

仮想環境とパッケージ 対話環境での入力行編集とヒストリ置換

Python仮想環境の主要なポイント

- **仮想環境は、ベースとなるPythonのバージョンを維持する:**
 - 例えば、あなたがPython12で仮想環境を作成した場合、その仮想環境はPython12の環境として機能し続けます。PCに後からPython13がインストールされたとしても、仮想環境内のPythonバージョンが自動的に13に変わることはありません。これにより、作成時の環境が保護されます。
- **古いバージョンでしか動かないモジュールへの対応:**
 - もしPython10でしか動作しないモジュールを使いたい場合は、まずPCにPython10の実行環境を構築し、そのPython10を指定して仮想環境を作成するのが正しい手順です。
 - 例えば、Python13の環境にPython10用のモジュールを直接インストールしようとしても、互換性の問題から基本的に動作しません。
- **仮想環境の主な目的は「隔離」と「再現性」:**
 - Pythonの標準ライブラリ以外のモジュール（サードパーティ製モジュール）は、それぞれの仮想環境ごとにインストールが必要です。これは、システムにインストールされているPythonのバージョンとは独立して、プロジェクトごとに必要なライブラリのセットを管理するためです。
- **環境を隔離する理由の一つ:**
 - Python自体のバージョンアップのペースと、サードパーティライブラリのアップデートのペースは一致しません。例えば、Pythonのメジャーバージョンは変わっていても、使用しているライブラリは常に新しいバージョンがリリースされ、APIの変更や非推奨化が行われることがあります。
 - この時、もし環境が隔離されていないと、更新されたサードパーティモジュールが以前作成したプログラムと互換性がなく、予期せぬ不具合が発生する可能性があります。
 - （新しく作成されたプログラムは、最新のモジュールに合わせて開発されるため、通常は問題ありません。）
- **PyCharmのようなIDEがプロジェクトごとに仮想環境を自動生成する理由:**

- これはまさに「隔離」を目的としています。各プロジェクトの依存関係の衝突を防ぎ、例えば異なるプロジェクト間でライブラリのバージョン新旧による不具合を回避できます。
- （注: PyCharmのような特定のIDEに関する話は試験範囲外ですが、仮想環境の理解を深める上での参考として役立ちます。）

- **これらのことから、プロジェクトの再現性を保証します:**

- 仮想環境によって、いつ、誰がそのプロジェクトを開いても、開発時と同じ環境で安定して動作させられるようになります。

仮想環境の生成（venvモジュール）

- venvモジュールを使う
 - PyCharmなどに頼らず、自作する方法
 - Windows以外では、細かい部分を調べる必要があるかもしれません
 - 特にコマンドの違いに気をつけるべきです
 - Windowsの方も、問題集などで確認した方が良いと思われます
 - 私は、コマンドが違うので、答が分かりませんでした

生成手順（コマンドプロンプトの場合：補足情報も少しあります）

- Windows以外では、テキストに載っている情報だけ補足します
- もしうまく行かない場合、私の環境との違いなど考えられますので、AIに聞けば解決できると思います

1. コマンドプロンプトから、作成したいフォルダへ移動する

- Windows11環境では、当該フォルダから、右クリックで、カレントフォルダに移動したコマンドプロンプトを開くことが出来る
 - 簡単に当該フォルダから、出来るはずですので、分からない場合はAIに聞くと良いです
 - Windows10でも少しだけややこしいが出来ます

2. 以下のコマンドを実行（learn_list は、フォルダ名）

- `F:\～\test_venv> python -m venv learn_list`
 - Mac/Linuxでは、`python3` ということです

3. フォルダ作成の確認後、当該フォルダ\Scripts\activate.bat とタイプしEnterキーです

- `\` は、Windowsの場合は、英数モードで `円マーク` です
- コマンドプロンプトで、新しくタイプする内容です
- 私の場合：参考にしてください
 - `F:\～\test_venv> learn_list\Scripts\activate.bat`
- Mac/Unixの場合： `source フォルダ名/bin/activate`

4. これが上手くいくと、以下のように表示が変わります

- `(learn_list) F:\~\test_venv>`
 - 成功の場合、(当該フォルダ：私の場合、`learn_list`) が先頭になります

5. 試しに、`pip install pygame` として、pygameモジュールを、pipインストールしてみます

- うまく行くと `Successfully installed pygame-2.6.1` のように表示されます
 - バージョンなどは、違う可能性があります

6. これで、このフォルダ (`learn_list`) を、仮想環境とすることが出来ました

7. 注意1 :

- 後日この環境を使う場合は、再度有効化しなければなりません
 - 当該フォルダからコマンドプロンプトで `Scripts\activate.bat` を実行してください
 - 私の場合： `F:\~\learn_list>Scripts\activate.bat`

● 注意2 : Pythonを複数インストールしていると、期待しているバージョンでない可能性があります

- 私の場合Cドライブ直下に、python12とpython13がインストールされています
 - バージョンの確認方法：コマンドプロンプトで `python --version` で確認できます
 - `--` : ハイフンは2つです
 - 私の場合、python 13 にしたかったのですが、python 12 でした
- 対処方法：もう一度最初から作り直した方が、無難だと思います
 - 少しコマンドプロンプトが重いかもしれませんが、待てば、うまく行くと思います
- a. ここで以下のように `-3.13` などを入れる (フォルダ名は `learn_dict` としました)
 - `F:\~\test_venv>py -3.13 -m venv learn_dict`
- b. そして、有効化する (activate)
 - `F:\~\test_venv>learn_dict\Scripts\activate.bat`
- c. Pythonのバージョンを確認する (私の場合、Python13を期待している)

```
(learn_dict) F:\~\test_venv>python --version
Python 3.13.1
```

d. これで、目的のバージョンにすることが出来ました

仮想環境の詳細

- 以下のコードを実行して、この仮想環境の詳細を調べました
 - ポイント
 - `import sys` および `sys.path`
 - 仮想環境の状態 (詳細は、Geminiに聞きました)
 - 雰囲気だけの理解でよいと思います

以下、Geminiの説明です

Me

公式テキストに載っているようなコードを試すと、以下のようになりました。

`sys`モジュール の、`sys.path` の行っていることは何でしょうか。

```
(learn_dict) F:\~\test_venv\learn_dict>python
Python 3.13.1 (tags/v3.13.1:0671451, Dec 3 2024, 19:06:28)
[MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', 'C:\\Py313\\python313.zip', 'C:\\Py313\\DLLs', 'C:\\Py313\\Lib',
'C:\\Py313', 'F:\\~\\test_venv\\learn_dict',
'F:\\~\\test_venv\\learn_dict\\Lib\\site-packages']
>>>
```

Gemini

`sys.path` の出力は、Pythonの仮想環境の仕組みを理解する上で非常に重要な情報です。

`sys.path` は、
Pythonがモジュール（`import` 文で読み込むファイル）を探しに行くディレクトリのリストです。
Pythonインタープリタが `import some_module` という行を見つけると、
この `sys.path` にリストされているディレクトリを順番に検索し、
`some_module.py` というファイルや
`some_module` という名前のパッケージディレクトリを探します。

あなたの `sys.path` 出力とその意味

あなたの `sys.path` の出力は、
Python 3.13.1の仮想環境内で実行されたものとして、以下のように解釈できます。

```
['',
'C:\\Py313\\python313.zip',          # Pythonの標準ライブラリ（圧縮形式）
'C:\\Py313\\DLLs',                  # PythonのDLLsなど、内部で必要なファイル
'C:\\Py313\\Lib',                   # Pythonの標準ライブラリの大部分
'C:\\Py313',                        # Pythonのインストールディレクトリ自体
'F:\\~\\test_venv\\learn_dict',
                                # 現在の作業ディレクトリ（スクリプトが実行されている場所）
'F:\\~\\test_venv\\learn_dict\\Lib\\site-packages']
```

```
# ★この仮想環境にインストールされた外部モジュール（pipで入れるもの）
# `pip install`はここに入る
#     仮想環境がアクティブである必要がある
#     また、ここにインストールすべき！
#         当該の仮想環境に入れなかった場合、他の所にあっても、
#         Pythonは、パッケージが無いとみなす
```

各要素の解説

1. `''` (空文字列):

- これは通常、
カレントワーキングディレクトリ（現在Pythonスクリプトを実行しているディレクトリ）
を意味します。
Pythonはまず、実行中のスクリプトと同じディレクトリにあるモジュールを探します。

2. `C:\\Py313\\python313.zip`、`C:\\Py313\\DLLs`、 `C:\\Py313\\Lib`、`C:\\Py313` :

- これらは、
仮想環境がベースとしているPythonの標準ライブラリや内部的なファイルのパスです。
- あなたのPCの `C:\\Py313` にインストールされているPython 3.13.1の標準ライブラリ
（`os`、`sys`、`math` など、Pythonが最初から持っている機能）がここから読み込まれます。
- 仮想環境は、システムにインストールされているPythonのコピーではなく、
シンボリックリンクやパスの設定によって、そのPythonインタプリタを利用します。

3. `F:\\~\\test_venv\\learn_dict\\Lib\\site-packages` :

- **これが仮想環境の最も重要な部分の一つです！**★**
- `site-packages` ディレクトリは、**`pip install` コマンドでインストールされたすべての外部ライブラリ（サードパーティモジュール）が格納される場所**です。
- あなたが `pip install pygame` と実行した場合、
`pygame` はこのディレクトリ内にインストールされ、
Pythonはこのパスを検索することで `pygame` を見つけて `import` できるようになります。
- 仮想環境がアクティベートされている場合、
この `site-packages` ディレクトリのパスが `sys.path` に優先的に
（あるいは、その仮想環境専用のパスとして）含まれます。

`sys.path` が教えてくれること

- モジュールの探索順序:** Pythonが `import` 文を実行する際に、
どのディレクトリをどのような順序で探すかを示しています。
通常は、現在のディレクトリ → 仮想環境の `site-packages`
→ ベースのPythonの標準ライブラリ、といった順序です。

- **仮想環境の動作確認:** `sys.path` の中に、
アクティベートしている仮想環境の `Lib\site-packages`
(Mac/Linuxなら `lib/pythonX.X/site-packages`) のパスが含まれていることを確認することで、
その仮想環境が正しく機能しているかを判断できます。
このパスが存在していれば、`pip install` で入れたモジュールは
その仮想環境内からしか利用できません。
- **依存関係の隔離:** `sys.path` に仮想環境固有の `site-packages` が含まれることで、
その仮想環境にインストールされたモジュールは、
システム全体のPython環境や他の仮想環境に影響を与えないことが視覚的に確認できます。

つまり、あなたの `sys.path` の出力は、
`learn_dict` という仮想環境が正しくアクティベートされており、
その仮想環境がベースとしている `C:\Py313` のPython 3.13.1の標準ライブラリと、
仮想環境専用の `site-packages` ディレクトリの両方を利用できる状態にあることを示しています。

pipによるパッケージ管理

- !独断と偏見で、出そうなところをまとめました！
 - pipはどこからインストールするか？
 - Python3.4以降は、インストール時に付いている
 - pipでインストールするのは、主にパッケージである
 - モジュール・パッケージ
 - モジュール：単一のファイル
 - パッケージ：モジュールをまとめたもの
 - ほとんどは、パッケージとしてインストールする
 - pipで始まるコマンド
 - PyPI (Python Package Index) の役割:
 - パッケージが公開されている場所である、という程度の理解。

pip インストール

- ベストプラクティス：仮想環境で行なう(activateする：アクティブにする)
 - そうでないと、仮想環境内にインストールされない
 - PyCharmが、仮想環境を作るのは、ベストプラクティスだからである
- Windows
 - (仮想環境のフォルダ) パス>`python -m pip install パッケージ`

- Mac
 - (仮想環境のフォルダ) パス\$python -m pip install パッケージ
- 補足：ここでも当然、仮想環境で行うことが前提である
 - == (バージョン指定子)で、パッケージのバージョンを指定する
 - 例：python -m pip install requests == 2.6.0
 - requests：パッケージ名
 - --upgrade オプションで、当該パッケージを、最新バージョンに出来る
 - 例：python -m pip install --upgrade requests
 - アンインストールする時：install ➡ uninstall で、OK
 - スペース で区切って、複数のパッケージに対して実行できる
 - 例：python -m pip uninstall numpy pygame
 - numpy, pygame：パッケージ名

以下の pip コマンド は、Windows も Mac も共通でした

- 仮想環境に生成、アクティブ化の場合は、違う部分に気をつける
- pip 関連に関しては、コンソール画面の違いはあるが、試験範囲では、同じだと思われる
- 当然以下も、仮想環境をアクティブにして 実行すべきである

show：パッケージの情報を表示する

- pip show パッケージ名

list：仮想環境にインストールされた全てのパッケージを表示する

- pip list

freeze と requirements.txt (Geminiに尋ねました)

- ポイント
 - pip freeze
 - pip list も、インストールされているパッケージの一覧を表示しますが、その形式は人間が読むのに適しています。
対して、pip freeze は、pip install 向けの形式を出力 する
 - pip freeze > requirements.txt
 - 出力するリストを、テキストファイルに出力している

- requirements.txt という名前は、ルールに近い慣習である
- *1 ○ ~~cat requirements~~: requirements.txtの内容を表示する *1
cat requirements.txt
(拡張子 .txt が抜けていました)
追記: Windowsでは、
type requirements.txt
です。
- pip install -r requirements.txt
 - requirements.txt に記載されたパッケージをすべて一括で、
指定されたバージョンでインストールします。

by Gemini

公式テキストの pip freeze と requirements.txt に関する部分ですね。

Pythonプロジェクトの依存関係管理において非常に重要な概念です。詳しく解説します。

pip freeze と requirements.txt の役割

このセクションは、

Pythonプロジェクトで「どのパッケージの、どのバージョンを使っているか」を正確に記録し、その情報を他の人（または将来の自分）と共有するための仕組みを説明しています。

1. pip freeze の説明

pip freezeは pip listと同様にインストールされたパッケージのリストを作るものだが、こちらはpip install向けの形式を出力する。

pip list も現在インストールされているパッケージの一覧を表示しますが、その形式は人間が読むのに適しています。

「pip install向けの形式を出力する」：ここが pip freeze の肝 です。

pip freeze は、単にパッケージ名を表示するだけでなく、

pip install コマンドでそのまま使える形式で出力します。

具体的には、パッケージ名==バージョン番号 の形式です。

例：requests==2.7.0

2. pip freeze > requirements.txt

出力されるリストをrequirements.txtにセーブするのが一般的な使い方である。

(仮想環境のフォルダ) ... pip freeze > requirements.txt

> の意味：これはLinuxやmacOSのシェル（ターミナル）や Windowsのコマンドプロンプトで使われる「リダイレクト」という機能です。

pip freeze が通常画面に出力する内容を、画面に出力する代わりに、の後に指定されたファイル (requirements.txt) に書き込みます。

なぜこのファイル名？： requirements.txt というファイル名は、

Pythonコミュニティにおける事実上の標準です。

この名前のファイルを見たら、

「このプロジェクトが必要とするパッケージのリスト」だと誰もが理解します。

具体的な例：

このコマンドを実行すると、requirements.txt というファイルが作成（または上書き）され、その中には以下のような内容が書き込まれます。

```
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

これは、このプロジェクトが novas のバージョン 3.1.1.3、
numpy のバージョン 1.9.2、
requests のバージョン 2.7.0 に依存していることを示します。

3. requirements.txt の用途

requirements.txt はバージョンコントロール向けにコミットしたり、
アプリケーションとともに配布するのに使える。

バージョンコントロール向けにコミット：

Git などのバージョン管理システムに requirements.txt を含めてコミットすることで、
プロジェクトの依存関係をコードと一緒に管理できます。

これにより、requirements.txt があれば、たとえプログラムの修正中に問題が起きても、
そのプログラムが正しく動いていた時の

『土台となる部品（外部パッケージ）の組み合わせ』を簡単に再現できます。

これで、プログラムがエラーを出してしまった場合でも、
以前の動く状態に戻して原因を探しやすくなります。

アプリケーションとともに配布：

プロジェクトを他の開発者に渡す際や、アプリケーションをデプロイする際、
このファイルを含めることで、受け取った側は簡単に必要な環境を構築できます。

4. pip install -r requirements.txt

ユーザー側では install -r で必要なパッケージをすべてインストールできる。

(仮想環境のフォルダ) ... python -m pip install -r requirements.txt

-r オプションの意味：

pip install の -r または --requirement (省略されていない書き方) オプションは、

「指定されたファイルに記載されているパッケージをすべてインストールする」という指示です。

これにより、requirements.txt に書かれたすべてのパッケージが、
指定された厳密なバージョンで一括してインストールされます。

再現性の確保：これが最も重要な点です。

もしプロジェクト A が requests==2.7.0 に依存しているのに、
別の開発者が requests の最新版（例えば 3.0.0）をインストールしてしまったら、
互換性の問題でコードが動かなくなる可能性があります。

requirements.txt と pip install -r を使うことで、
どの環境でも全く同じパッケージの組み合わせを再現できるため、
開発環境のズレによるバグを防ぎ、チーム開発やデプロイをスムーズにします。

まとめ

pip freeze：現在インストールされているパッケージとその正確なバージョンを、pip install で読み込み可能な形式で出力します。 pip freeze > requirements.txt

requirements.txt：

pip freeze の出力を保存する、プロジェクトの依存関係リストファイルです。
この名前が標準です。

pip install -r requirements.txt：

requirements.txt に記載されたパッケージをすべて一括で、
指定されたバージョンでインストールします。

この一連の流れは、

Pythonプロジェクトの「再現性」と「共同開発の効率化」に不可欠な仕組みです。

対話環境(インタプリタ)について

- IDLEやコマンドプロンプトで python とタイプすると >>> が出てくる環境のこと
- ポイント

1. タブ補完機能の存在とその利用 ([Tab] キー)

- 「Tabキーで補完が出来る」というのは
今まで宣言された変数などの候補を出してくれる ということ

2. ヒストリ機能の存在と利用 (.python_history ファイル)

- 「入力履歴が保存されており再利用できる」というのは、
↑ キーで、前に打ったコマンドを呼び出せること

3. 拡張された対話型インタプリタの存在 (bpython, IPython)

- 普通のインタプリタ < bpython < IPython という関係、
 - この中で IPythonが最も高機能な対話型インタプリタ だということ

END