

Python/Djangoの環境構築 (チーム開発・大規模開発向け)

チームでの開発・大規模開発の特徴と懸念

1. タスク分担や作業期間の長期化によって、実装知識が偏る
自分の担当した機能は把握しているが、そうでない部分は内容を知らない、といったことが起こる
担当部分が変わったり、範囲が増えたときにスムーズに対応できるようにしておく必要がある
(一人で開発している場合も、時間が空いたことで実装方法や対応内容を忘れることがある)
2. コードスタイルの不統一
開発メンバーや、実装した時期によって、コードの書き方に違いが出てしまう
ちょっとしたことでも、多くのファイルにまたがってコードを読んでいくと混乱のもとになりやすい
3. 実装の変更による、予期せぬ不具合の発生
プログラミングにおいて最も多いエラーのひとつは、null アクセスと未定義エラーと言われている
Pythonをはじめとする動的型付け言語は、コンパイルという工程がない分これらのエラーを発見しづらい
※型を意識しなくてよい手軽さとのトレードオフであり、大規模開発ではデメリットも大きい

懸念をツールの利用と仕組みで改善する

- チーム開発・大規模開発の懸念や不安点を、開発メンバーの注意力やレビューだけで払しょくすることは簡単ではない
- そこで、コードの品質を確保するためのツールを導入し、開発フローに取り入れて仕組化する
- 開発メンバーは、アルゴリズムや業務要件の実装に集中し、生産性を高めることができる

Python/Django のチーム開発用環境構築

1. pyenv
Python 本体のバージョン管理ツール。複数のPythonのバージョンを切り替えて利用できるようにする。
2. Poetry
Python パッケージを管理し、依存関係を解決するツール。pip では機能不足のため、これを利用する。
3. mypy
Python の型チェックツール。型不一致によるエラーを実行前に発見できる。
開発時のコード補完表示で実装速度も向上する。
4. Black
PEP8 のスタイルに従うためのコードフォーマッター。メンバー間のコードスタイルの違いを解消できる。
5. Flake8
PEP8 のコーディングルール違反を確認するリンター。コードスタイルだけでなく、不具合の原因になりうる記述に対しても警告してくれる。

※ 自動テスト関連は別資料でまとめます

1-1. pyenvの導入 (Windows)

[pyenv-win の公式ページの手順](#)に従ってインストールを進める
PowerShell を起動し、以下のコマンドを実行

```
Invoke-WebRequest -UseBasicParsing -Uri "https://raw.githubusercontent.com/pyenv-win/pyenv-win/master/pyenv-win/install-pyenv-win.ps1" -OutFile "./install-pyenv-win.ps1"; &"./install-pyenv-win.ps1"
```

Unauthorized エラーが発生する場合は、PowerShell を管理者権限で開いたPowerShellのプロンプトから以下のコマンドを実行し、上記のインストールコマンドを再実行する

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine
```

pyenv コマンドを実行し、コマンドヘルプが表示されれば完了

1-2. pyenvの利用

1. インストール可能なバージョンの一覧を表示

```
$ pyenv install --list
```

Available versions:

2.1.3

2.2.3

... 以下略

2. バージョンを指定してインストール(3.10.x)

```
$ pyenv install 3.10
```

Downloading Python-3.10.8.tar.xz...

-> <https://www.python.org/ftp/python/3.10.8/Python-3.10.8.tar.xz>

Installing Python-3.10.8...

Installed Python-3.10.8 to /home/y-uchiida/.pyenv/versions/3.10.8

3. デフォルトで動作するバージョンを設定

```
$ pyenv global 3.10
```

```
$ python --version
```

Python 3.10.8

4. 現在のディレクトリで動作するバージョンを指定

```
$ pyenv local 3.10
```

2-1. Poetry の導入 (1)

[Poetry の公式ページの手順](#)に従ってインストールを進める
PowerShell を起動し、以下のコマンドを実行

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | py -
```

py コマンドが見つからないとエラーが出る場合は、代わりに以下を実行 (py をpython に置き換える)

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | python -
```

インストールコマンドを実行した後、Poetry の実行プログラムに対してPATHを通す
環境変数の設定画面を開き、PATHを設定する

2-1. Poetry の導入 (2)

```
PS C:\Users\y-uchiida\Documents\develop> (Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | python -  
Retrieving Poetry metadata
```

```
# Welcome to Poetry!
```

```
This will download and install the latest version of Poetry,  
a dependency and package manager for Python.
```

```
It will add the 'poetry' command to Poetry's bin directory, located at:
```

```
C:\Users\y-uchiida\AppData\Roaming\Python\Scripts
```

```
You can uninstall at any time by executing this script with the --uninstall option,  
and these changes will be reverted.
```

```
Installing Poetry (1.4.0)
```

```
Installing Poetry (1.4.0): Creating environment
```

```
Installing Poetry (1.4.0): Installing Poetry
```

```
Installing Poetry (1.4.0): Creating script
```

```
Installing Poetry (1.4.0): Done
```

```
Poetry (1.4.0) is installed now. Great!
```

```
To get started you need Poetry's bin directory (C:\Users\y-uchiida\AppData\Roaming\Python\Scripts) in your 'PATH'  
environment variable.
```

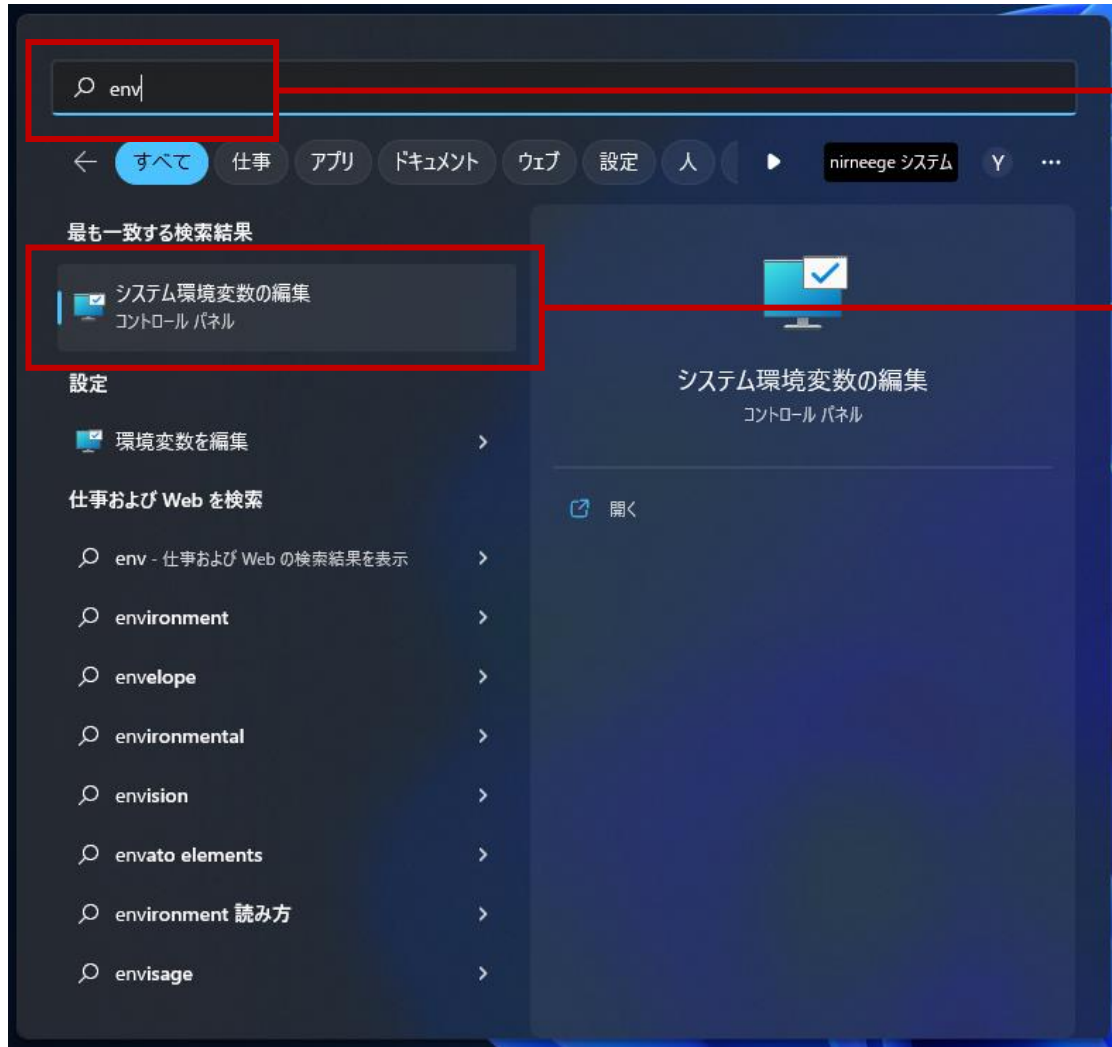
```
Alternatively, you can call Poetry explicitly with 'C:\Users\y-uchiida\AppData\Roaming\Python\Scripts\poetry'.
```

```
You can test that everything is set up by executing:
```

```
'poetry --version'
```

インストールコマンドの実行に成功すると、
コマンドプログラムをインストールしたパスが表示される
このパスを、環境変数に追加する

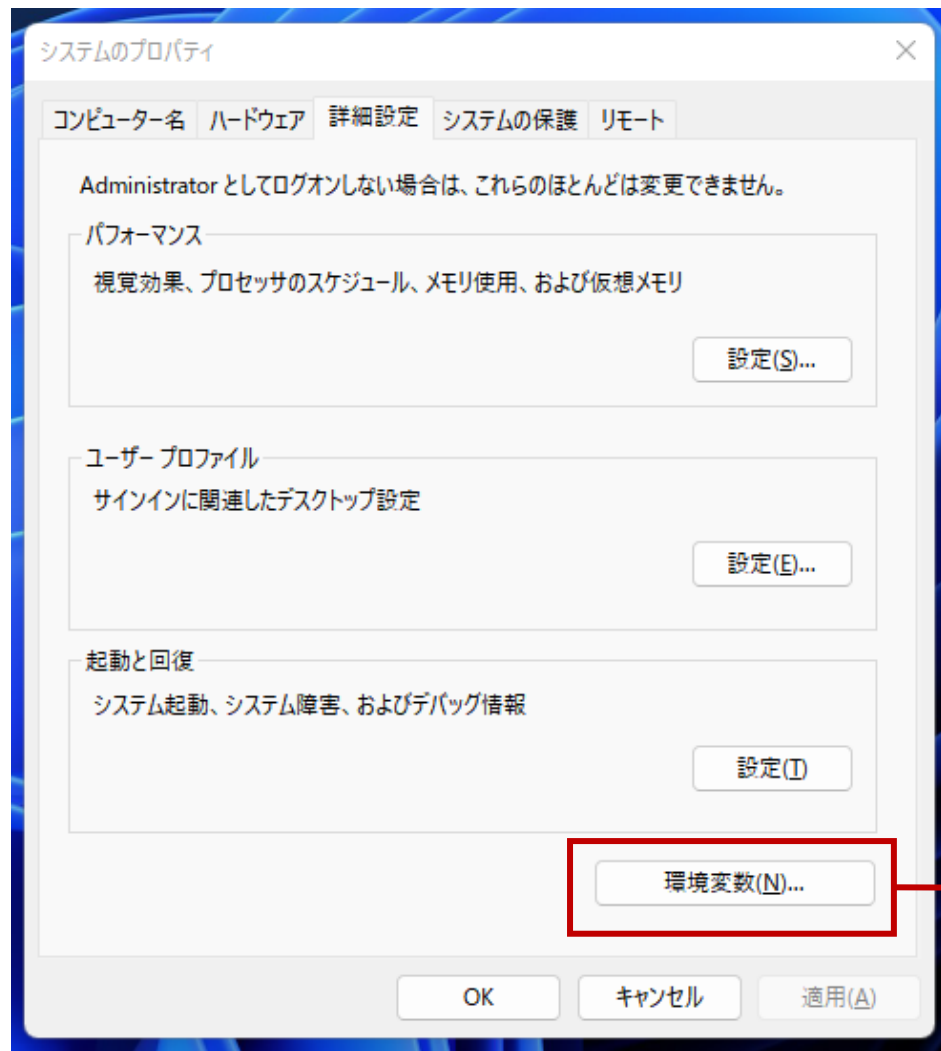
2-1. Poetry の導入 (3)



Windows キーを押してスタートメニューを表示する
検索欄に「env」と入力する

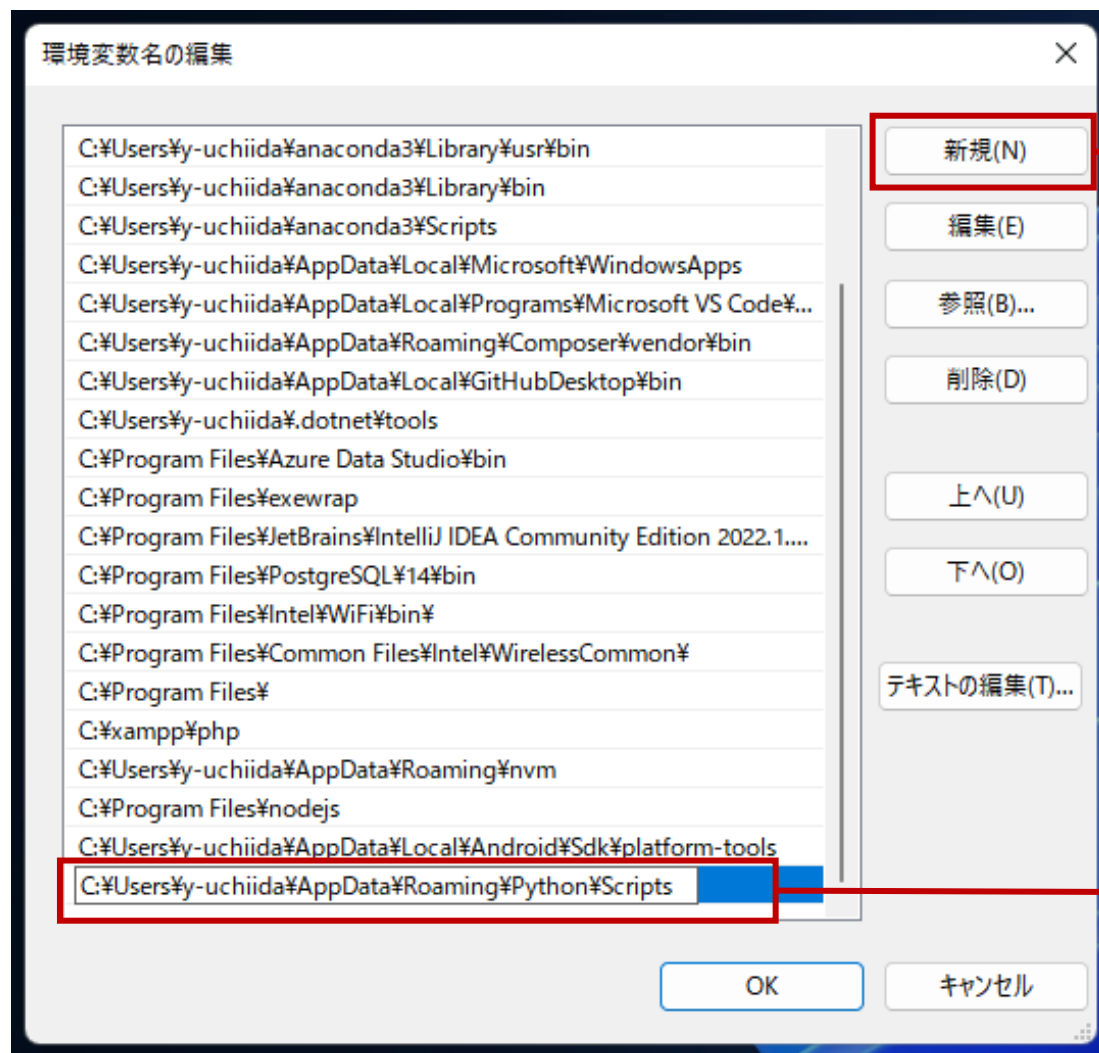
システム環境変数の編集 が表示されるのでクリック

2-1. Poetry の導入 (4)



システムのプロパティウィンドウの
「環境変数」ボタンをクリック

2-1. Poetry の導入 (5)



「新規」ボタンをクリック

入力欄が追加されるので、インストールコマンド実行時に表示されたPoetryのパスを追記

2-1. Poetry の導入 (6)

PowerShell のウィンドウを開いて、以下のコマンドを実行する
Poetry のバージョン情報が表示されれば、インストール完了

```
poetry --version
```

2-2. Poetry の利用 (1)

Poetry は、Pythonアプリケーションのプロジェクトを作成できる
以下のコマンドを実行すると、poetry_new_sample というフォルダにプロジェクトが作成される

```
# 新しくプロジェクト用のフォルダを作る
```

```
poetry new porty_new_sample
```

```
Created package porty_new_sample in porty_new_sample
```

```
# 作成したプロジェクトのフォルダに移動
```

```
cd porty_new_sample
```

```
#作成された内容を表示
```

```
ls
```

```
ディレクトリ: C:\porty_new_sample
```

Mode	LastWriteTime	Length	Name
d----	2023/03/12 22:11		porty_new_sample
d----	2023/03/12 22:11		tests
-a----	2023/03/12 22:11	327	pyproject.toml
-a----	2023/03/12 22:11	0	README.md

2-2. Poetry の利用 (2)

Poetry は、Pythonアプリケーションのプロジェクトを作成できる
以下のコマンドを実行すると、poetry_new_sample というフォルダにプロジェクトが作成される

```
# 新しくプロジェクト用のフォルダを作る
```

```
poetry new porty_new_sample
```

```
Created package porty_new_sample in porty_new_sample
```

```
# 作成したプロジェクトのフォルダに移動
```

```
cd porty_new_sample
```

```
#作成された内容を表示
```

```
ls
```

```
ディレクトリ: C:\porty_new_sample
```

Mode	LastWriteTime	Length	Name
d----	2023/03/12 22:11		porty_new_sample
d----	2023/03/12 22:11		tests
-a----	2023/03/12 22:11	327	pyproject.toml
-a----	2023/03/12 22:11	0	README.md

2-2. Poetry の利用 (3)

Poetry プロジェクトには、専用の仮想環境が作成されるので、venv などの仮想環境ツールは必要ない

```
# 仮想環境に入る
```

```
poetry shell
```

```
Creating virtualenv porty-new-sample-if_XtSRG-py3.9 in C:\Users\y-uchiida\AppData\Local\pypoetry\Cache\virtualenvs
```

```
Spawning shell within C:\Users\y-uchiida\AppData\Local\pypoetry\Cache\virtualenvs\porty-new-sample-if_XtSRG-py3.9
```

```
# 実行後、ターミナルに仮想環境名が表示される (porty-new-sample-py3.9)
```

```
(porty-new-sample-py3.9) PS C:\porty_new_sample>
```

```
# 出る時はexit (deactivate でもOK)
```

```
exit
```

2-2. Poetry の利用 (4)

Poetry プロジェクトにパッケージを追加する場合は、`poetry add` コマンドを利用する
このコマンドでパッケージを追加すると、パッケージのバージョン管理を正確に行える

```
# Django 3.2 を追加  
# poetry.lock ファイルに依存関係が追加される  
# poetry add は、pip install と同じように使える  
poetry add django==3.2
```

```
Updating dependencies  
Resolving dependencies... (0.1s)
```

```
Writing lock file
```

```
Package operations: 4 installs, 0 updates, 0 removals
```

- Installing asgiref (3.6.0)
- Installing pytz (2022.7.1)
- Installing sqlparse (0.4.3)
- Installing django (3.2)

2-2. Poetry の利用 (5)

Poetry プロジェクトからパッケージを削除する場合は、`poetry remove` コマンドを利用する
パッケージを削除すると、依存関係に基づいて不要なパッケージもまとめて削除される

```
# Django パッケージをプロジェクトから削除する
```

```
poetry remove django
```

```
Updating dependencies
```

```
Resolving dependencies...
```

```
Writing lock file
```

```
Package operations: 0 installs, 0 updates, 4 removals
```

- Removing asgiref (3.6.0)
- Removing django (3.2)
- Removing pytz (2022.7.1)
- Removing sqlparse (0.4.3)

3-1. mypy の導入

[mypy の公式ページの手順](#)に従ってインストールを進める
を起動し、以下のコマンドを実行

```
# mypy をインストール (Poetry で作成したプロジェクトの場合)
```

```
poetry add --group dev mypy
```

```
Using version ^1.1.1 for mypy
```

```
Updating dependencies
```

```
Resolving dependencies...
```

```
Writing lock file
```

```
Package operations: 4 installs, 0 updates, 0 removals
```

- Installing mypy-extensions (1.0.0)
- Installing tomli (2.0.1)
- Installing typing-extensions (4.5.0)
- Installing mypy (1.1.1)

3-2. mypy の利用 (1)

型不整合を含むプログラム `mypy_test.py` を作成する

```
import datetime

# datetime 型に、String型の値を代入
var: datetime.datetime = "foo"
```

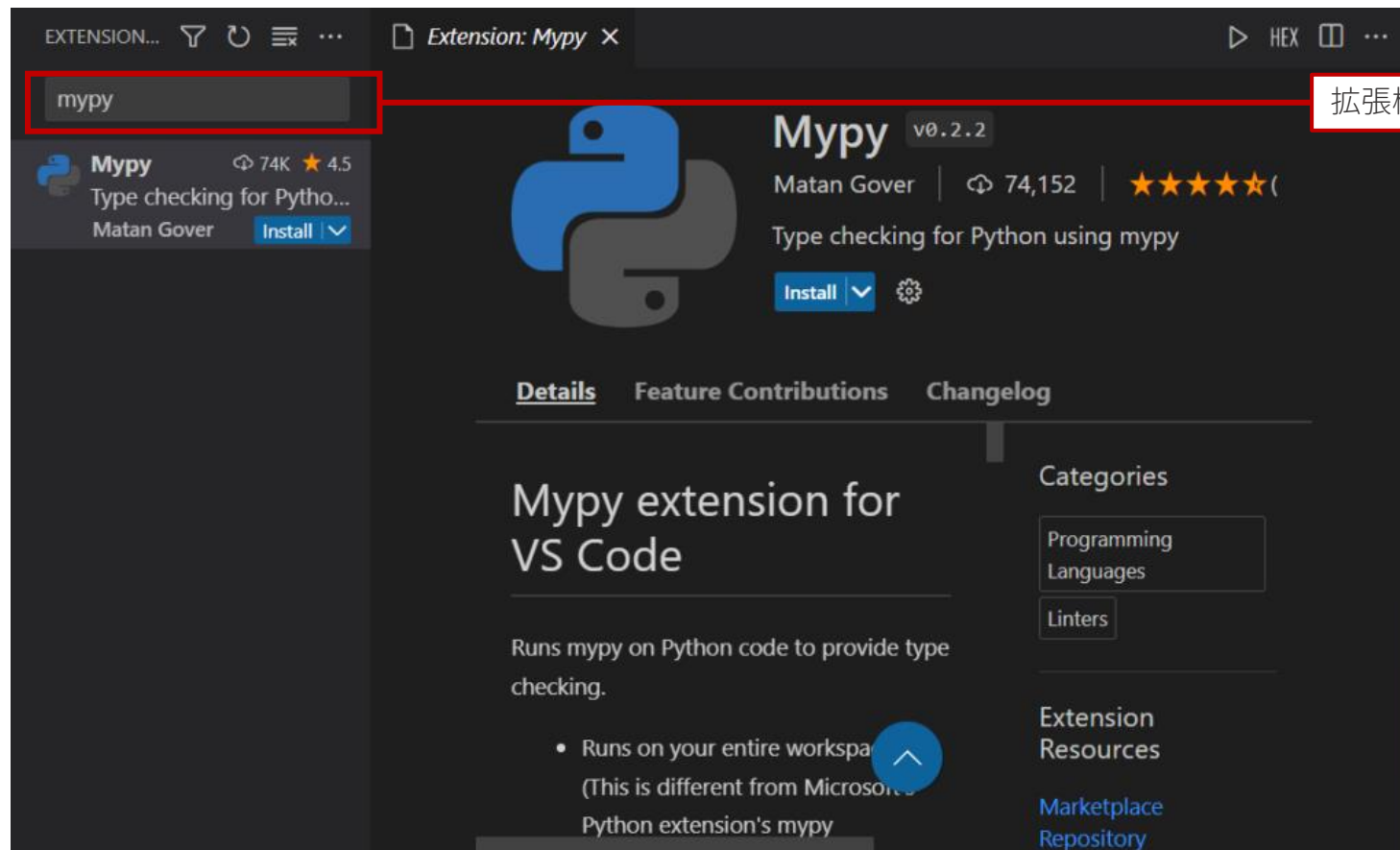
ターミナルから、`mypy` <検査対象のファイル> のコマンドを実行

```
# mypy_test.py を型チェックするコマンド
mypy mypy_test.py

mypy_test.py:4: error: Incompatible types in assignment (expression has type "str", variable has type "datetime") [assignment]
Found 1 error in 1 file (checked 1 source file)
```

3-2. mypy の利用 (2)

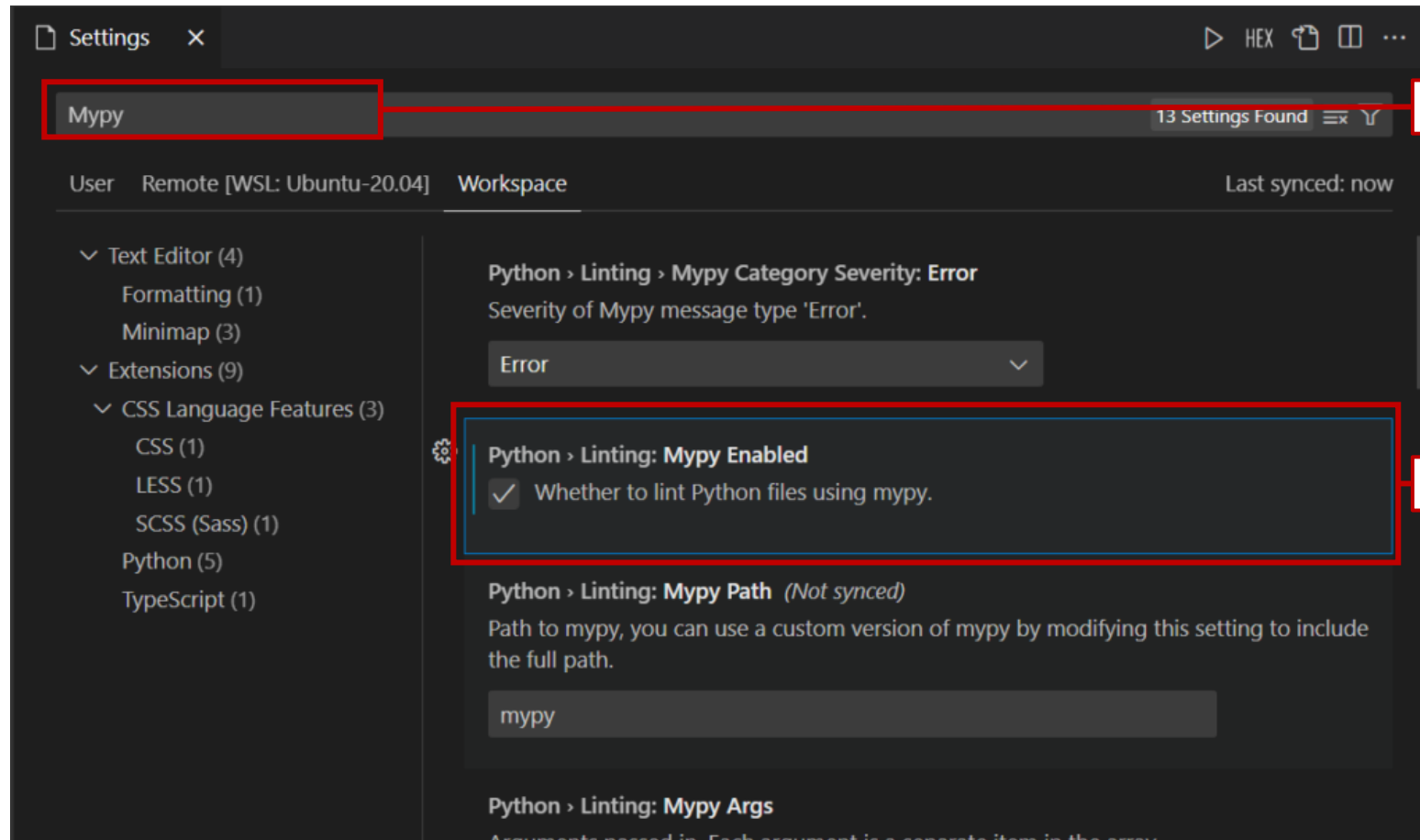
VSCode 上で型エラーの警告を表示するには、拡張機能を導入する



拡張機能 mypy を検索する

3-2. mypy の利用 (3)

VSCode の設定画面から、Mypy を有効化する

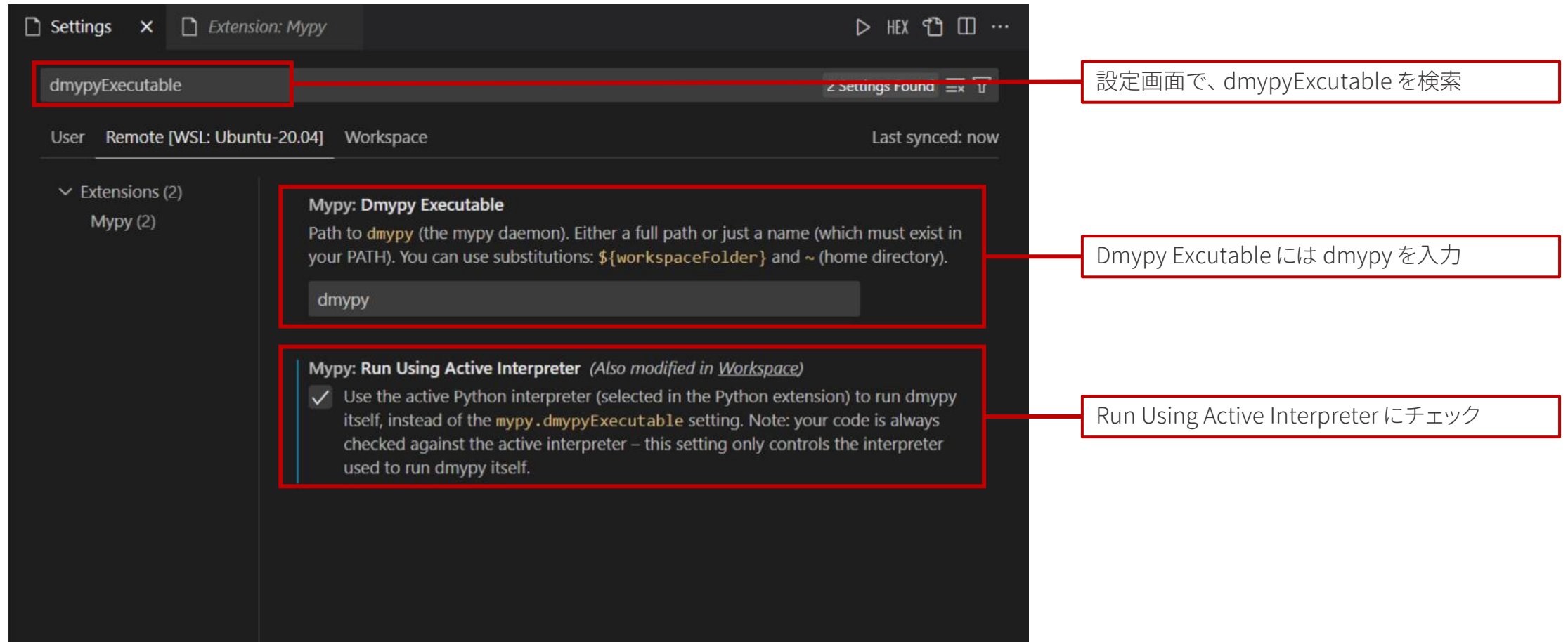


設定画面で、Mypy を検索

Mypy Enabled にチェック

3-2. mypy の利用 (4)

VSCode の設定を開き、Mypy: Run Using Active Interpreter にチェックをつける



The screenshot shows the VS Code settings interface for the Mypy extension. The search bar at the top contains 'dmypyExecutable'. The left sidebar shows 'Extensions (2)' and 'Mypy (2)'. The main panel displays two settings:

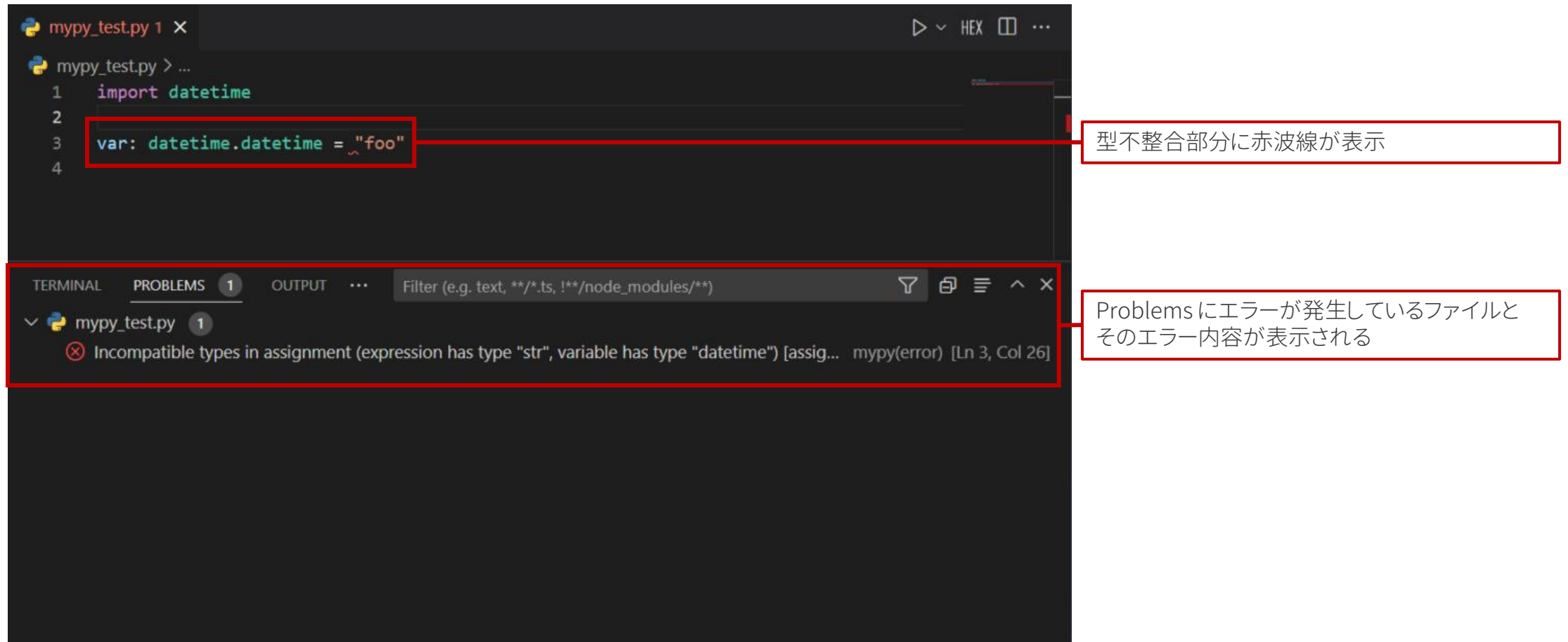
- Mypy: Dmypy Executable**
Path to **dmypy** (the mypy daemon). Either a full path or just a name (which must exist in your PATH). You can use substitutions: `${workspaceFolder}` and `~` (home directory).
The input field contains 'dmypy'.
- Mypy: Run Using Active Interpreter** (Also modified in *Workspace*)
☒ Use the active Python interpreter (selected in the Python extension) to run dmypy itself, instead of the `mypy.dmypyExecutable` setting. Note: your code is always checked against the active interpreter – this setting only controls the interpreter used to run dmypy itself.

Annotations with red boxes and lines point to the following elements:

- The search bar containing 'dmypyExecutable' with the text: 設定画面で、dmypyExecutable を検索
- The input field for 'Mypy: Dmypy Executable' containing 'dmypy' with the text: Dmypy Executable には dmypy を入力
- The checkbox for 'Mypy: Run Using Active Interpreter' with the text: Run Using Active Interpreter にチェック

3-2. mypy の利用 (5)

VSCode の設定を開き、Mypy: Run Using Active Interpreter にチェックをつける



The screenshot shows the VS Code interface with a Python file named `mypy_test.py` open. The code in the editor is:

```
1 import datetime
2
3 var: datetime.datetime = "foo"
4
```

A red box highlights the line `var: datetime.datetime = "foo"`, and a red wavy line is under the string `"foo"`. A red line connects this box to a callout: **型不整合部分に赤波線が表示** (A red wavy line is displayed at the type mismatch part).

Below the editor, the **PROBLEMS** panel is active, showing one error:

- `✖ Incompatible types in assignment (expression has type "str", variable has type "datetime") [assign... mypy(error) [Ln 3, Col 26]`

A red box highlights this error message, and a red line connects it to a callout: **Problems にエラーが発生しているファイルとそのエラー内容が表示される** (The file in which an error has occurred and the error content are displayed in Problems).

4-1. Black の導入

[Black の公式ページの手順](#)に従ってインストールを進める
を起動し、以下のコマンドを実行

```
# Black をインストール(Poetry で作成したプロジェクトの場合)
```

```
poetry add --group dev black
```

```
Updating dependencies
```

```
Resolving dependencies...
```

```
Writing lock file
```

```
Package operations: 9 installs, 0 updates, 0 removals
```

- Installing colorama (0.4.6)
- Installing click (8.1.3)
- Installing mypy-extensions (1.0.0)
- Installing packaging (23.0)
- Installing pathspec (0.11.0)
- Installing platformdirs (3.1.1)
- Installing tomli (2.0.1)
- Installing typing-extensions (4.5.0)
- Installing black (23.1.0)

4-2. Black の利用 (1)

フォーマット違反を含むプログラムformat_test.py を作成する

```
def main():  
    print("this"      +      "is"      +      'main')  
    return           0
```

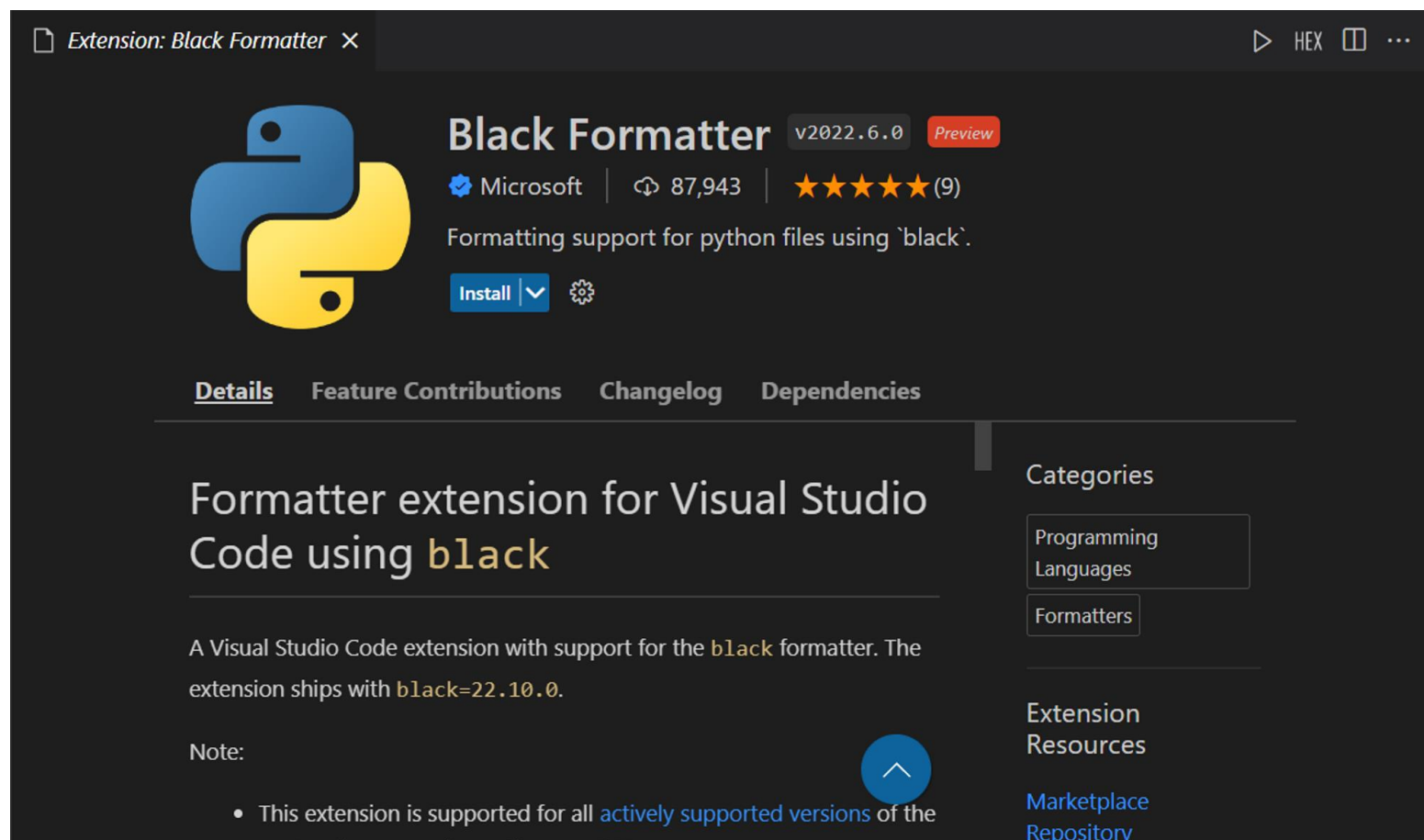
ターミナルから、black <検査対象のファイル> のコマンドを実行

```
# format_test.py を自動フォーマットするコマンド  
black format_test.py
```

```
All done! ✨ 🍰 ✨  
1 file left unchanged.
```


4-2. Black の利用 (2)

VSCodeでBlack を利用する場合、拡張機能をインストールする



The screenshot shows the 'Black Formatter' extension page in the Visual Studio Code marketplace. The extension is by Microsoft, has 87,943 downloads, and a 5-star rating from 9 reviews. It is currently in 'Preview' mode for version v2022.6.0. The description states it provides 'Formatting support for python files using `black`'. The 'Install' button is visible with a dropdown arrow and a settings gear icon. Below the main description, there are tabs for 'Details', 'Feature Contributions', 'Changelog', and 'Dependencies'. The 'Details' tab is active, showing a sub-description: 'Formatter extension for Visual Studio Code using **black**'. It also mentions 'A Visual Studio Code extension with support for the **black** formatter. The extension ships with **black=22.10.0**.' A 'Note' section at the bottom states: 'This extension is supported for all [actively supported versions](#) of the'. On the right side, there are sections for 'Categories' (Programming Languages, Formatters), 'Extension Resources' (Marketplace, Repository), and a 'HEX' icon at the top right.

Extension: Black Formatter ✕

 **Black Formatter** v2022.6.0 Preview

Microsoft | 87,943 | ★★★★★ (9)

Formatting support for python files using `black`.

[Install](#) ▼ ⚙

[Details](#) [Feature Contributions](#) [Changelog](#) [Dependencies](#)

Formatter extension for Visual Studio Code using **black**

A Visual Studio Code extension with support for the **black** formatter. The extension ships with **black=22.10.0**.

Note:

- This extension is supported for all [actively supported versions](#) of the

Categories

- Programming Languages
- Formatters

Extension Resources

- [Marketplace](#)
- [Repository](#)

4-2. Black の利用 (3)

Black の拡張機能でフォーマットされるように、設定画面からdefault formatterを変更

The screenshot shows the VS Code settings interface. The search bar at the top contains the text '@lang:python defaultFormatter'. Below the search bar, the settings list shows the 'Editor: Default Formatter' setting. The dropdown menu for this setting is open, showing several options. The 'Black Formatter' option is selected, and the 'None null' option is marked as 'default'. A red box highlights the search bar and the settings list. A red line points from the search bar to a text box on the right that says '@lang:python DefaultFormatter を検索'. Another red line points from the 'Black Formatter' option to a text box on the right that says 'Default Formatter をBlackに設定'.

@lang:python defaultFormatter

@lang:python DefaultFormatter を検索

Editor: **Default Formatter** (Also modified elsewhere)
Defines a default formatter which takes precedence over all other formatter settings. Must be the identifier of an extension contributing a formatter.

Black Formatter
None null default
Auto Rename Tag formulahendry.auto-rename-tag
autoDocstring - Python Docstring Generator
Black Formatter ms-python.black-formatter
C/C++ Compile Run
Formatting support for python files using black.

Default Formatter をBlackに設定

4-2. Black の利用 (4)

デフォルト設定がautopep8 になっているので、Formatting Provider をblackに変更

The screenshot shows the VS Code Settings interface. The search bar at the top contains the text 'python.formatting.provider', which is highlighted with a red box. To the right of the search bar, a red box contains the text 'python.formatting.provider を検索'. Below the search bar, the 'Workspace' tab is selected. On the left sidebar, 'Extensions (1)' and 'Python (1)' are listed. The main settings area shows 'Python > Formatting: Provider' with a red box around it. Below this, a dropdown menu is set to 'black', also highlighted with a red box. To the right of this dropdown, a red box contains the text 'Formatting Provider を black に変更'. The settings page also shows '1 Setting Found' and 'Last synced: now'.

python.formatting.provider

python.formatting.provider を検索

Python > Formatting: Provider (Also modified in [Workspace](#))

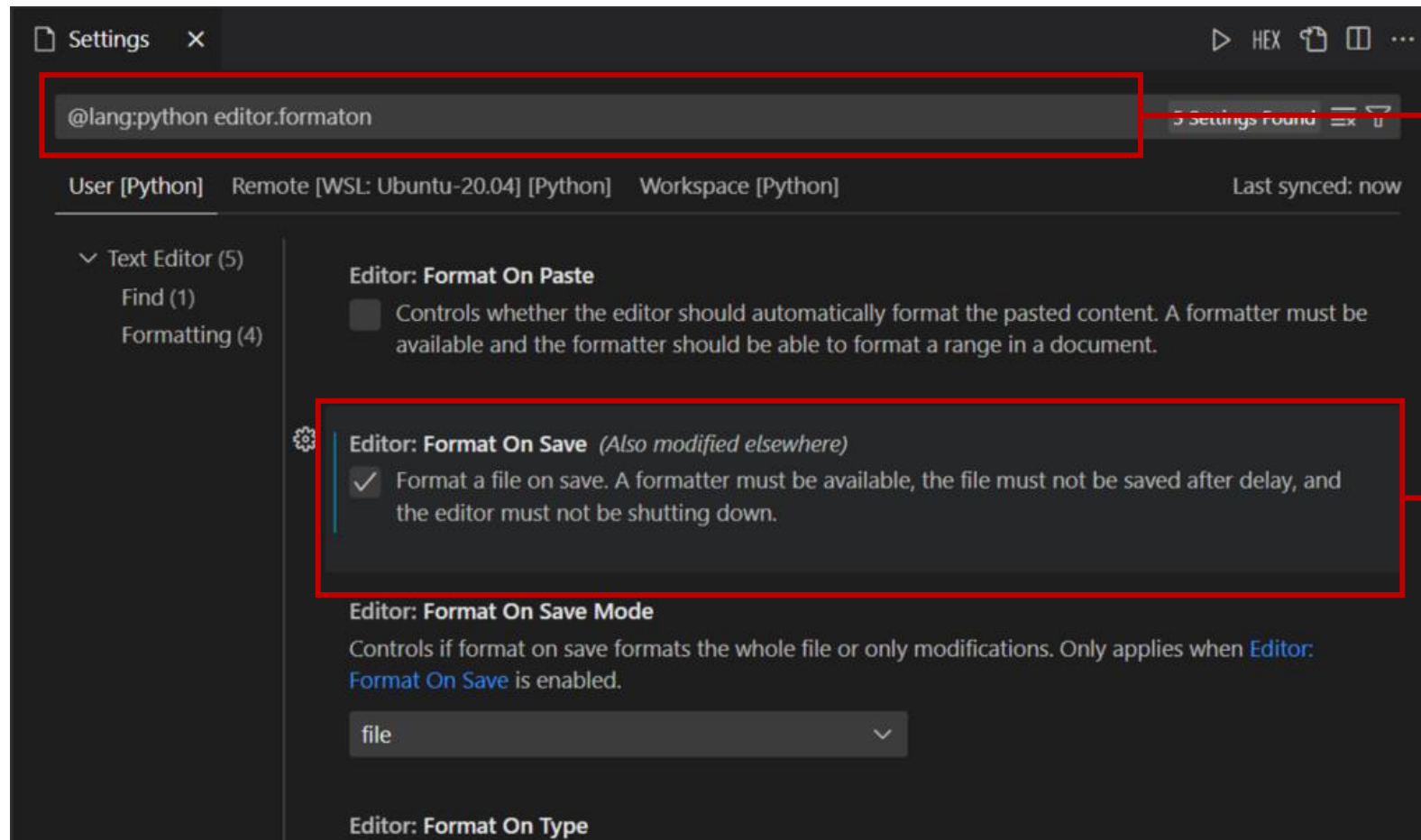
Provider for formatting. Possible options include 'autopep8', 'black', and 'yapf'.

black

Formatting Provider を black に変更

4-2. Black の利用 (5)

VSCode での保存時に字度フォーマットされるように、Format on Save を有効化



@lang:python editor.formaton を
検索

Editor: Format On Save にチェック

5-1. Flake8 の導入

[Flake8 の公式ページの手順](#)に従ってインストールを進める
を起動し、以下のコマンドを実行

```
# Flake8 をインストール(Poetry で作成したプロジェクトの場合)
```

```
poetry add --group dev Flake8
```

```
Using version ^6.0.0 for flake8
```

```
Updating dependencies
```

```
Resolving dependencies...
```

```
Writing lock file
```

```
Package operations: 4 installs, 0 updates, 0 removals
```

- Installing mccabe (0.7.0)
- Installing pycodestyle (2.10.0)
- Installing pyflakes (3.0.1)
- Installing flake8 (6.0.0)

5-2. Flake8 の利用 (1)

PEPのコーディングルール違反を含むプログラムflake8_test.py を作成する

```
if 1 is 1: # リテラルの比較に is は使わない
    print("always true") # インデントはスペース4つにする
```

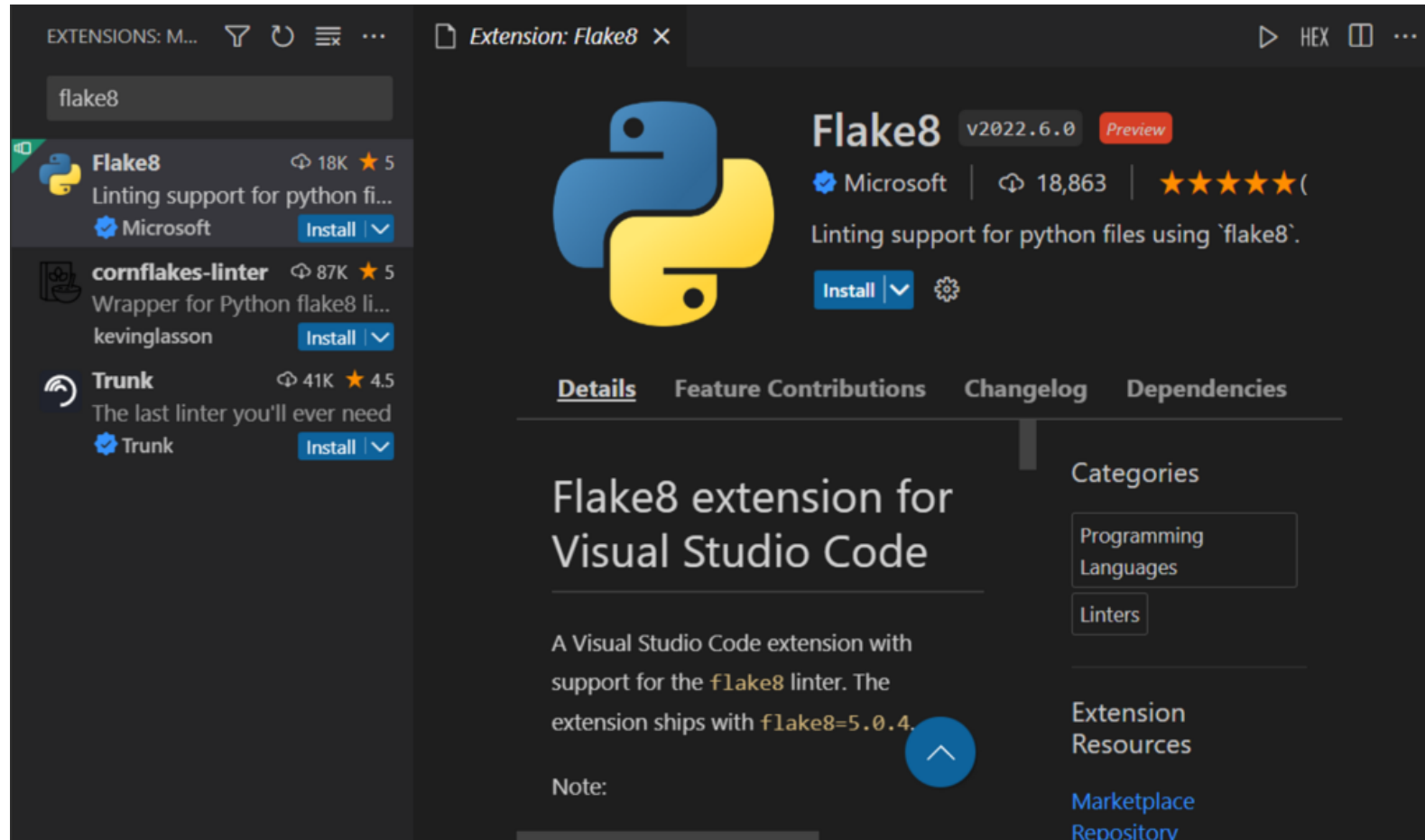
ターミナルから、flake8 <検査対象のファイル> のコマンドを実行

```
# flake8_test.py をチェックするコマンド
flake8 flake8_test.py
```

```
flake8_test.py:1:4: F632 use ==/!= to compare constant literals (str, bytes, int, float, tuple)
flake8_test.py:2:3: E111 indentation is not a multiple of 4
```

5-2. Flake8 の利用(2)

VSCodeでFlake8 を利用する場合、拡張機能をインストールする



5-2. Flake8 の利用 (3)

デフォルトのリンターとして、Pylint が有効になっているので、無効にしておく

The screenshot shows the VS Code Settings interface for the Flake8 extension. The search bar at the top contains the text `python.linting.pylintEnabled`. Below the search bar, the settings are categorized by 'User', 'Remote [WSL: Ubuntu-20.04]', and 'Workspace'. The 'Python > Linting: Pylint Enabled' setting is highlighted, showing a checkbox that is currently checked. The description of this setting is 'Whether to lint Python files using pylint.'.

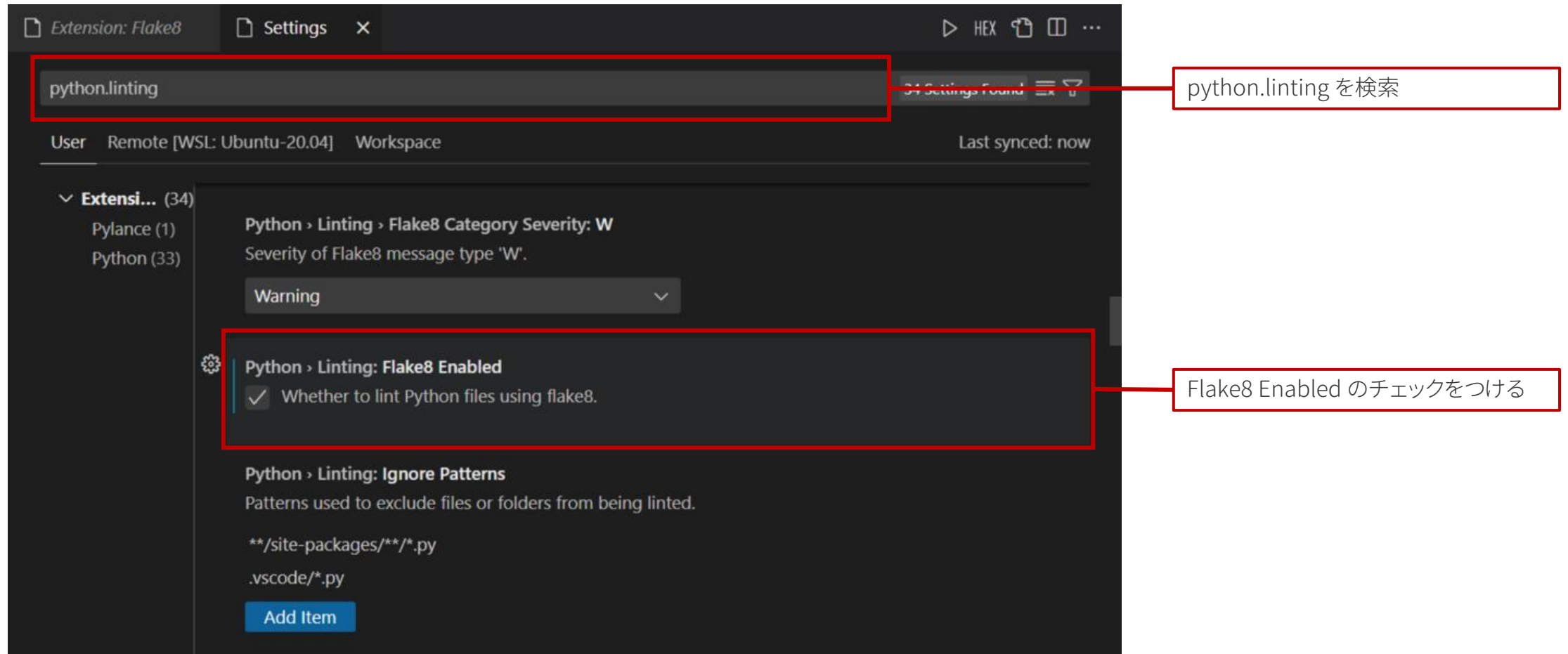
python.linting.pylintEnabled
を検索

Python > Linting: Pylint Enabled
☒ Whether to lint Python files using pylint.

Pylint Enabled のチェックを外す

5-2. Flake8 の利用 (4)

Flake8 の機能を有効化する



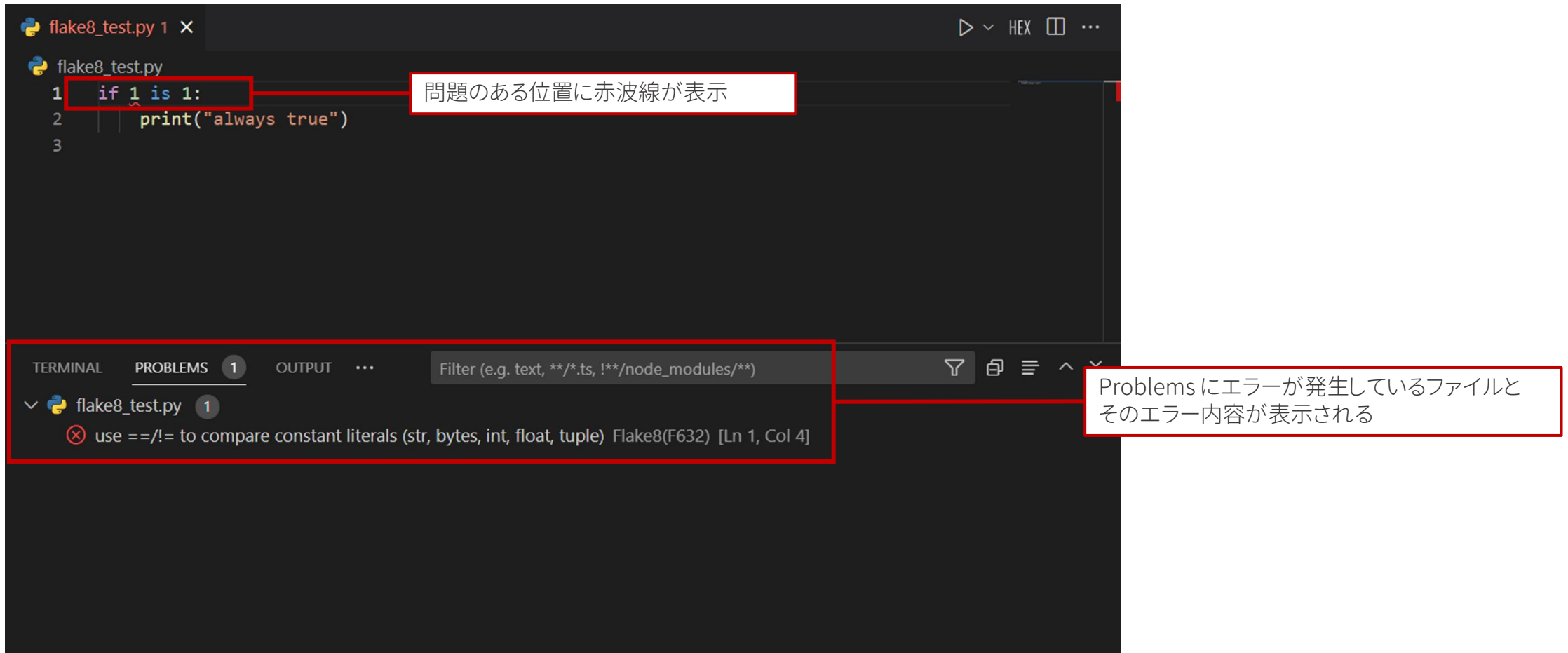
The screenshot shows the VS Code settings interface for the 'Extension: Flake8'. The search bar at the top contains 'python.linting', which is highlighted by a red box. A red line connects this box to a text label 'python.linting を検索'. Below the search bar, the 'Python > Linting > Flake8 Category Severity: W' setting is visible. Further down, the 'Python > Linting: Flake8 Enabled' section is highlighted by a red box, containing a checked checkbox 'Whether to lint Python files using flake8.'. A red line connects this box to a text label 'Flake8 Enabled のチェックをつける'. The left sidebar shows the 'Extensions' view with 'Python (33)' selected.

python.linting を検索

Flake8 Enabled のチェックをつける

5-2. Flake8 の利用 (5)

VSCode上で、コーディングスタイルに違反しているコードに警告が表示されることを確認



まとめ

- チーム開発や大規模開発では、メンバー間での分担や作業の長期化に伴って、バグや不具合が発生しやすくなる
- メンバーの意識や目視・手動での対応は困難なので、ツールを利用することが望ましい
- メンバー間の開発環境の差をなくす: `pyenv`, `poetry`
- 動的型付けのデメリットを解消する: `mypy`
- コードスタイルを統一する: `Black`
- コーディングルールを自動テスト: `Flake8`