Python実行環境の話

K. Yamaguchi

実行環境とは?

実行環境とは?

- 例えば
  - macOS
  - pyenv でインストール（ビルド）された Python

```
/Users/hogehoge/.pyenv/versions/3.12.6 % tree -L 2 -F
./
├── bin/
│   ├── 2to3 -> 2to3-3.12*
│   ├── 2to3-3.12*
│   ├── idle -> idle3.12*
│   ├── idle3 -> idle3.12*
│   ├── idle3.12*
│   ├── pip -> pip3.12*
│   ├── pip3*
│   ├── pip3.12*
│   ├── pydoc -> pydoc3.12*
│   ├── pydoc3 -> pydoc3.12*
│   ├── pydoc3.12*
│   ├── python -> python3.12*
│   ├── python-config -> python3.12-config*
│   ├── python3 -> python3.12*
│   ├── python3-config -> python3.12-config*
│   ├── python3.12*
│   ├── python3.12-config*
│   └── python3.12-gdb.py*
├── include/
│   └── python3.12/
├── lib/
│   ├── libpython3.12.dylib*
│   ├── pkgconfig/
│   └── python3.12/
└── share/
    └── man/
```

実行環境とは?

- 例えば
  - macOS
  - uv でダウンロードされた
    Python Standalone Builds

```
/Users/hogehoge/.local/share/uv/python/cpython-
3.12.6-macos-aarch64-none
% tree -L 2 -F
./
├── bin/
│   ├── 2to3 -> 2to3-3.12*
│   ├── 2to3-3.12*
│   ├── idle3 -> idle3.12*
│   ├── idle3.12*
│   ├── pip*
│   ├── pip3*
│   ├── pip3.12*
│   ├── pydoc3 -> pydoc3.12*
│   ├── pydoc3.12*
│   ├── python -> python3.12*
│   ├── python3 -> python3.12*
│   ├── python3-config -> python3.12-config*
│   ├── python3.12*
│   └── python3.12-config*
├── include/
│   └── python3.12/
├── lib/
│   ├── itcl4.2.2/
│   ├── libpython3.12.dylib*
│   ├── pkgconfig/
│   ├── python3.12/
│   ├── tcl8/
│   ├── tcl8.6/
│   ├── thread2.8.7/
│   └── tk8.6/
└── share/
    └── man/
```

tcl とかが入ってくるのは
Standalone たるゆえん

実行環境とは?

- 例えば
  - Ubuntu
  - pyenv でインストール（ビルド）された Python

```
/home/hogehoge/.pyenv/versions/3.12.6 % tree -L 2 -F
./
├── bin/
│   ├── 2to3 -> 2to3-3.12*
│   ├── 2to3-3.12*
│   ├── idle -> idle3.12*
│   ├── idle3 -> idle3.12*
│   ├── idle3.12*
│   ├── pip -> pip3.12*
│   ├── pip3*
│   ├── pip3.12*
│   ├── pydoc -> pydoc3.12*
│   ├── pydoc3 -> pydoc3.12*
│   ├── pydoc3.12*
│   ├── python -> python3.12*
│   ├── python-config -> python3.12-config*
│   ├── python3 -> python3.12*
│   ├── python3-config -> python3.12-config*
│   ├── python3.12*
│   ├── python3.12-config*
│   └── python3.12-gdb.py*
├── include/
│   └── python3.12/
├── lib/
│   ├── libpython3.12.so -> libpython3.12.so.1.0*
│   ├── libpython3.12.so.1.0*
│   ├── libpython3.so*
│   ├── pkgconfig/
│   └── python3.12/
└── share/
    └── man/
```

実行環境とは?

- 例えば
  - Debian Linux bullseye
  - apt install python3 python3-pip
    pip3 install requests
    だけした時のシステムのPython

```
/# tree -F /usr/bin/
/usr/bin/
├── X11 -> ./
(略)
├── pip*
├── pip3*
(略)
├── py3clean*
├── py3compile*
├── py3versions -> ../share/python3/py3versions.py*
├── pydoc3 -> pydoc3.9*
├── pydoc3.9*
├── pygettext3 -> pygettext3.9*
├── pygettext3.9*
├── python3 -> python3.9*
├── python3-config -> python3.9-config*
├── python3.9*
├── python3.9-config -> aarch64-linux-gnu-python3.9-config*
(略)
└── zipinfo*

/# ls -ld /usr/include/python3.9/ /usr/lib/python3*/
drwxr-xr-x 4 root root  4096 Dec  5 14:26 /usr/include/python3.9/
drwxr-xr-x 1 root root 20480 Dec  5 14:26 /usr/lib/python3.9/
drwxr-xr-x 1 root root  4096 Oct 19 05:17 /usr/lib/python3/
```

この話はしません

```
/# ls /usr/lib/python3/dist-packages/
_distutils_hack  hgext3rd                pip                 setuptools                wheel-0.34.2.egg-info
hgdemandimport   mercurial               pip-20.3.4.egg-info  setuptools-52.0.0.egg-info
hgext            mercurial-5.6.1.egg-info  pkg_resources       wheel
/# ls /usr/local/lib/python3.9/dist-packages/
certifi                   charset_normalizer-3.4.0.dist-info  requests                urllib3-2.2.3.dist-info
certifi-2024.8.30.dist-info  idna                              requests-2.32.3.dist-info
charset_normalizer        idna-3.10.dist-info               urllib3
```

Un*xソフトウェアのふつうの構成

- 例えば
  - ./configure --prefix=$HOME/mecab でビルドした mecab
  - /usr /usr/local にも同じ構造が現れる

```
/home/hogehoge/mecab % tree -L 2 -F
./
├── bin/
│   ├── mecab*
│   └── mecab-config*
├── etc/
│   └── mecabrc
├── include/
│   └── mecab.h
├── lib/
│   ├── libmecab.a
│   ├── libmecab.la*
│   ├── libmecab.so -> libmecab.so.2.0.0*
│   ├── libmecab.so.2 -> libmecab.so.2.0.0*
│   ├── libmecab.so.2.0.0*
│   └── mecab/
├── libexec/
│   └── mecab/
└── share/
    └── man/
```

Windows の
話はしません

Pythonの基本的な構造

基本的な構造

- (どこか)/bin
  - コマンドラインインタフェース（=実行バイナリとスクリプト）
- (どこか)/lib
  - OS の共有ライブラリ
- (どこか)/lib/python3.X
  - 標準ライブラリ
- (どこか)/lib/python3.X/site-packages
  - pip のインストール先

以降
"実行ファイル"

Mac の dylib
Linux の so

Pythonコンパイラ
と仮想マシンの
本体はこれ

pipで --user を指定したり
するとまた別の話

LinuxのシステムのPython
だとまた別の話

siteモジュールが適切に有効に
なっていると、という話

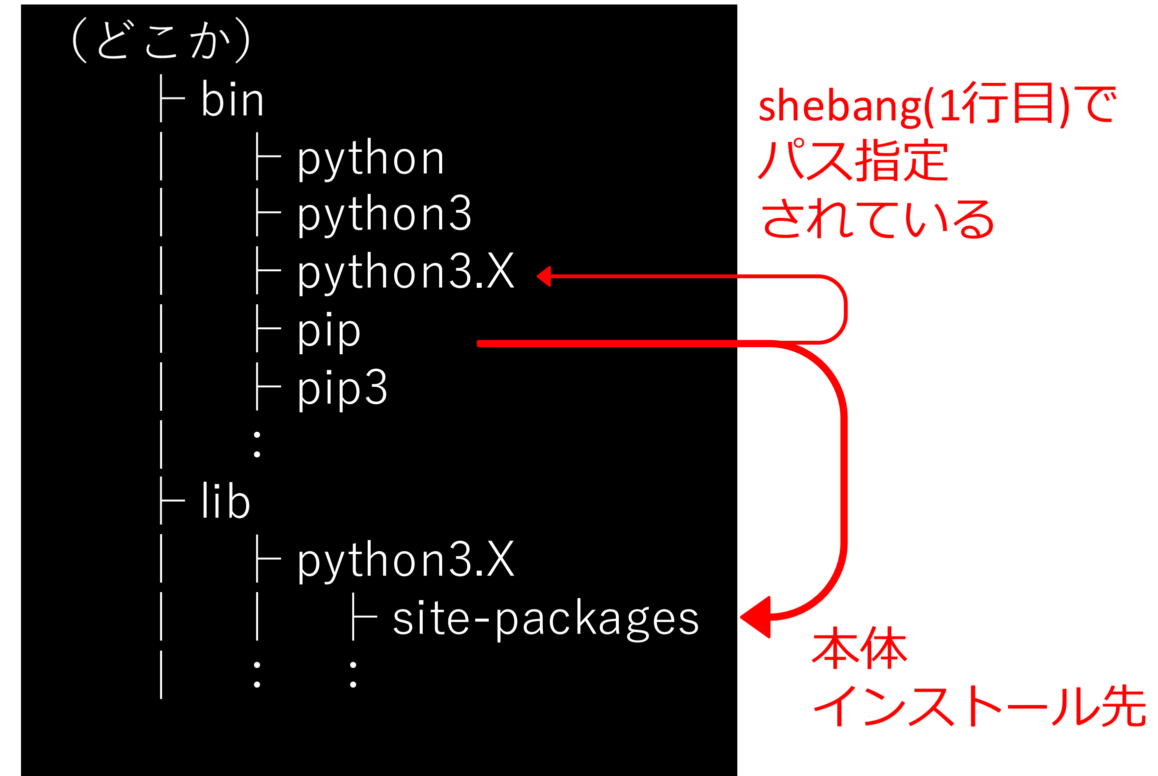
基本的な構造（普段気にしない方）

- (どこか)/include
  - pip インストールで C言語のビルドがある時に使われる（多分）

- (どこか)/share/man
  - マニュアル

基本的な構造

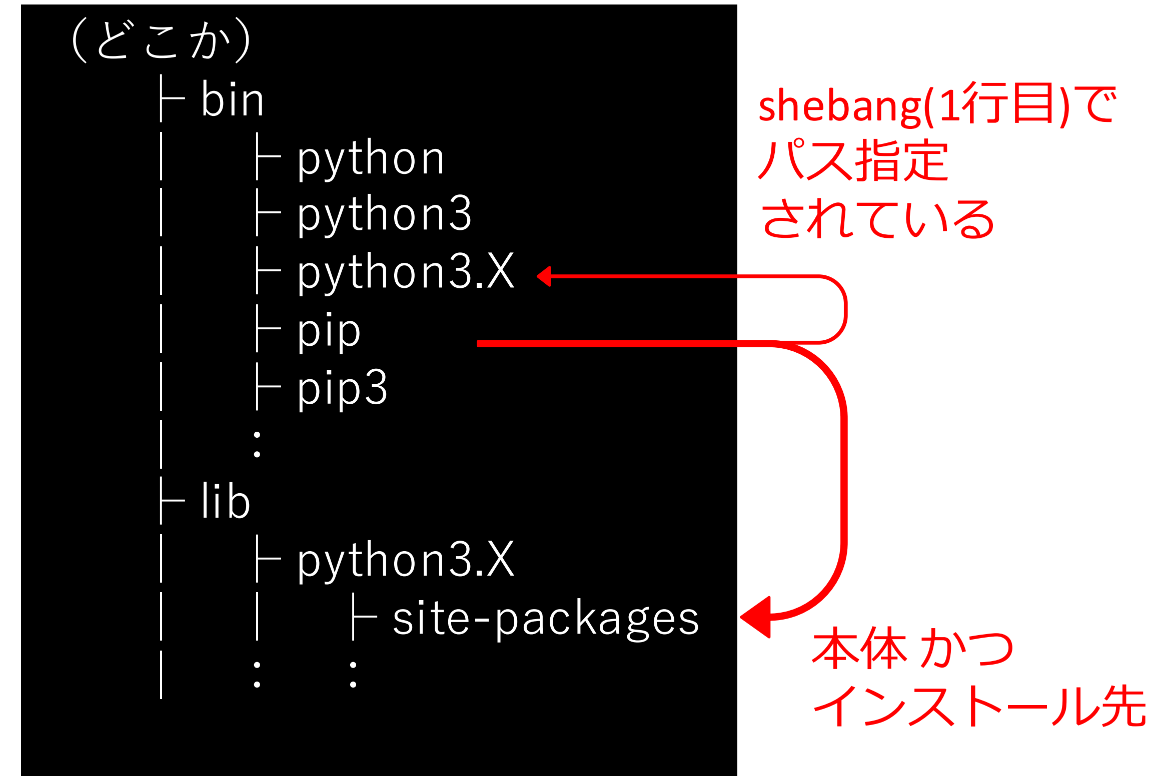
- ふだん意識する場所はここだけ →
  - bin と site-packages 以外はインストール後は基本的に不変なので
- python3.X が実行バイナリ
  - python3 python はシンボリックリンク
- pip pip3 はスクリプト
  - shebang で python3.X と紐付いている
  - 処理の本体は Python ライブラリ lib/python3.X/site-packages/pip の中
  - 一般ユーザが動かすと (どこか)/lib/python3.X/site-packages と (どこか)/bin がインストール先になる



Un*x系の場合 (Windowsはちょっと違う)

基本的な構造

- (どこか)とはどこなのか?
  - Un*x系のシステム用 だと /usr /usr/local
  - pyenv だと ${PYENV_ROOT}/versions の下にそれぞれ
  - uv だと ${UV_PYTHON_INSTALL_DIR} の下にそれぞれ
  - Mac 用の Cpython 公式インストーラを使うと /Library/Frameworks/Python.framework/Versions/3.X
  - Mac Homebrew だと /opt/homebrew/Cellar/python@3.X/3.X.Y/Frameworks/Python.framework/Versions/3.X



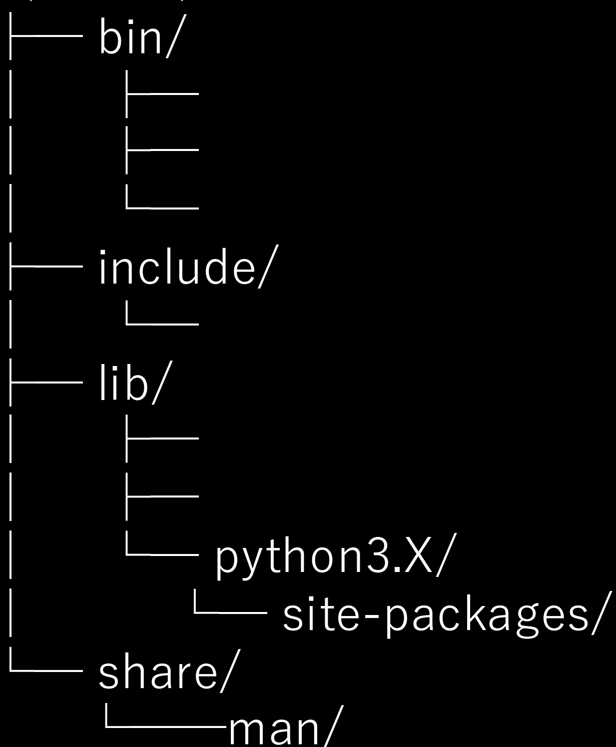
Un*x系の場合 (Windowsはちょっと違う)

その(どこか)
と
その中の共通的な構造
のことを
私たちは普段
「Python環境」と呼んでいる

個人の感想です

仮想環境

まずフルセットの環境

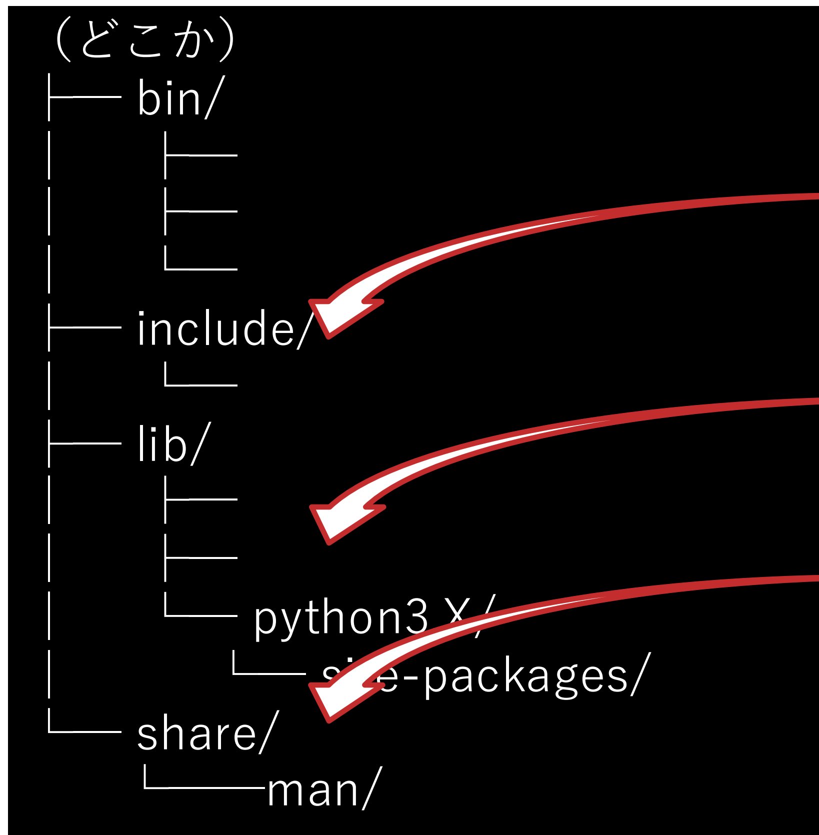


- bin と site-packages 以外はインストール後は基本的に不変なので

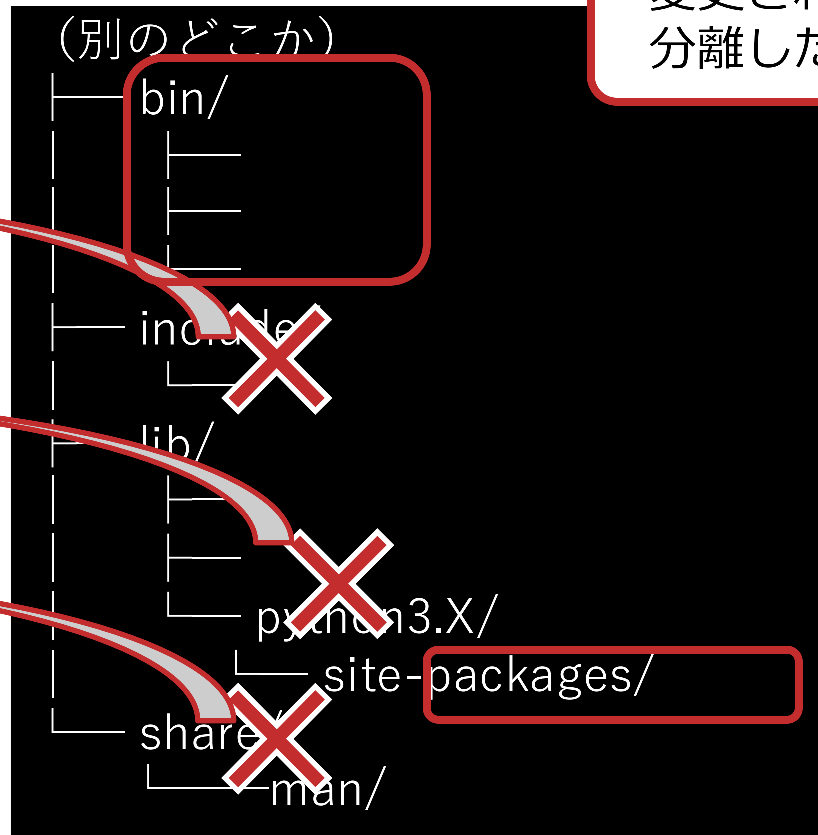
仮想環境とは

- 変更される bin と site-packages だけを別の場所に分離しよう = 仮想環境

フルセットの環境



仮想環境



変更されるところ
分離したいところ

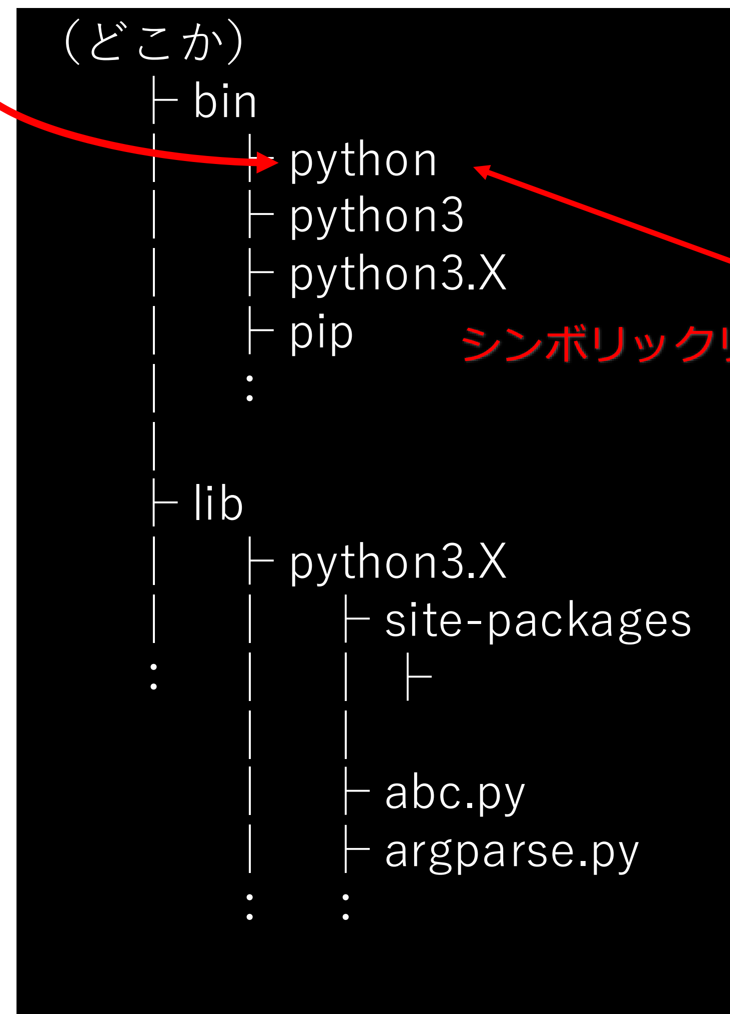
書き換わらないものはどうせ同じなのだから
フルセットの環境を読めばいい

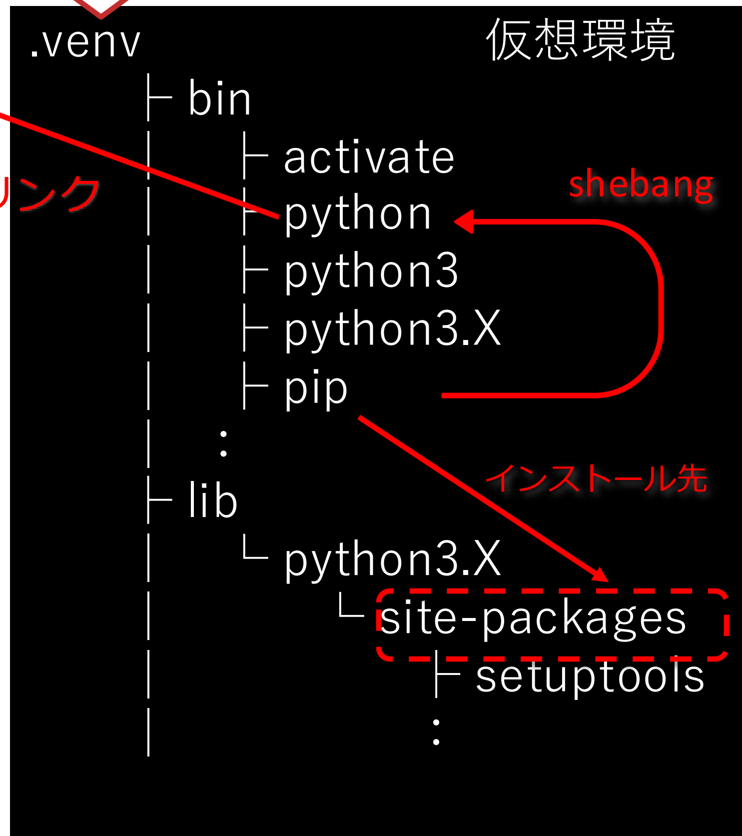
仮想環境の中身

- 以下のものが作られる
  - 隔離された site-packages
  - 実行ファイル（リンク）
  - 仮想環境を使うツール
    - activate スクリプト
      - 仮想環境の bin を優先させる
      - プロンプトに仮想環境名が出る
    - deactivate シェル関数で元に戻す
    - これらはおまけ
- venv でも virtualenv でも
  pipenv でも
  pyenv-virtualenv でも
  conda でも、仮想環境と言ったら
  「bin と site-packages の分離」
  という動機は共通

(例)
python -m venv .venv

作成





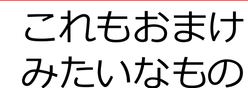
pyenv

pyenv

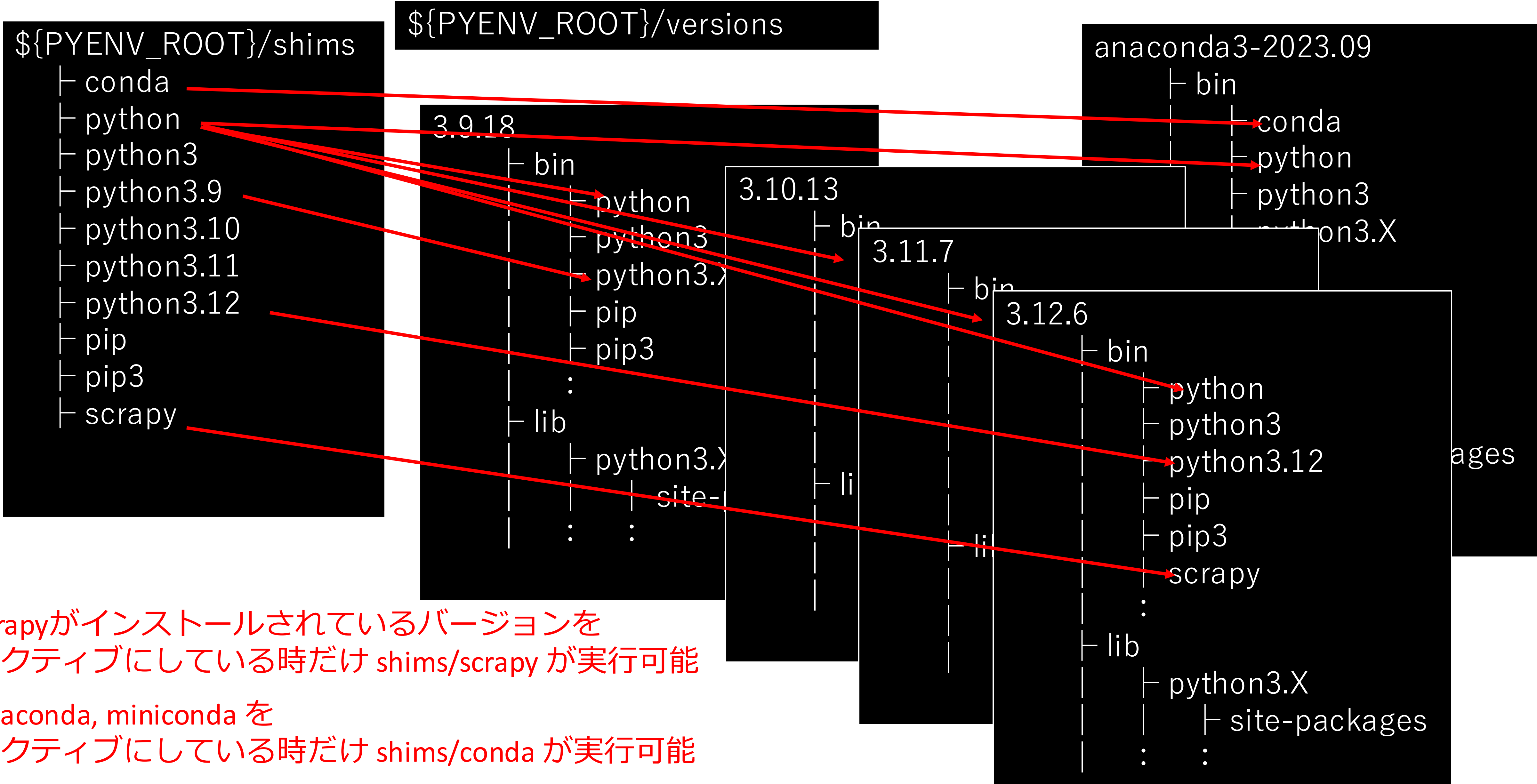
- pyenv は仮想環境ではない
- 複数の環境を一つのディレクトリの下で管理するもの
  - 3.12系と3.13系の同居とか、CPython と PyPy の同居 とかが目的
- 指定したバージョンを自動でインストールしてくれるのが便利
  - ソースをダウンロードして自動ビルド
  - インストーラをダウンロードして自動インストール

pyenv

- 複数環境に存在する実行ファイルへのエイリアスを管理する
  - shims 配下の実行ファイル
  - 実行すると、アクティブな環境の実行ファイルに処理を委譲する
    - その分ちょっとだけオーバーヘッドがある



pyenv



pyenv

- “アクティブ”とは? = shims の実行ファイルはどれを起動するの?
  - 環境変数 PYENV_VERSION があるならそれ
    - pyenv shell ~ でセットされる → そのシェルを閉じる（か pyenv shell --unset を実行する）まで最優先になる

私はよく使います

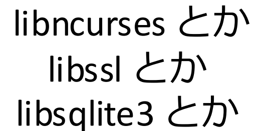
  - .python-version というファイルがカレントディレクトリにあるならそれ
    - pyenv local ~ でカレントディレクトリに .python-version ファイルができる
  - カレントディレクトリから親へ親へと .python-version を探して見つかったらそれ
  - ${PYENV_ROOT}/version というファイルがあるならそれ
    - pyenv global ~ でファイルができる
    - 特に設定しなければ ~/.pyenv/

（みつからないならsystem）

Anaconda

Anaconda

- Anaconda は Python のディストリビューション（配布形態）の一つ
- 独自のパッケージレポジトリを持つ ← conda コマンドで管理する
  - CPython の各種 OS 向けコンパイル済みバイナリ
    - OS 側ライブラリのコンパイル済みバイナリ
  - 各種 Python ライブラリの各種 OS 向けコンパイル済みバイナリ



- pip (https://pypi.org/) とは別の依存関係を持っている
  - conda と pip は混ぜて使わない
  - Aというパッケージが、「pip の B」と「conda の C」から要求されていて、要求バージョンが違う時
    - pip install B → conda install C の順に実行するか、 逆順に実行するかで A のインストール元とバージョンが変わる
  - 依存関係でトラブルが起きると、conda からも pip からも自動解決できない

  - （重要）conda で入るものは可能な限り conda で入れてから、どうしても入らないものだけ pip で入れる

昔語り

- numpy のようなライブラリを入れる時、C言語のビルド環境と知識が必要だった
  - pip install numpy するとソースのダウンロードとC言語のビルドが始まる（みたいなイメージ）

- Anaconda はコンパイル済みバイナリを配布しているのが便利だった
  - Windows 環境では普通には Cコンパイラが入っていないので特に便利

- 今は https://pypi.org/ で、OS ごと CPUアーキテクチャごと Python バージョンごとのバイナリ提供が広くおこなわれるようになった
  - Windows でも Mac でもまず困らない
  - マイナな CPUアーキテクチャの Linux は今も事情は変わらない
  - Mac の CPUアーキテクチャが新しくなった時は大変だった

- 今は Anaconda でなく CPython で環境を使っても困らない

uv

uv

- フルセットの実行環境が${UV_PYTHON_INSTALL_DIR}の下にそれぞれ入る
  （ドキュメントに記載ない……。普通は ${HOME}/.local/share/uv/python）
  - uv python install で Python Standalone Builds のバイナリが入る
- そしてそこの環境はほぼ意識しない
  - そこの中が変更されることはないから
  - 要求された実行環境がなければ勝手に入るから
  - （Homebrew 配下の Python が使われたりするので、”どこを使っているか?” は意識した方はよい）

UV_BREAK_SYSTEM_PACKAGES
を意識してセットしない限り(?)

- uv python pin でカレントに .python-version が作られる
- uv init で pyproject.toml などが作られる

uv

- uv venv でカレントの .venv に仮想環境を作る

- uv add で仮想環境の site-packages に何を入れるか? を決める
  - 指定したものが pyproject.toml に書かれる（これが制約）
  - 制約に対して依存性を解決して実際に入ったものが uv.lock に書かれる

- pyproject.toml と uv.lock があれば uv sync で site-packages を復元できる
  - ここが高速（Rust製であることと優秀なキャッシュ）

これはPoetryもそう

- uv run で仮想環境の実行ファイルを実行する
  - uv run python hoge.py が基本になる

.venv/bin/activate
を使うのは自由

uv

- uv 自体は Python を必要としない
  - uv は Python 実行環境を管理する側、Python は uv で管理される側

- uv の仮想環境には bin/pip がない
  - そもそも uv は pip を置き換える存在
  - uv add/uv remove でやろう
  - （uv pip でインストールはできるけれど uv sync で元に戻される）

uvx

- uvx pdf2sb で実行できる
  - ${UV_CACHE_DIR}/archive-v0/ (なにかユニークな仮想環境) が勝手にできて
  - ${UV_CACHE_DIR}/archive-v0/ (なにかユニークな仮想環境) /bin に pdf2sbが入る
  - この時 pdf2sb のための仮想環境を意識してない
  - （作られた仮想環境は放っておいてよいらしい。uv clean などでクリーンアップされる?）

- uv tool install pdf2sb とかすると
  - ${HOME}/.local/share/uv/tools/pdf2sb/ ができる
  - ${HOME}/.local/share/uv/tools/pdf2sb/bin に pdf2sbが入る
  - ${HOME}/.local/bin にシンボリックリンクができる
  - uvx pdf2sb で起動できる
  - ${HOME}/.local/bin にパスを通していたら pdf2sb で起動できる

まだちょっと
よくわかってない

uvx

- PyPI に実行ファイルをインストールするパッケージがある
  - pdf2sb とか tqdm とか
- Homebrew みたいな使われ方もされるかも

入らなかったこと

- requirements.txt というものがあってね……
- pyenv でビルドした Python は OS の共有ライブラリに依存してるから
  OS がバージョンアップすると動かなくなる……
- PyPy とか Jython とか
- OS 共有ライブラリと Python ライブラリ（バインディング）の関係
- Python Standalone Builds とは