désactivée, et toute violation non corrigée d'une règle sélectionnée fait passer le code de sortie à 1.

En contrepartie, quelqu'un doit assumer le choix des règles. Ruff conseille lui-même de les énumérer explicitement avec `select` plutôt que d'empiler les `extend-select`, de commencer par E et F puis d'ajouter un groupe à la fois, et d'utiliser `ALL` avec discernement, car il active de nouvelles règles à chaque mise à niveau.

À quoi cela ressemble en pratique

Ruff couvrait Python dans un dispositif imposant zéro avertissement dans six langages, où chaque langage disposait de son propre linter. Chaque diagnostic comptait comme une erreur et les suppressions locales étaient interdites, ce qui excluait pour Ruff le commentaire `# noqa` : un problème Python devait être corrigé dans le code, sans autre échappatoire. Les hooks de pré-commit et de pré-push lançaient les linters et les tests, si bien que la vérification s'exécutait avant même la création du commit.

Dans l'état documenté de l'outillage en octobre 2026, ce site confiait à Ruff une mission plus restreinte : les 22 scripts auxiliaires qui accompagnent la construction Hugo. La page des crédits cite Ruff parmi les outils qui maintiennent la rigueur des sources. Le jeu de règles de `pyproject.toml` est mesuré plutôt que choisi au jugé. Le 2026-10-15, chacune des 54 familles de règles de Ruff a été exécutée sur ces scripts, et 39 n'ont rien signalé. Les familles silencieuses dont l'objet correspond à ce code ont été ajoutées aux onze initiales, portant la sélection à 33. La famille de sécurité S reste activée, avec cinq règles ignorées par leur code, chacune accompagnée d'un commentaire expliquant pourquoi le motif signalé est ici la variante sûre d'un motif dangereux ; les autres règles de la famille restent donc bloquantes. Trois règles propres à Ruff sur les caractères Unicode ambigus sont ignorées parce que les textes du site utilisent délibérément de vrais tirets cadratins, demi-cadratins et lettres danoises.

La configuration de pré-commit lance `ruff check` et `ruff format --check` sur tout fichier Python sous `scripts/` qu'un commit modifie, et la tâche de correction du projet lance les deux mêmes commandes sous leur forme qui applique les corrections. À côté de Ruff se trouve basedpyright, un vérificateur de types, avec un commentaire qui délimite leurs responsabilités : un linter ne suit pas une

valeur au-delà d'une frontière de fonction, et, lors de sa première exécution, le vérificateur de types a trouvé deux véritables défauts que Ruff avait laissés passer sans diagnostic.

Où les problèmes surviennent

La plupart des problèmes avec Ruff proviennent d'une configuration plus littérale que les personnes qui l'ont écrite.

Une mise à niveau fait échouer une branche que personne n'a modifiée. Du code accepté la veille signale de nouvelles violations après la mise à jour de Ruff. La cause habituelle est `select = ["ALL"]`, qui adopte chaque règle ajoutée par une version, associé à une version non épinglée. Épinglez la version de Ruff pour qu'une mise à niveau soit un commit relu, et énumérez les familles voulues par le projet au lieu d'utiliser `ALL` : une nouvelle famille n'arrivera ainsi que lorsque quelqu'un l'ajoutera.

Un dossier cesse de suivre les règles du projet. Des violations interdites par la configuration racine passent dans un sous-répertoire, ou une règle désactivée par l'équipe s'y déclenche. Un `ruff.toml`, ou un `pyproject.toml` contenant une table `[tool.ruff]`, se trouve dans ce dossier et, comme Ruff utilise le fichier le plus proche en entier, tous les réglages de la racine cessent de s'appliquer en dessous. `ruff check --show-settings`, exécuté sur un fichier situé là, affiche le chemin de configuration retenu. Supprimez le fichier imbriqué ou faites-le hériter avec `extend = "../pyproject.toml"`.

Une suppression survit à sa raison d'être. Un commentaire `# noqa` avait une bonne raison d'exister, le code a changé, et le commentaire ne masque plus rien, ou masque le problème suivant sur cette ligne. Un simple `# noqa` est pire : il réduit au silence toutes les règles de la ligne. Sélectionnez RUF100, qui signale les commentaires `noqa` ne supprimant plus aucun diagnostic, et PGH004, qui exige un code précis, et laissez `--fix` supprimer ceux qui ne servent plus. Ici, l'import est désormais utilisé : le commentaire ne protège donc rien et RUF100 le signale :

```
import json  # noqa: F401

print(json.dumps({"ok": True}))
```

Le formateur et le linter ne sont pas d'accord. `ruff format` réécrit un fichier, puis `ruff check` signale le résultat. Certaines règles de lint contrôlent ce que décide le formateur, comme les virgules finales (COM812), l'indentation (W191, E111) et le style des guillemets (les règles Q). Le formateur ne peut par ailleurs respecter la longueur de ligne que dans la mesure du possible : E501 peut donc encore se déclencher sur du code formaté. Lisez les avertissements affichés par `ruff format`, car la documentation donne elle-même leur absence comme critère d'accord entre les deux ; ignorez toute règle citée par un avertissement et considérez E501 comme une invitation à restructurer la ligne.

Une correction en masse change le comportement. Après un `ruff check --fix --unsafe-fixes` sur toute l'arborescence, des tests échouent ou un gestionnaire d'erreurs cesse d'intercepter ce qu'il interceptait auparavant. Les corrections non sûres peuvent modifier le comportement à l'exécution ou supprimer des commentaires, et l'exemple de la documentation change lui-même l'exception levée par un morceau de code. Gardez `--unsafe-fixes` hors des hooks et des scripts, appliquez les corrections non sûres une règle à la fois avec `--select` et lisez le diff, puis ne classez les corrections d'une règle comme sûres avec `extend-safe-fixes` qu'après sa vérification par l'équipe.

Pour aller plus loin

- Ruff — Ce que Ruff remplace, la vitesse qu'il annonce et l'organisation qui le maintient.
- Le linter Ruff — Sélection et priorité des règles, corrections sûres et non sûres, commentaires `noqa` et codes de sortie.
- Le formateur Ruff — Compatibilité avec Black et règles de lint en conflit avec le formatage.
- Règles — Chaque règle par code, son origine et la possibilité de la corriger.
- Configurer Ruff — Découverte des fichiers, règle du fichier le plus proche, `extend` et options de ligne de commande.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que Ruff ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/ruff/