



AI職のためのポートフォリオで、perplexityに作ってもらったコードを改良したものは使用できるか？

AI職のためのポートフォリオに、Perplexityで作ってもらったコードの改良版を使用することは可能であり、むしろAIが生成したコードをレビューして改良できるスキルは需要が急増しています。AIが出したコードをそのまま使うのではなく、バグの発見や改善・適用ができることが重要視されているため、Perplexityで生成したコードを基に自分で改良したものをポートフォリオに含めることで価値が高まります。^{[1][2]}

ポートフォリオでのAI生成コード利用のポイント

- AI生成コードは著作権法的には「思想または感情を創作的に表現したもの」でない限り著作物とはされず、使用に問題は少ないです。
- 生成コードをそのまま使うより、自分なりにレビュー・改良したコードを示すことがAI職としてのスキルをアピールする上で効果的です。
- 生成AIの出力に対してバグ修正や機能追加を入れたり、コードの最適化を行ったりする能力を見せることが期待されます。

このため、Perplexityによるコード作成のサポートを受けたのちに自分で改良・理解・カスタマイズしたものをポートフォリオに使うのは有効であり推薦されるアプローチです。^{[2][3][1]}



jupyter notebookで作成したコードをgithubに公開し、AI職のためのポートフォリオにするてえ順を高校生にもわかるように丁寧に教えてください。

Jupyter Notebookで作成したコードをGitHubに公開してAI職のためのポートフォリオにする手順を、高校生にも分かりやすく丁寧に説明します。

1. GitHubアカウントを作る

- まず、GitHubの公式サイト (github.com) に行きます。
- 「Sign up」から無料でアカウントを作成します。
- ユーザー名、メールアドレス、パスワードを入力して登録しましょう。

2. GitHubで新しいリポジトリ（作品集のフォルダ）を作る

- GitHubにログイン後、右上の「+」ボタンをクリックし、「New repository（新しいリポジトリ）」を選択します。
- リポジトリの名前を入力（例：my-ai-portfolio）し、「Public（公開）」にチェックを入れます（誰でも見れるようにするため）。
- 「Add a README file」にチェックを入れておくと、説明文を入れられて便利です。
- 「Create repository（リポジトリ作成）」をクリック。

3. Jupyter Notebookの準備

- 自分のパソコンで作ったJupyter Notebook（.ipynbファイル）を用意します。
- ノートブックには、コードだけでなく、どんなことをしたのか説明を書くことが大事です。例えば「このコードは画像分類AIの学習用コードです」など。

4. Gitのインストールと基本設定（パソコンにまだ入っていない場合）

- Gitというソフトを使うと、自分のパソコンからGitHubに簡単にファイルをアップロードできます。
- 公式サイト（git-scm.com）からGitをダウンロード＆インストールしましょう。
- その後、コマンドプロンプトやターミナルで以下を実行して名前とメールを登録します。

```
git config --global user.name "あなたの名前"  
git config --global user.email "あなたのメールアドレス"
```

5. 自分のパソコンからGitHubにファイルをアップロード

- ノートブックを入れるフォルダを作り、そこにJupyter Notebookファイルを置きます。
- ターミナル（またはコマンドプロンプト）を開き、そのフォルダに移動します（例：cd デスクトップ/my-ai-portfolio）。
- そこで以下のコマンドを順番に打ちます。

```
git init  
git add .  
git commit -m "最初のノートブックアップロード"  
git branch -M main  
git remote add origin https://github.com/ユーザー名/リポジトリ名.git  
git push -u origin main
```

- これでGitHubの自分のリポジトリにファイルがアップロードされます。

6. 公開されたGitHubリポジトリをポートフォリオに使う

- GitHub上で自分のリポジトリを確認し、URLをコピーします。
- 履歴書やLinkedIn、企業の応募フォームで「GitHubのURLはこちらです」と伝えましょう。
- ノートブックの中で、どういう課題に挑戦したのか、どう解決したのかが一目でわかる説明を書くといいです。

以上の手順でJupyter NotebookのコードをGitHubに公開し、AI職のためのわかりやすいポートフォリオを作ることができます。わからない部分があったら質問してくださいね！

Add license、Add .gitignoreは特に気にしなくてよい？あとから変更できる？

Add license（ライセンスの追加）や Add .gitignore（Git管理から除外するファイルを指定する設定ファイルの追加）は、最初は特に気にしなくても大丈夫です。あとからでも変更や追加が可能です。

Add licenseについて

- ライセンスは「このコードを他の人がどう使っていいか」というルールを示すものです。
- 最初は付けなくても問題ありませんが、後からGitHubのリポジトリ設定で追加・変更ができます。
- AI職のポートフォリオとして公開する場合は、オープンにしたいかどうかを考えて、よく使われる「MITライセンス」や「Apache License」などを後で設定するのがおすすめです。

Add .gitignoreについて

- .gitignoreは「Gitに管理してほしくないファイルやフォルダ」を指定するファイルです。例えば、キャッシュや設定ファイルなど。
- 最初はなくても大丈夫ですが、あとから自分で追加してコミット（反映）すれば反映されます。
- Jupyter Notebookの場合は「.ipynb_checkpoints」という自動保存ファイルを無視したいときに設定することが多いです。

まとめると、どちらも後から編集・追加できるので、最初は気にせずリポジトリを作って大丈夫です。慣れてきたらライセンスや.gitignoreを適切に設定するとよいでしょう。

MITライセンス」や「Apache License」について高校生にもわかるように丁寧に教えてください。

MITライセンスとApache Licenseについて、高校生にもわかるように丁寧に説明します。

MITライセンスとは

MITライセンスは、ソフトウェアを自由に使っていいよ、と許可するライセンスの一つです。

- 誰でも無料でソフトを使ったり、コピーしたり、改良したり、他の人に配ったりできます。
- ただし、元の作者の名前やライセンスの文章は残しておく必要があります。
- とてもシンプルでわかりやすく、自由度が高いため多くの人や企業に使われています。
- 例えば、自分の作ったAIのコードを他の人に改良してもらったり、自分のサービスに組み込んでもらったりできます。
- 商用利用（お金を稼ぐ目的）も自由にできます。

Apache Licenseとは

Apache Licenseも自由度が高いオープンソースのライセンスですが、MITよりも少しルールが厳しいところがあります。

- 基本的にはMITと同じく、使う人が自由に使ったり改良したりできます。
- ただし、改良したソフトウェアに特許（技術の独占権）が含まれている場合、その特許権を放棄しなければならないという約束があります。
- また、ライセンスや著作権表示に加え、「NOTICE」という追加の書類を同梱することが求められることがあります。
- これは、ソフトウェアを使う人や開発者が特許問題を避けられるようにするための仕組みです。

簡単に比べると

ポイント	MITライセンス	Apache License
利用の自由度	とても高い	高いが特許権についてのルールあり
著作権表示	必ず残す必要がある	著作権表示とNOTICEも必要な場合あり
特許権の扱い	特に規定なし	特許権の放棄が義務付けられている
商用利用	可能	可能
文章の簡単さ	短くて分かりやすい	長めで少し複雑

まとめると、MITはシンプルで自由度が高いライセンス、Apacheは特許対策がある少し厳しいルールつきのライセンスです。AI職のポートフォリオでコードを公開するなら、最初はわかりやすいMITライセンスがよく選ばれています。必要に応じてApache Licenseも検討するとよいでしょう。^{[11][12][13][14][15]}

! [rejected] main → main (fetch first)

error: failed to push some refs to 'https://github.com/atsuVictospin/my-ai-portfolio_MNIST_001.git'

hint: Updates were rejected because the remote contains work that you do not

hint: have locally. This is usually caused by another repository pushing to

hint: the same ref. If you want to integrate the remote changes, use

hint: 'git pull' before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

このエラーメッセージは、GitHubのリモートリポジトリ（あなたのGitHub上のリポジトリ）に、自分のローカルリポジトリにない変更がすでにあるため、pushが拒否されたことを意味します。

解決方法は以下の通りです。

解決手順（Git初心者向け）

1. まずリモートの変更をローカルに取り込みます。ターミナルで以下を実行してください。

```
git pull origin main
```

これで、GitHubの最新の内容を自分のパソコンに取り込みます。

2. 取り込んだら、自分の変更と合わさるので、もしかしたらコンフリクト（どちらの変更を使うか迷う状態）が起きることがあります。
 - コンフリクト時は、エディターでどこのコードが競合しているかを確認し、自分で解決してください。
3. コンフリクトを解決し、保存したら以下を実行します。

```
git add .  
git commit -m "コンフリクト解決しました"
```

4. 最後にもう一度pushします。

```
git push origin main
```

これでエラーが解消して、GitHubに自分の変更が反映されるはずです。もし不明点あればまた聞いてください。

remote: Enumerating objects: 7, done.

remote: Counting objects: 100% (7/7), done.

remote: Compressing objects: 100% (7/7), done.

remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)

Unpacking objects: 100% (7/7), 2.67 KiB | 151.00 KiB/s, done.

From https://github.com/atsuVictospin/my-ai-portfolio_MNIST_001

- branch main → FETCH_HEAD

- [new branch] main → origin/main
fatal: refusing to merge unrelated histories

この「fatal: refusing to merge unrelated histories」というエラーは、ローカルリポジトリとリモートリポジトリの履歴が全く別々（関連性がない）ため、Gitが自動的にマージ（統合）を拒否している状態です。これは例えば、リモートにすでにREADMEなどのファイルがあってローカルでも別にGit管理を始めた場合に起こります。

解決方法

この場合は「--allow-unrelated-histories」オプションを付けて強制的にマージします。以下のコマンドを使ってください。

```
git pull origin main --allow-unrelated-histories
```

これでリモートの内容とローカルの内容を無理やり統合できます。

もしコンフリクトが起きたら

- エディタで変更が競合している部分を手動で修正してください。
- 修正が終わったら、

```
git add .  
git commit -m "コンフリクト解消"
```

をして、コンフリクト状態から抜けましょう。

その後

もう一度pushを試してください。

```
git push origin main
```

この手順で問題は解決するはずです。質問あればどうぞ！

error: remote origin already exists.

「error: remote origin already exists.」というエラーは、すでにリモートリポジトリ（origin）が設定されている状態で、同じ名前のリモートを追加しようとしたときに出了。

解決方法は以下のいずれかです。

1. 既存のリモート設定を確認する

ターミナルで次のコマンドを実行して、現在のリモート設定を確認します。

```
git remote -v
```

ここで表示されるURLが正しいGitHubリポジトリであれば、改めて追加する必要はありません。

2. もしリモートURLを変更したい場合（たとえば間違っている場合）

既存のremote originを一度削除してから新しく追加します。

```
git remote remove origin  
git remote add origin https://github.com/ユーザー名/リポジトリ名.git
```

3. 追加不要なら、そのままpushやpullを使う

リモートは既に登録されているので、例えばプッシュはこうします。

```
git push origin main
```

このエラーはリモートの設定の重複だけの問題なので、慌てずリモートURLを確認して正しい操作を選べば対応できます。わからなければまた質問してくださいね。

git init

```
git add .  
git commit -m "最初のノートブックアップロード"  
git branch -M main  
git remote add origin https://github.com/ユーザー名/リポジトリ名.git  
git push -u origin main
```

あなたが書いたGitコマンドは、ローカルのJupyter NotebookなどのファイルをGitHubの新しいリポジトリにアップロードする基本的な流れです。以下のように使います。

Gitコマンドの意味と使い方

1. `git init`
→ 今いるフォルダをGitの管理下に置きます。
2. `git add .`
→ フォルダ内のすべてのファイルをGitの管理対象に追加します。
「`.`」は「このフォルダすべて」という意味です。
3. `git commit -m "最初のノートブックアップロード"`
→ 今追加した変更を「最初のノートブックアップロード」というメッセージ付きで記録します。

4. `git branch -M main`
→ メインのブランチ名を "main" にします。（GitHubの標準ブランチ名）
5. `git remote add origin https://github.com/ユーザー名/リポジトリ名.git`
→ GitHubのリポジトリ（ネット上の自分の作品フォルダ）を「origin」という名前で登録します。
※ここはあなたの実際のGitHubユーザー名とリポジトリ名に置き換えてください。
6. `git push -u origin main`
→ ローカルの「main」ブランチの内容をGitHubの「origin」リポジトリに送ります。

注意点

- 「git remote add origin ...」でエラーになる場合は、すでにremote設定があるので「git remote remove origin」を先に実行してから再度追加してください。
- GitHubリポジトリのURLは自分のアカウントのページで新規リポジトリを作った後に表示されるので、そのURLをコピーして使います。

この流れでJupyter NotebookのコードをGitHubに公開できます。わからないところがあれば質問してください。



anacondaのenvで環境構築した場合、モジュールなどの情報をgithubに挙げる必要がある？

Anacondaの環境構築（env）で作った仮想環境のモジュール情報は、GitHubにあげる「必要は基本的にあります」。理由とやり方は以下の通りです。

なぜモジュール情報をGitHubにあげる必要があるか？

- あなたのコードを他の人が同じ環境で動かせるようにするため。
- Pythonやライブラリのバージョン違いで動かないトラブルを防ぐため。
- 自分でも別のパソコンや将来再現できるようにするため。

どうやってモジュール情報をGitHubにあげるのか？

1. Anacondaの仮想環境を保存するため、現在の環境のパッケージ情報を書き出します。
ターミナルで、仮想環境をアクティベートした状態で以下のコマンドを実行します。

```
conda env export > environment.yml
```

これで「environment.yml」というファイルに必要なモジュールとバージョン情報が保存されます。

2. この environment.yml ファイルをGitHubのリポジトリにアップロードします。
こうすることで、他の人も同じ環境を作りやすくなります。

3. 他の人や未来の自分がこの環境を再現するときは、

```
conda env create -f environment.yml
```

というコマンドを使って、同じ環境を作れます。

環境やモジュールの再現はAI職向けのポートフォリオで非常に大切なので、必ずenvironment.ymlを作ってGitHubに含めることをおすすめします。[22] [23] [24]



environment.ymlはどこに保存さえた？

environment.yml ファイルは、次のコマンドを実行した「カレントディレクトリ（現在開いているフォルダ）」に保存されます。

```
conda env export > environment.yml
```

つまり、ターミナルやコマンドプロンプトでそのコマンドを打ったときに、開いているフォルダ（例えばデスクトップやプロジェクトのフォルダ）に environment.yml が作られます。

確認方法

- ターミナルで pwd（Mac/Linuxの場合）や cd（Windowsの場合）を使って今いる場所を確認してください。
- そこに environment.yml ファイルがあるはずです。
- ファイルエクスプローラー（フォルダ画面）で見つけてGitHubにアップロードします。

もし違う場所に保存したい場合は、保存時にパスを指定してください。たとえばデスクトップに保存したい場合は、

```
conda env export > ~/Desktop/environment.yml
```

（Windowsはパスの書き方が少し違います。）

わからなければ教えてくださいね。

もし違う場所に保存したい場合は、保存時にパスを指定してください。たとえばデスクトップに保存したい場合は、

text

```
conda env export > ~/Desktop/environment.yml
```

Windowsはパスの書き方を教えてください

WindowsでAnacondaの環境情報を指定した場所に保存する場合のパスの書き方は以下のようになります。

例えば、デスクトップに environment.yml を保存したい場合は、コマンドプロンプトやPowerShellで次のように書きます。

```
conda env export > %USERPROFILE%\Desktop\environment.yml
```

ここで、

- %USERPROFILE% はWindowsのユーザーフォルダのパス（例: C:\Users\あなたの名前）
- \Desktop\ はデスクトップのフォルダを示します

つまり、%USERPROFILE%\Desktop\environment.yml がデスクトップ上のファイルのフルパスになります。

もし他の場所に保存したい場合も、Windowsのフォルダパスをこのように書けばOKです。

例：

```
conda env export > C:\path\to\your\folder\environment.yml
```

まとめると、Windowsではパスを %USERPROFILE%\Desktop\ のように書いてあげれば、好きな場所に環境ファイルを保存できます。

必要に応じてそのファイルをGitHubにアップロードしてください。これで環境の再現がしやすくなります。

✿

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 6 threads

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 1.38 KiB | 1.38 MiB/s, done.

Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To https://github.com/atsuVictospin/my-ai-portfolio_MNIST_001.git

5c76910..d53ef45 main → main

branch 'main' set up to track 'origin/main'.

Gitのpushが成功し、あなたのローカルの変更がGitHubのリモトリポジトリに正常に反映されたことを示しています。

- 「Enumerating objects」「Counting objects」から始まり「Writing objects: 100%」「remote: Resolving deltas: 100%」は正常な動作ログです。
- 「main → main」はローカルのmainブランチがリモートのmainブランチに反映されたことを意味します。
- 「branch 'main' set up to track 'origin/main」はローカルのmainブランチがリモートのmainブランチと連携する設定になったことを示しています。

これでGitHubにJupyter Notebookなどのファイルがアップロードされ、ポートフォリオとして公開できる状態になりました。もし公開後に変更があれば、同様にコミットしてpushすれば最新版になります。何かほかに質問があれば教えてください。

environment.ymlのprefixは公開してもよい？

environment.ymlに含まれるprefixは、その環境が作られたあなたのパソコン上の絶対パスで、個人のユーザー名やフォルダ構造が含まれていることが多いです。この情報をGitHubなどの公開場所にそのまま上げるのはおすすめしません。

理由

- 個人情報（ユーザー名やフォルダ構造）が含まれている可能性があるため。
- 他の人がこのprefixを使って環境を作るとエラーになる可能性があるため。
- 環境を作り直すときはデフォルトの場所に作るので、このパスは不要であり、消して問題ありません。

どうすればよい？

- environment.ymlを作るときにprefixを除外するか、
- 作成後にenvironment.ymlファイルからprefix:の行を手動で削除してください。

例として、conda env exportの際に次のコマンドを使うとprefixを除外できます。

```
conda env export --no-builds | grep -v "^prefix:" > environment.yml
```

(Windowsの場合はgrep -vが使えないのでエディタで手動削除するか、別の方法を使ってください。)

まとめると、prefixは公開せずに削除するのが一般的で安全です。そうすると他の人も環境をスムーズに再現できます。[41] [42] [43]

公開用のenvironment.yml作成時のベストプラクティス

公開用の environment.yml を作成するときのベストプラクティスは以下の通りです。これに従うと、誰でも問題なく環境を再現でき、不要な情報を含まない安全なファイルになります。

公開用 environment.yml ベストプラクティス

1. prefix は削除する

- 先述の通り、パソコン特有の絶対パスである prefix: の行は必ず削除しましょう。
- これを残すと他人が環境を作れなかったり、自分の個人情報が漏れる恐れがあります。

2. 不要なビルド情報の除外

- conda env export に --no-builds オプションを付けると、環境のビルド番号など細かい情報を除外でき、柔軟な環境再現が可能です。
- 例：

```
conda env export --no-builds > environment.yml
```

3. pipでインストールしたパッケージ情報も入れる

- conda環境内で pip 経由で入れたものも含めるため、通常の conda env export コマンドで対応できます。

4. 最小限の環境にする（必要なパッケージだけに絞る）

- 長期間使っている環境は使わないパッケージも入りがちです。
- 新規に環境を作って必要なパッケージだけ入れ、environment.ymlを作るのがおすすめです。

5. 手動で不要な項目をチェック&編集

- 例えば環境変数や特殊な設定、パーソナルなパスなどが残っていないか最終確認しましょう。

6. コメントを入れてわかりやすくする

- ファイルの先頭に「このファイルは～のための依存関係です」など簡単な説明をコメントで入れると親切です。

簡単な作成例コマンドまとめ

```
conda env export --no-builds > environment.yml
```

その後、テキストエディタで

- `prefix:` を探して削除
- コメントを追加（任意）

この手順で作られた `environment.yml` は、誰でも簡単に

```
conda env create -f environment.yml
```

で環境を再現でき、AI職ポートフォリオとしても使いやすいファイルになります。

github上で変更した内容を、ローカルリポジトリで適用させる方法を高校生にもわかるように丁寧に教えてください。

GitHub上で変更した内容を自分のパソコン（ローカルリポジトリ）に反映させる方法を、高校生にも分かりやすく丁寧に説明します。

1. 「git pull」を使う

GitHubにある最新の変更（アップデート）を自分のパソコンに取り込むには、ターミナルやコマンドプロンプトで次のコマンドを実行します。

```
git pull origin main
```

- `git pull` は「リモートから変更を引っ張ってくる」という意味です。
- `origin` はGitHubのリモートリポジトリの名前です。（普通はこれが使われます）
- `main` はブランチの名前で、最新の変更があるところです。

2. pullすると何が起こるか？

- GitHub上で追加・編集したファイルやコードが、自分のパソコンのファイルに自動的に反映されます。
- もし自分のパソコンで同じ部分を変えている場合は、どちらを使うか迷う「コンフリクト」が起きることがあります。

3. コンフリクトが起きたらどうする？

- ターミナルやGitがコンフリクトを知らせます。
- エディタ（メモ帳やVSCodeなど）で該当ファイルを開くと、どこが競合しているか特別な記号で示されています。
- 自分が残したいコードだけを残し、余計な部分を削除して保存します。
- その後、以下のコマンドで解決をGitに伝えます。

```
git add .  
git commit -m "コンフリクト解消"
```

4. まとめ

- GitHubの最新の変更をローカルに取り込みたいなら → `git pull origin main`
- もし競合が起きたら → 手動でファイルを修正し、`git add` と `git commit` で確定する

これで簡単にGitHub上の更新を自分のパソコンに適用できます。わからないところがあれば気軽に質問してくださいね。

remote: Enumerating objects: 9, done.

```
remote: Counting objects: 100% (9/9), done.  
remote: Compressing objects: 100% (6/6), done.  
remote: Total 6 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)  
Unpacking objects: 100% (6/6), 2.06 KiB | 110.00 KiB/s, done.  
From https://github.com/atsuVictospin/my-ai-portfolio\_MNIST\_001
```

- branch main → FETCH_HEAD
d53ef45..c5a8431 main → origin/main
Updating d53ef45..c5a8431
Fast-forward
 README.md | 6 ++++++
environment.yml | 2 +-
2 files changed, 7 insertions(+), 1 deletion(-)

このメッセージは、GitHub上の最新の変更をローカルリポジトリに問題なく取り込めたことを示しています。

- remote: Enumerating objects や Unpacking objects は正常にデータがGitHubからダウンロードされたことを表します。
- Fast-forward はあなたのローカルの履歴がGitHubの最新に追いついただけで、コンフリクト（競合）なしに更新が完了したという意味です。
- 更新されたファイルが README.md と environment.yml で、それぞれ内容の追加や一部修正があったことがわかります（7行の追加、1行の削除）。

このように、`git pull origin main` の操作で、安全にGitHubの変更をローカルに反映できています。引き続き作業を進めてください。もし質問があればお知らせください。

1. https://note.com/shimada_g/n/nc1bded110b52
2. <https://ai-respecter.com/perplexity-prompt/>
3. <https://shift-ai.co.jp/blog/31145/>
4. <https://note.com/brightiers/n/n21a6f6dff5f0>
5. <https://generative-ai.sejuku.net/blog/564/>
6. <https://www.lemon8-app.com/@popdigitalselling/7537898454133228050?region=jp>
7. <https://officeforest.org/wp/> 【2025年夏】これで追いつく！生成ai「gemini」これだけ/
8. <https://oshiete.ai/search?orderBy=favoriteCountDesc>
9. <https://www.i3design.jp/in-pocket/12510>
10. <https://www.lemon8-app.com/@thepnateephojan/7489873098490315272?region=jp>
11. <https://digi-mado.jp/article/75041/>
12. <https://crexgroup.com/ja/development/security/what-is-mit-license/>
13. https://qiita.com/takobuta_dev/items/120e710ab99721e98f8d
14. <https://e-words.jp/w/MITライセンス.html>
15. <https://www.plan-b.co.jp/blog/creative/10428/>
16. https://detail.chiebukuro.yahoo.co.jp/qa/question_detail/q10234062765
17. <https://yamory.io/blog/about-mit-License>
18. https://www.tento-net.com/blog/thinkabout_copyright_license
19. https://www.mediaprimestyle.jp/column_post/mit-license/
20. <https://manumaruscript.com/oss-licences-comparison/>
21. <https://github.com>
22. https://qiita.com/ozaki_physics/items/13466d6d1954a0afeb3b
23. <https://zenn.dev/kuuki/articles/anaconda-copy-env>
24. https://note.com/jtamas_engineer/n/n4512a08e188d
25. https://www.reddit.com/r/learnpython/comments/lmv46f/install_package_in_github_into_a_conda_environment/
26. https://koji.noshita.net/courses/compbio/archived_misc/envpython/
27. https://gen-todoroki.hatenablog.com/entry/virtual_environment
28. <https://qiita.com/kensussu/items/67784f8a3f3b474e7866>
29. https://www.reddit.com/r/learnpython/comments/j7ani2/installing_library_from_github_in_anaconda/
30. https://py4basics.github.io/0_Preparations.html
31. <https://sakizo-blog.com/206/>
32. <https://docs.conda.io/docs/user-guide/tasks/manage-environments.html>
33. <https://stackoverflow.com/questions/41274007/anaconda-export-environment-file>
34. <https://note.com/heroyuki/n/n2b821cc52508>
35. <https://docs.conda.io/projects/conda/en/stable/commands/env/export.html>
36. <https://github.com/conda/conda/issues/11114>
37. <https://www.monicathieu.com/posts/2024-05-20-conda-env-export-from-history.html>

38. https://qiita.com/ozaki_physics/items/13466d6d1954a0afeb3b
39. <https://www.anaconda.com/docs/getting-started/working-with-conda/environments>
40. <https://qiita.com/nshinya/items/cb1cffabc3305c907bc5>
41. <https://zenn.dev/keitakahata/articles/228b726c90963f>
42. <https://qiita.com/junkor-1011/items/cd7c0e626aedc335011d>
43. <https://hiroki-sawano.hatenablog.com/entry/2018/11/11/122808>
44. <https://techgym.jp/column/python-package/>
45. <https://learn.microsoft.com/ja-jp/visualstudio/python/managing-python-environments-in-visual-studio?view=vs-2022>
46. <https://namileriblog.com/python/conda-commands/>
47. <https://paper.hatenadiary.jp/entry/2020/08/14/213648>
48. <https://sakizo-blog.com/206/>
49. <https://docs.github.com/ja/code-security/dependabot/working-with-dependabot/dependabot-options-reference>
50. <https://www.palantir.com/docs/jp/foundry/transforms-python/unit-tests>