

非プログラマ、科学計算にRやPython使う人向けの Git, GitHubチュートリアル

目次

- §0 イントロダクション
 - §0-1 このチュートリアルの内容
 - §0-2 バージョン管理とは
 - §0-3 どうして学ぶ必要があるのか
 - §0-4 参考にしたサイトなど
- §1 準備
 - §1-1 Githubアカウントの作成
 - §1-2 RとRStudioのインストール or アップデート
 - §1-3 Gitがすでにインストールされているかチェック
 - §1-4 Gitが入っていなかった場合
 - §1-5 Gitとgithubを繋げる。
 - §1-6 gitクライアントについて (オプション)
 - §1-7 SSHキーの設定
 - SSHキーをもっているかを確認する。
 - SSHキーを作成する。
 - SSH-agentが有効かどうかを確認する。
 - SSHキーを GitHubに登録する。
- §2 個人開発編
 - §2-1 初めてGitHubでリポジトリを作ってRStudioで使ってみる。
 - Githubでリポジトリを作る。
 - RStudioでリポジトリをローカル (PC上) にクローン (コピー) する。
 - ローカルでファイルを変更してコミットしてプッシュする。
 - README.mdファイルの作成
 - ここからの流れの説明
 - 変更点をステージに送る。
 - コミットする。
 - 変更点をプッシュする。
 - §2-2 日常的な作業をしてみる。 (Pull, Commit, Push)
 - まずはプル
 - 次に編集してコミット
 - 最後にプッシュ
 - §2-3 RStudio特有の問題 .gitignoreによる追跡の除外
 - §2-4 初めてのブランチ 家とラボの両方でコードを書く
 - ブランチがあると何が嬉しいのか。
 - 実際にブランチを使ってみる。
 - 1. 新しいブランチを作つてみる。
 - 2. ブランチで変更してコミットしてみる。

- 3. マージする。
- §2-5 コンフリクトに対応してみる。
- §2-5 VSCodeをGitHubと連携してみる。
- §3 共同開発編
 - §3-1
- Appendix1 パスとディレクトリ
 - パスとディレクトリ
 - 絶対パスと相対パス
 - 「いる」とは何か
 - ホームディレクトリとルートディレクトリ
- Appendix2 そもそもGitとGithubとは？
- Appendix3 Markdown
 - 基本的な記法
 - 見出し
 - 太字
 - 斜体
 - 箇条書き
 - 番号付きリスト
 - リンク
 - 画像
 - Markdownのメリット
 - Markdownのデメリット
 - まとめ

§0 イントロダクション



書きとどめよ。議論したことは、風の中に吹き飛ばしてはいけない (ガリレオガリレイ)

§0-1 このチュートリアルの内容

このチュートリアルは、プログラマではないけれど科学計算のためにRやPythonを使う人に向けた、GitとGitHubの基本的な使い方の解説です。

このチュートリアルを通じて、GitとGitHubの基本的な概念や操作方法を学び、効率的なプロジェクト管理や共同作業ができるようになることを目指します。

このチュートリアルは大きく3つに分かれています。

- 準備編
- 個人開発編
- 共同開発編

最初の準備編ではGitやGitHubを使う準備をします。ここはGitをどんな使い方で使うにしても大切な部分です。慣れていないと難しい部分も多いですが、頑張りましょう。

次の個人開発編は通常Git, GitHubのチュートリアルではあまり出てこない、「一人で」バージョン管理をする場合の使い方を学びます。最初はRを念頭にRStudioとGitHubの連携についてみていきます。

Git等はシステム系で出てくることが多く、ネット上の情報も多くは情報系の人が書いています。そういう人たちの作るプログラムは基本的に多人数で共同して作成しており、成果物自体がアプリ等の形で世に出ることが多いです。一方で非プログラマで科学的研究にGitを用いる私たちの場合、解析プログラムは世界中を見渡しても多くて数人、なんなら自分しか使わないことがあります。これは研究テーマ自体の独自性に寄与しているので割と皆さんそうなのではないかと思います。また機密性の観点から、共有できる範囲が広くて共同研究者レベルまでということもあり、そもそも共同開発になりにくいです。ただGitは共同開発だけではなくて、個人開発にも大変便利です。2番目の章では個人開発のためのGitの使い方を学びます。gitを少し触ったことのある人に言うなら、Pull requestやForkといった概念はここでは出てきません。ブランチは少し出でますが、なくても良いかもしれません。

最後の共同開発編は、複数人で解析プログラムを作成する場合の使い方です。具体的にはPythonの解析用の基本コードやRの図表作成用コードは共有してもいいのかなと思っております。またここを勉強すると世界中のラボで作られたコードを使ったり改良したりといった作業ができるようになります。ただ前述の通り、生物系では必須ではないかもしれません。

§0-2 バージョン管理とは

みなさんはwordファイルに「report_20240630.docx」や「report_latest.docx」、「レポート最新版の最新版.docx」といったような名前をつけたことがあるでしょうか？

もしあるなら、みなさんはすでにバージョン管理をしたことがあると言うことです。

バージョン管理はまさに先ほどのように、テキストやスクリプトを更新するたびに新しいバージョンを作成し、変更点を追跡するプロセスです。

先ほどのファイル名を変更してバージョン管理をすると言うのも一つの手です。

ただしみなさんも体感したことがあるかもしれません、上記の方法だと

- どれが最新版か分からなくなる。（「最新版.docx」と「最終版.docx」はどちらが最新版？？）
- どういう変更を加えたのか分からなくなる。（昨日のエクセルではうまく解析できたのに。。。）

といった問題が起こりがちです。

GitやGitHubはこれらの問題を解決してくれるツールと考えれば良いと思います。

§0-3 どうして学ぶ必要があるのか

コードの変更履歴を全て残せる。一般に研究室において、データの管理は厳しく言われることが多いと思います。一方で解析するコードを自分で書いたりした場合にはあまり厳しく管理について言われることはないのではないかでしょうか。コードを本気で（手動で）管理したいなら毎回解析に使用したソースコードをノートに貼り付ければ良いと思いますが、そんなことをしている人は見たことがありません。

情報系等ではソースコードはGitで管理することが多いようです。Git管理すれば、「あの時のコード」がどんなコード

であって、その後どんな変更を加えられたのかを全て記録することができます。これによって、「あの時のコードであればうまく解析できたのに」と涙を飲む回数を減らせるはずです。

コードの再利用に対してのコストが減る 少しでもプログラムを書いたことのある人なら、1ヶ月前の自分の書いたコードがどんなふうに見えるかわかると思います。基本的には同じコードを使って解析するのは自分自身がメインになると思いますが、解析するごとにソースコードを読んでまた理解するのは大変時間がもったいないと思います。ましてや他の人(ラボの先輩など)のコードを読む際にはより時間がかかります。GitHubでコードと共にドキュメントを残することで、コードで何をしたいのか、簡単に使い始めるのにはどうすればいいのかということをシェアできます。このことで、コードの再利用に対してのコストが減少し、研究がより早く進むことが期待できます。

コードのシェアが求められることが多くなっている またもう一つの大きな点として、コードシェアが求められることが多くなってきています。適当な論文で"github"と検索してみるとわかると思いますが、最近は生物系の論文でも解析コードをGithubでシェアしていることが多いです。これは再現性の担保の観点から必要とされているからでしょう。生データとコードがシェアされていれば、データ解析の部分で変なことが起こっていないかを世界中で検証可能です。実際に、生物系のジャーナルで論文投稿した際に「解析に使ったコードはGithubなどでシェアしてくださいね」という要請が来た例が身近にありました。このような必要性の観点からもGithubを勉強することは大事だと思います。

§0-4 参考にしたサイトなど

<https://happygitwithr.com>

§1 準備

それでは必要なソフトなどをインストールしていきます。ここが一番大変なところです。(つまらないですが)

§1-1 Githubアカウントの作成

<https://github.com>. こちらからアカウントを作成します。日本語の記事ではこちら。

https://reffect.co.jp/html/create_github_account_first_time. がわかりやすいと思います。

注意点としては、アカウント名は比較的いろんなことに使うので、恥ずかしくなくて長過ぎないものにしておきましょう。

メールアドレスとユーザー名はすぐに使うので覚えておきましょう。

§1-2 RとRStudioのインストール or アップデート

1. Rをインストール. ここか、ミラーサイトからどうぞ

<https://cloud.r-project.org>

もしすでにRをインストールしているなら、スキップで大丈夫ですが、良い機会なのでRが最新版か確認して、必要ならアップデートしておくと良いでしょう。

2. RStudioのインストール ここからどうぞ

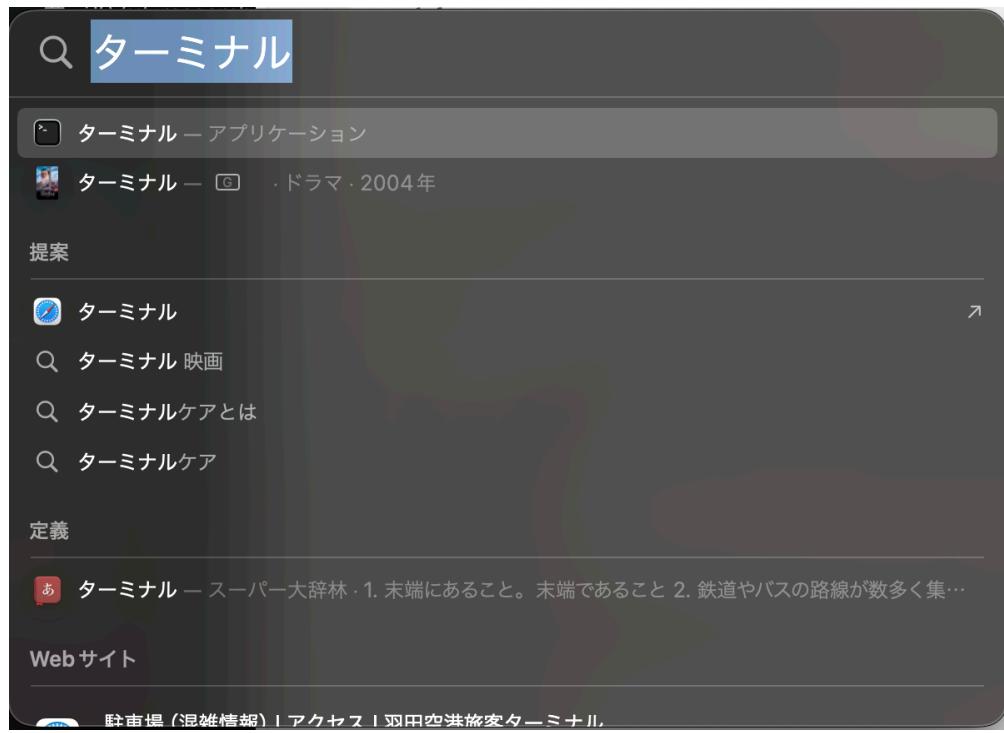
<https://www.rstudio.com/products/rstudio/download/preview/>

こちらももしもライнстールしているなら、スキップ可能です。アップデートは必要ならしてください。

§1-3 Gitがすでにインストールされているかチェック

最初にすでにgitがインストールされていないかチェックします。

ターミナル (windowsの場合にはコマンドプロンプト) を開きます。



以下を実行します。 Macの場合

```
▶ which git
```

Windowsの場合

```
▶ where git
```

(linuxの人はそもそもこの辺りで詰まることがないかと思うので、省略 というかLinuxを使うような方はもっとちゃんと独学してください。)これを入れてもし以下のようない返答が返ってきてているなら、すでにgitが入っています。

```
/usr/bin/git
```

ついでにGitのバージョンをチェックしてみましょう。

```
▶ git --version
## git version 2.45.2
```

こんな感じで返ってくればOK

§1-4 Gitが入っていなかった場合

インストールしてください。 こちらからどうぞ <https://git-scm.com/downloads>

§1-5 Gitとgithubを繋げる。

以下をmacの場合ターミナル、windowsの場合Git Bashで実行します。 GitBashは以下のように探すといいと思います。

GitBashを起動しよう

無事インストールされたら起動してみましょう。

どこにファイルが保存されているかわからない場合、Cortanaの検索窓に git bash と入力して検索すると出てきます。



Your name と Your addressを自分のものに変えてください。

```
git config --global user.name "Your name"  
git config --global user.email "Your address"  
git config --global --list
```

これで作ったGithubアカウントとあなたのPCのGitを繋げることができます。

§1-6 gitクライアントについて (オプション)

(最初は飛ばしてOK) GitやGitBashは便利ですが、見た目がいかつくて難しいです。文字がたくさん出てきて真っ黒の画面を見るのが好きならそのまま使えばいいと思います。

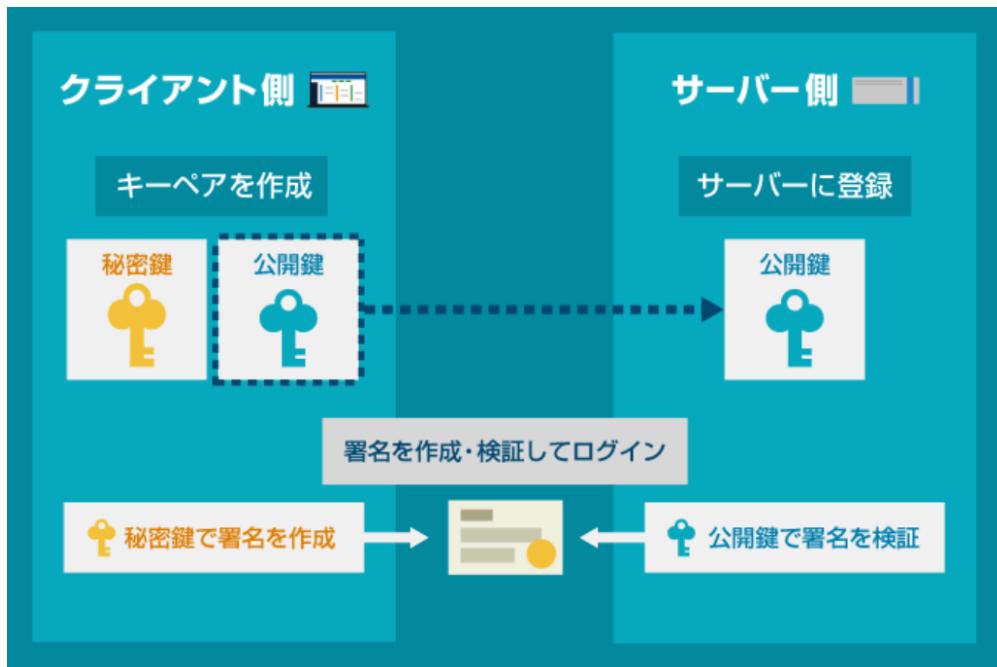
僕を含めて、プログラマじゃない人にはもっと画像が出てきた方が良いと思います。そのためにあるのがGitクライアントです。GitとGitクライアントの関係は簡単にいうならRとRStudioの関係にちかいと思います。RStudioは直感的に動かしやすいですが、裏で動いているのはRです。Rを直接動かすのは難しいと思います。そんな感じで、Gitを直感的に動かせるのがGit クライアントです。

ここまで書くと使った方が良さそうですが、正直RStudioやVisual studio codeの機能で同様のことができる、無理に使わなくて良いと思います。興味のある方はSourceTreeやGithub Desktopを調べてみてください。

§1-7 SSHキーの設定

(ここはめちゃくちゃ難しいです。頑張りましょう。)

SSHキーはGitとGithubが安全に通信するためのパスワードみたいなものと理解すれば良いと思います。公開するSSHキーを作成してPCに保存し、同様のものをGithubにも教えておくことで、認証を行います。以下の画像がわかりやすいかもしれません。



SSHキーをもっているかを確認する。

もしかしたらどこかでSSHキーをすでに作成しているかもしれません。そこで確認をしてみます。Macの方はターミナル、windowsの方はGitBashに移動して以下のコードを実行します。

▶ `ls -al ~/.ssh/`

これで存在しないと言われた場合にはキーを持っていません。ただ、仮に持っていたとしてもいい機会なので更新するといいかもしれません。

SSHキーを作成する。

▶ `ssh-keygen -t rsa`

この後に色々と出でますが、全部エンターで大丈夫です。最初に聞かれるのはSSHキーのファイル名、2番目と3番目はSSHキーにかけるパスワードですが、慣れないうちはパスワードをかけなくてもいいと思います。

こうすると、ホームディレクトリ [^1] 下に.sshというフォルダ [^2] が作成され、その中にid_rsaとid_rsa.pubが生成されます。

[^1]: ホームディレクトリって？？ってなった人は [Appendix1 パスとディレクトリ](#) を参照してみましょう。[^2]: 隠しフォルダになっているので、デフォルトの設定では見れないかもしれません。これを機にかくしフォルダを表示するよう に設定を変更することをお勧めします。

SSH-agentが有効かどうかを確認する。

ssh-agentはさっき作ったSSHキーを使ってくれるソフトウェアと思っておけばいいです。キーを作ったものの、うまくそのキーを使ってくれないと接続ができないです。

チェックのために、Macの人はターミナルで、windowsの人はGitBashで

```
▶ eval "$(ssh-agent -s)"
```

と打ってみてください。これで

```
Agent pid 11111
```

みたいなのが返ってくれば大丈夫です。ダメな人でMacの方は

```
▶ sudo -s -H
```

と打って(必要ならパスワードを打って)再度eval～を試してみてください。これはいわゆる"管理者権限で実行" みたいなやつです。作業が終わったら

```
▶ exit
```

で通常権限に戻すのをお忘れなく。

上記でもダメな場合やWindowsでダメな場合には私の手には負えないのでGitHubの該当箇所を当たるのが良いと思います。 <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

ここまでできたらssh-agentにキーを追加します。以下を実行してください。

```
▶ ssh-add ~/.ssh/id_rsa
```

Macの方でmacOS Sierra 10.12.2以降の方はもう一つやることがあります。まずは以下を実行です

```
▶ open ~/.ssh/config
```

これで

```
The file /Users/YOU/.ssh/config does not exist.
```

と返ってきた時には以下を実行してください。

```
touch ~/.ssh/config
```

再度以下を実行すると新しいウィンドウが開くと思います。

▶ open ~/.ssh/config

そしたら以下を書き込んでください。 Host github.com AddKeysToAgent yes IdentityFile ~/.ssh/id_rsa

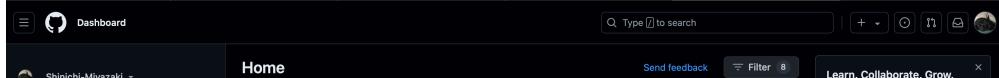
お疲れ様です。

SSHキーを GitHubに登録する。

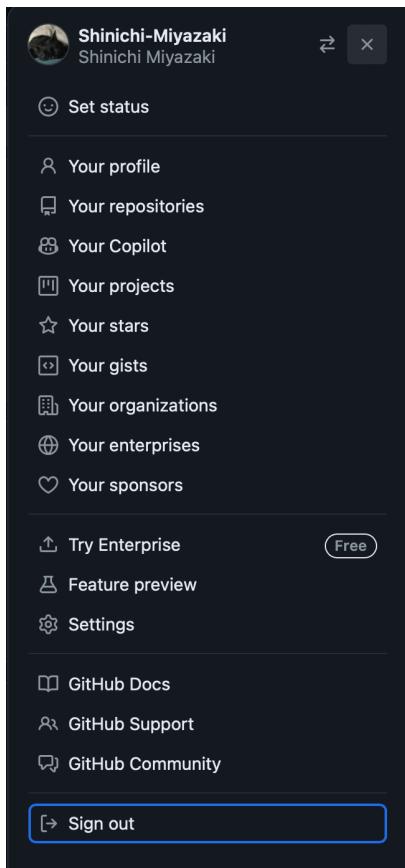
最後のステップはSSHキーをGitHubに登録するステップです。

1. まずはSSHキー(公開鍵)をコピーします。 .sshディレクトリ下にid_rsa.pubと言うファイルがあると思うので、こちらを適当なエディタで開きます。(windowsだとメモ帳、macだとテキストエディタ?) 内容をコピーしたらGitHubのホームページに行きます。(クリップボードを更新しないように!)
2. Githubのホームページから以下のように移動します。

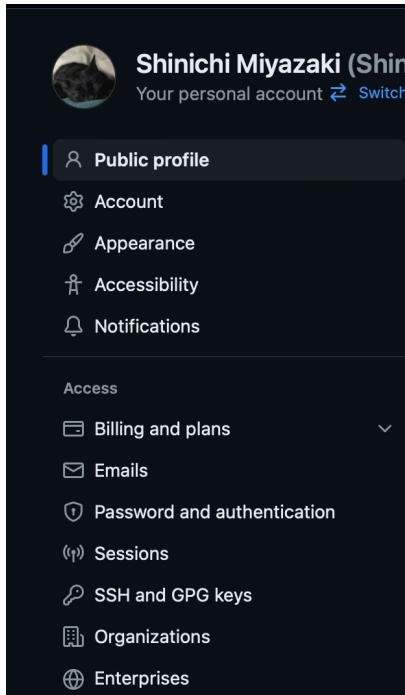
i. 自分のアイコンをクリック



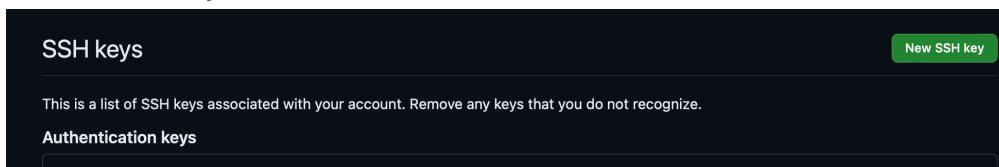
ii. 設定をクリック



iii. SSHをクリック



iv. New SSH keyをクリック



v. 適当なタイトル (大抵僕はPC名にしてます) をつけて、コピーしたキーを貼り付けてAdd

Add new SSH Key

Title

Key type

Authentication Key

Key

Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'

Add SSH key

これで完了です!

ここまでくると、あなたのPCがSSHと言う安全な方法を使ってGithubとやりとりできるようになりました。

普通はこの後はコマンドライン(真っ黒の画面)でGitを使う話に入るんですが、私たちは生物系で真っ黒の画面にはアレルギーがあるのでスキップします。もちろん慣れてきたら真っ黒い画面で色々してみてください。

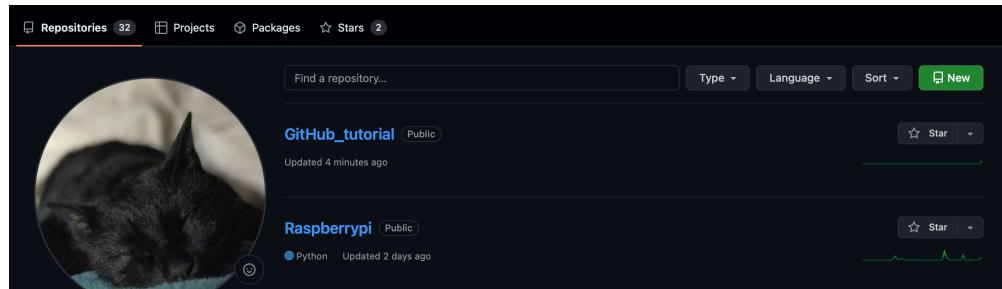
§2 個人開発編

§2-1 初めてGitHubでレポジトリを作つてRStudioで使ってみる。

Githubでリポジトリを作る。

それでは早速Githubでリポジトリを作つてみます。リポジトリといふのは感覚としてはディレクトリやフォルダと似たように捉えれば良いと思います。ここでいきなりわりかし重大な注意ですが、**慣れるまでは最初にGitHubでリポジトリを作る** というのを徹底した方がいいと思います。(特にRStudioユーザ) リポジトリはローカル(みなさんのPC上)でも作れます、RStudioではなぜかローカルで作ったリポジトリをGitHubで共有する際に色々と詰まつてしましました。自分だけならいいんですが、結局解決できなかつたので、、、あと参考にしたwebsiteもGitHubを最初に作るのを推奨しているので、、、

1. GitHubを開く



2. レポジトリの設定

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Required fields are marked with an asterisk (*).

Owner * **Repository name ***

 Shinichi-Miyazaki /

Great repository names are short and memorable. Need inspiration? How about **animated-palm-tree** ?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file
This is where you can write a long description for your project. [Learn more about READMEs](#).

Add .gitignore

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

Choose a license

A license tells others what they can and can't do with your code. [Learn more about licenses](#).

 You are creating a public repository in your personal account.

ここでは設定をします。

- 名前: 適当に でも覚えやすい名前にするといいです。これがPC上のディレクトリ名にもなるので長いと大変です。
- PrivateかPublicか: これは公開するかどうかです。研究関連で秘匿情報を含む可能性があるならPrivateが望ましいです。
- Add Readme: これはリポジトリの説明文を最初に追加しておくかどうかです。慣れないうちはなしでOK
- Add gitignore: これも最初は無視でOK
- Choose license: これも無視でOK みたいな感じで設定したら、リポジトリが完成です。

RStudioでリポジトリをローカル(PC上)にクローン(コピー)する。

今作ったリポジトリはGithub上(リモート)に存在しています。こちらを手元のPCを持ってきましょう。

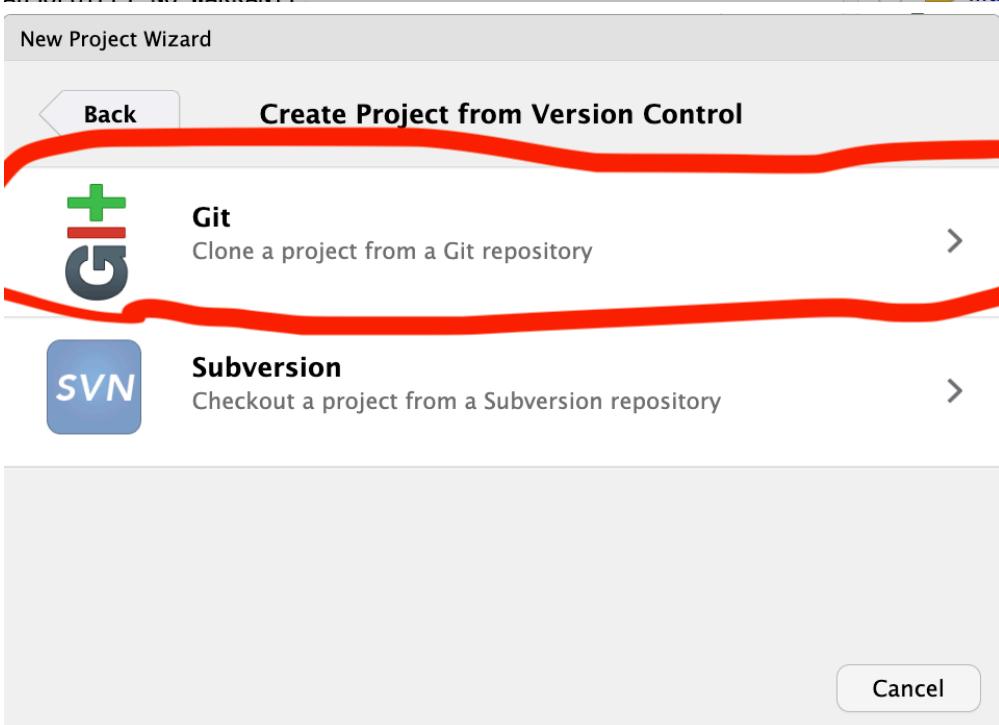
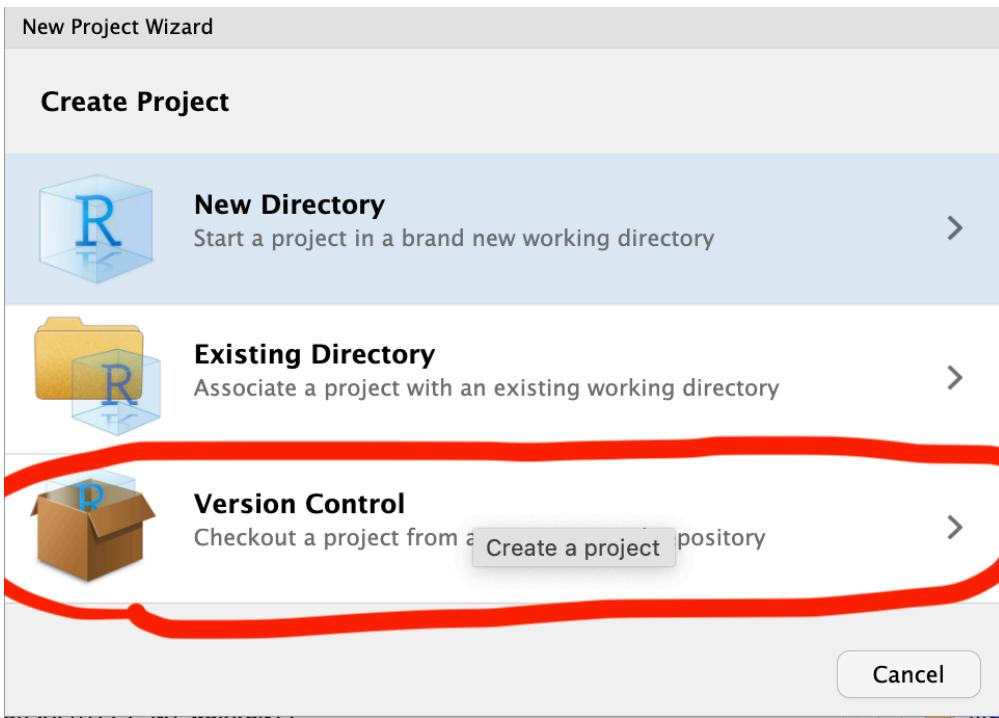
1. Githubでリポジトリを開く

2. リポジトリの右上の緑の部分を押す

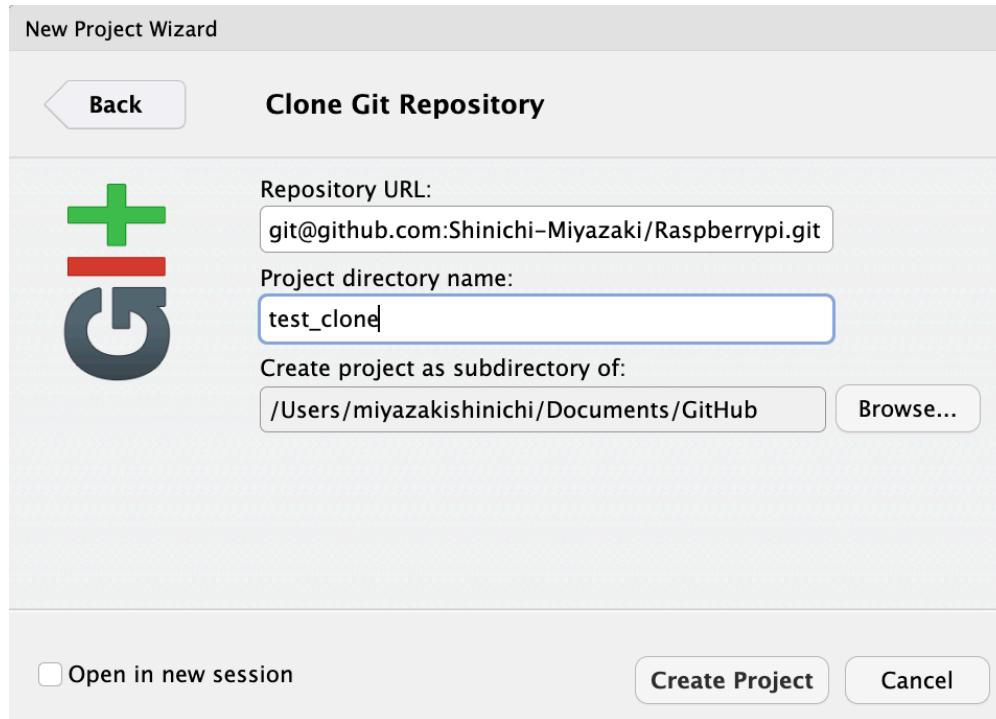
3. 下の画面が開くのでSSHの部分のクリップボードにコピーをおしてコピー

4. RStudioを開く

5. File/NewprojectからVersionControl/Gitを選択

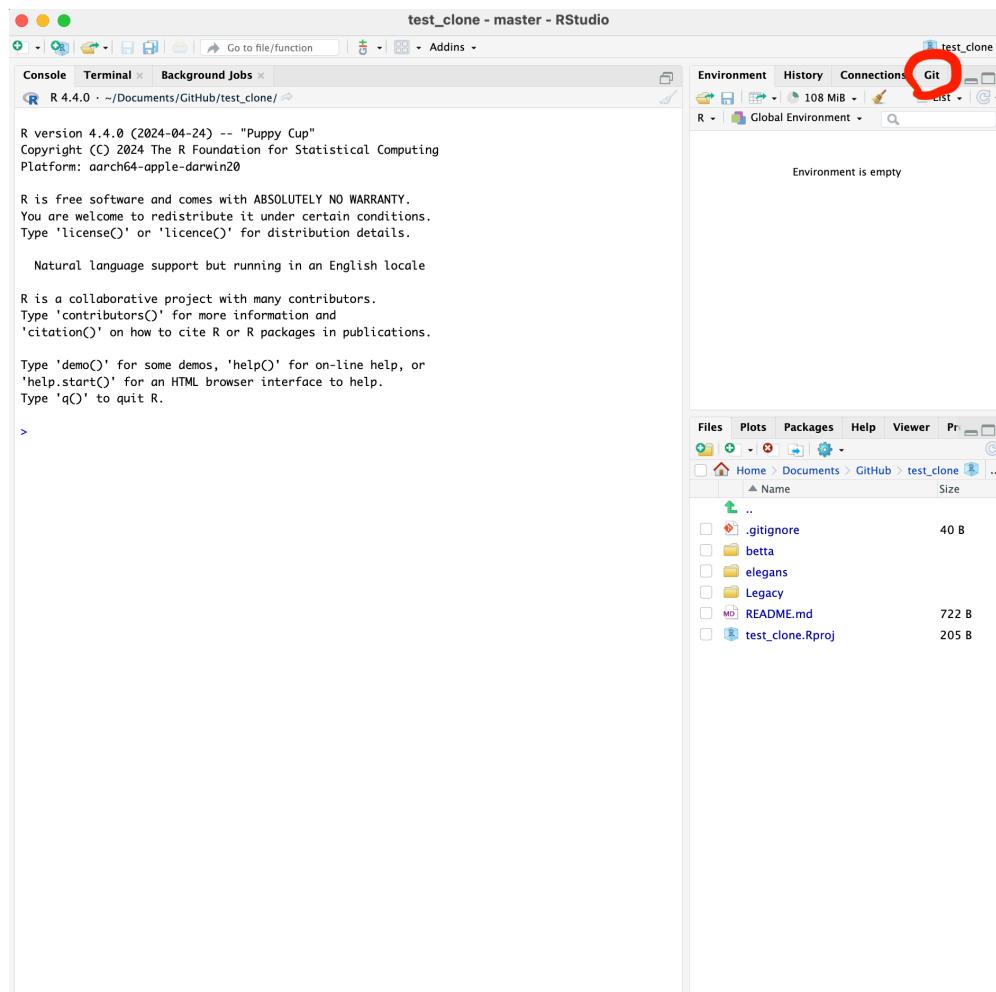


6. Project URL にクリップボードの中身をペースト



ここでどのフォルダに保存するかですが、ルート下にProjectsやrepos, githubとかの名前のディレクトリを作つて、そこに保存していくといいと思います。

7. 適当なフォルダ名をつけて保存 右上のPane (Environmentとかが入っているところ) に"Git"と表示されていれば成功です。



これでRStudioでGithubのリポジトリのコピーをひらけたことになります。

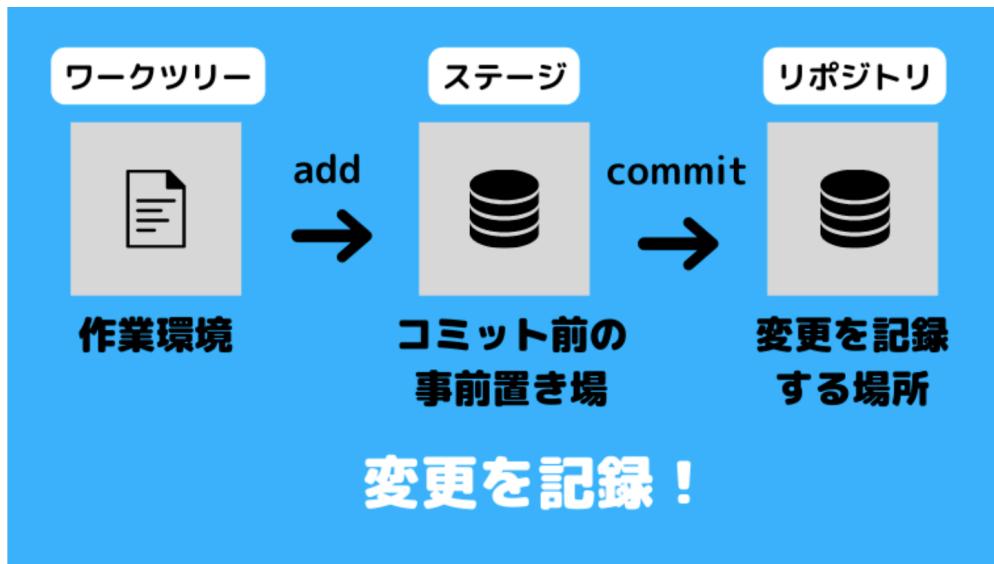
ローカルでファイルを変更してコミットしてプッシュする。

README.mdファイルの作成

それでは実際にファイルを作って、バージョン管理してみましょう。

レポジトリの説明書きであるREADME.mdというファイルを作ってみます。RStudioのfile/text fileでファイルを作成して、適当に書いてみましょう。Markdownに慣れていない方は[Appendix3 Markdown](#)を参照してみましょう。編集を終えたらSaveしたらファイル名を「README.md」に変えましょう。

ここからの流れの説明

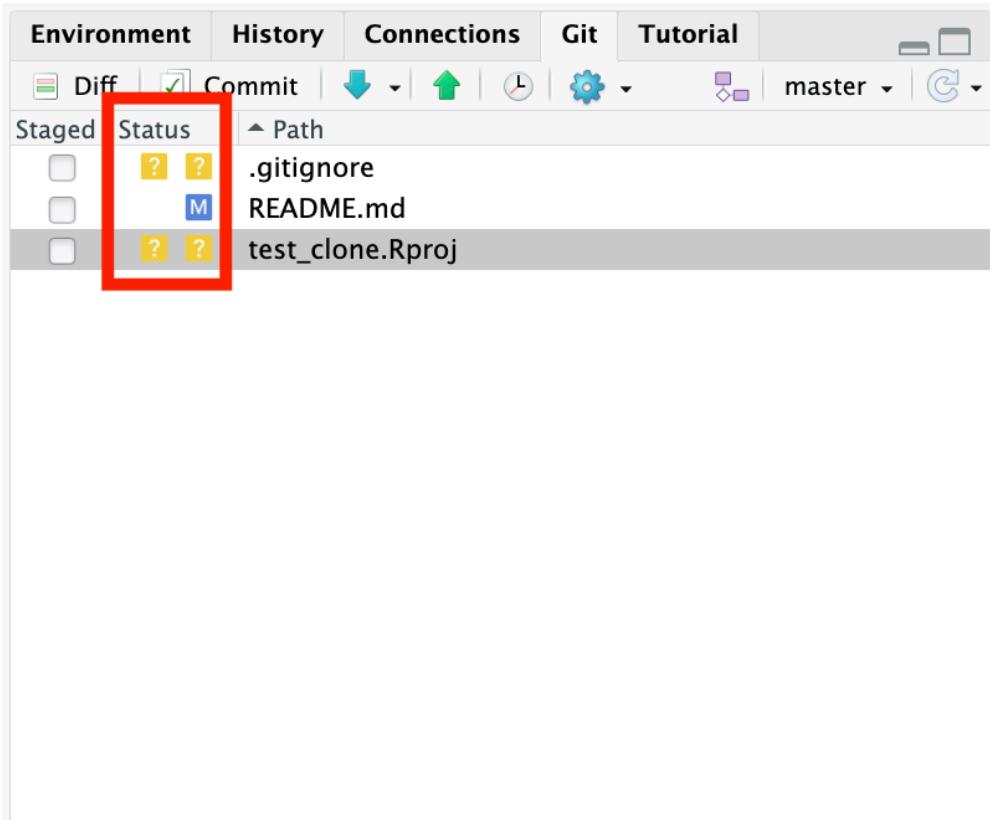


ステージにaddしてcommitしてpushする

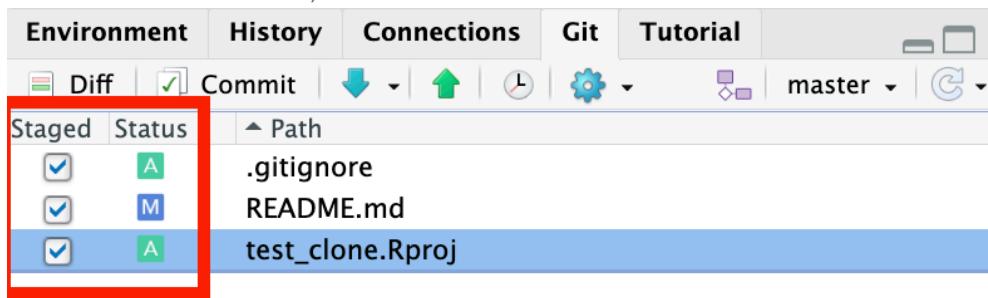
これがファイルを変更したときの基本的な流れです。

普段皆さんのがファイルを編集しているのがワークツリーです。そこから変更したものをステージという一時置き場において、その後コミットという動作でリポジトリに送ります。

変更点をステージに送る。



最初、Git Paneはこんな感じの見た目になっていると思います。列は左から - stage ステージに送るかどうかのチェックボックス - status 現在どういう状況か (ボックスが左なら変更がステージに送られている、右なら変更がまだステージに送られていない。両方に黄色のマークは新しく作成されたもの) - Path ファイルの場所 ステージに送りたいファイルにチェックボックスをつけま

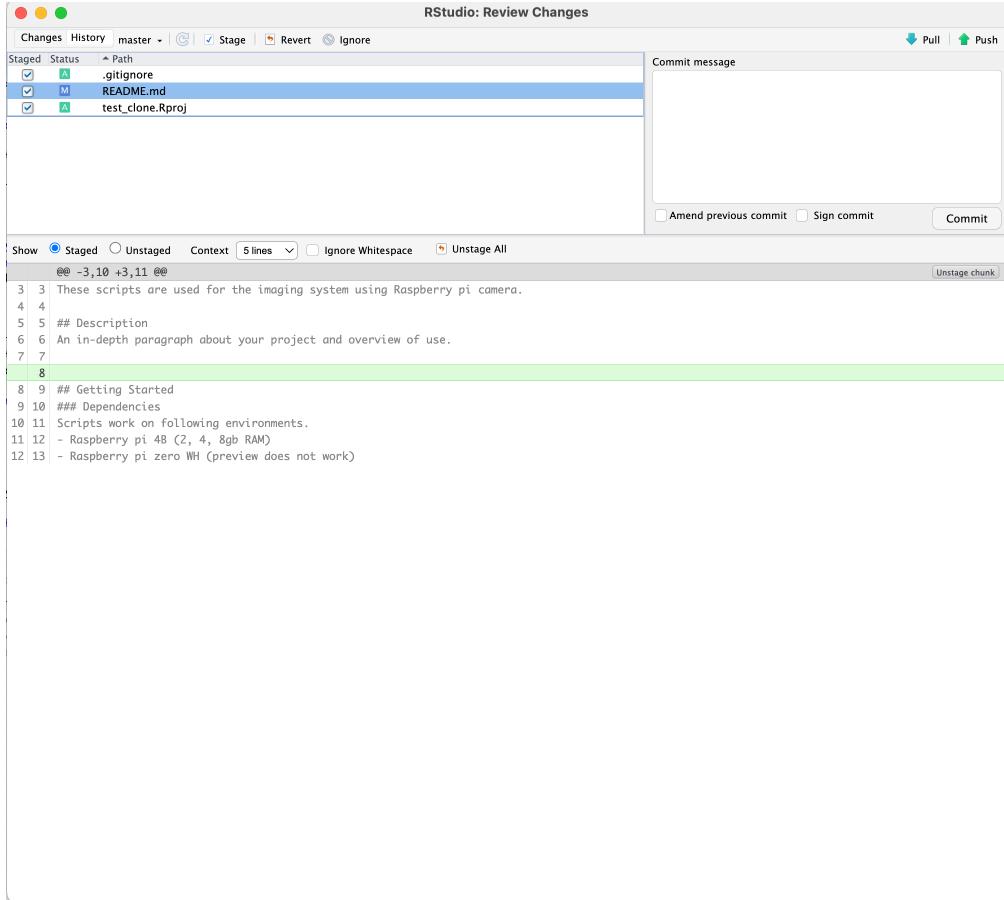


しょう。

コミットする。

コミットボタンを押しましょう。

以下のような画面が出ると思います。



左上の画面はさっき見たGit paneと似たような画面です。適当なファイルを選択すると、下の画面が変わることに気づくと思います。下の画面はどんな変更がなされたかが表示されています。赤色は削除された部分、緑色が追加された部分です。

右の画面はCommit messageを書く部分になります。ここはどんな変更を加えたのかを端的に書くと良いでしょう。スタイルとしては1行目に短いメッセージ、2行目を開けて3行目以降に詳しいメッセージみたいにすると良いと思います。

コミットメッセージの例

[modify] 凡例を追加

ggplotでグラフを書いた際に、自動で凡例をつけるように変更しました。凡例のラベルはデータの列名から取得するようにしています。

(1行目はGithubで見るとこんな感じです。)

A screenshot of a GitHub commit history. The commits are listed as follows:

Commit	Message	Date
.idea	change resolution	2 months ago
Legacy	bug fix	7 months ago
betta	bug fix	4 months ago
elegans	remove system build	4 days ago
README.md	add system build	4 days ago

The commit "change resolution" has a red oval circle around it.

変更点をプッシュする。

プッシュボタンを押しましょう。ここは特に他にすることはなくて、ただ押すだけです。

A screenshot of the RStudio "Review Changes" interface. The "Push" button is circled in red.

おすと色々と画面が出現すると思いますが、それは消してしまって大丈夫です。

Githubで変更が反映されているか見てみましょう。

The screenshot shows the GitHub repository 'GitHub_tutorial'. At the top, there are navigation links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. Below the header, the repository name 'GitHub_tutorial' is shown as public. The main area displays a list of commits. The first commit by 'Shinichi-Miyazaki' is highlighted with a red circle around the '19 Commits' link in the top right corner of the commit card. The commit details show 'write markdown' at 'ae03e20' committed 2 days ago. Other commits listed include 'add some modify', 'write markdown', 'Update GitHub_tutorial.md', and 'first commit'. On the right side, there's an 'About' section with 'No description, website, or topics provided.', a 'Readme' link, an 'Activity' section showing 0 stars and 1 watching, and a 'Releases' section with 'No releases published' and a 'Create a new release' button.

試しに、右側のcommitという部分を押してみると、これまでのコミットの履歴が出てきます。

This screenshot shows the 'Commits' page for the 'GitHub_tutorial' repository. The page title is 'Commits'. It lists 19 commits in chronological order. The commits are grouped by date: 'Commits on Jul 6, 2024' and 'Commits on Jul 5, 2024'. Each commit card includes the author (Shinichi-Miyazaki), the commit message, the commit hash, and a copy icon. The commit messages include 'write markdown', 'fix some typo', 'Update GitHub_tutorial.md', 'add message', and 'GitHub_tutorial.md を更新'. The commit 'GitHub_tutorial.md を更新' is marked as 'Verified'.

§2-2 日常的な作業をしてみる。 (Pull, Commit, Push)

前の章では初めてレポジトリを作ってそれをローカルにクローンして、ファイルを編集してコミットして、リモートにプッシュしてみました。次はすでに作ってあるリポジトリを使ってプログラムの更新をしてみましょう。

ここでは私は自分で作ったRCodesというレポジトリのなかのファイルを書き換えて、コミットしてプッシュしてみます。

まずはプル

まずはリモートの最新版でローカルのコードを更新します。

基本的にはローカルに最新版を置いておくのは得策ではなく、1日の作業が終わったらコミット & プッシュをしてリモートを最新版にしておくのが良いと思います。上記のようにしておけば、作業の開始は常にPullになるわけです。(ここでpullしないで進めていくと、のちに述べるコンフリクトという状況に陥ったりします。)

最初にpullをする方法を説明します。まずはRStudioで編集したいディレクトリ(プロジェクト)を開きます。

Git paneのところに下向きの緑矢印があると思いますので、そちらを押してもらうと、"Pull with rebase"という言葉が

The screenshot shows the RStudio interface with the GitHub tab selected. In the top right corner of the GitHub tab, there is a red circle around the 'Pull with Rebase' button.

出てくると思います。

こちらを押すと、Github上のリモートであるPCのローカルを更新してくれます。もしAlready up to dateというふうに出ていたら、ローカルがすでに最新版になっています。

次に編集してコミット

それでは最新版になったコードを書き直してみましょう。

ここはみなさんお好きなコードを書いていけばいいと思うのですが、私はRCodesというリポジトリの正規性の判定をしてくれるコードを書き直したくなっとします。

最初はこんな感じのコードを、

```

1 library(tidyverse)
2 library(gridExtra)
3 library(base)
4
5 Normality_test_plot <- function(data_array, plot_title){
6   pval <- shapiro.test(data_array)$p.value
7   g1 <- ggplot(data.frame(data_array), aes(data_array)) +
8     geom_histogram(bins=15) +
9     ggtitle(paste(plot_title)) +
10    theme_classic()+
11    scale_x_continuous(expand = c(0,0)) +
12    scale_y_continuous(expand = c(0,0))
13   g2 <- ggplot(data.frame(data_array), aes(sample = data_array)) +
14     geom_qq() +
15     geom_qq_line() +
16     ggtitle(paste("Shapiro-Wilk p-value:", signif(pval, 3))) +
17     theme_classic()+
18     scale_x_continuous(expand = c(0,0)) +
19     scale_y_continuous(expand = c(0,0))
20   gridExtra::grid.arrange(g1, g2, ncol=2)
21 }
22
23 # Example
24 data_array <- rnorm(100)
25 Normality_test_plot(data_array, "Normal distribution")
26

```

出来上がるプロットの部分があまり綺麗ではないし、軸のラベルも正しくなかったので、以下のように直してみまし

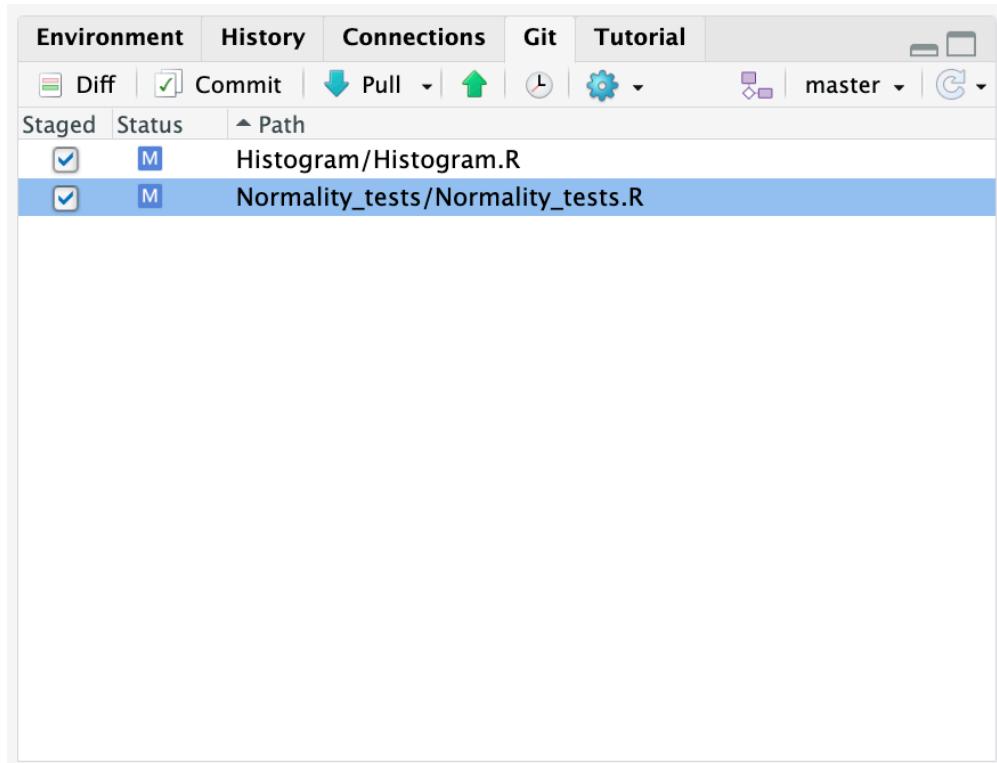
た。

The screenshot shows the RStudio interface. The code editor pane contains the `Normality_tests.R` script. The script defines a function `Normality_test_plot` that creates two ggplots: `g1` (a histogram) and `g2` (a Q-Q plot). It uses `gridExtra::grid.arrange` to combine them into a single grid. The `git` pane at the bottom indicates that the file has been staged for commit.

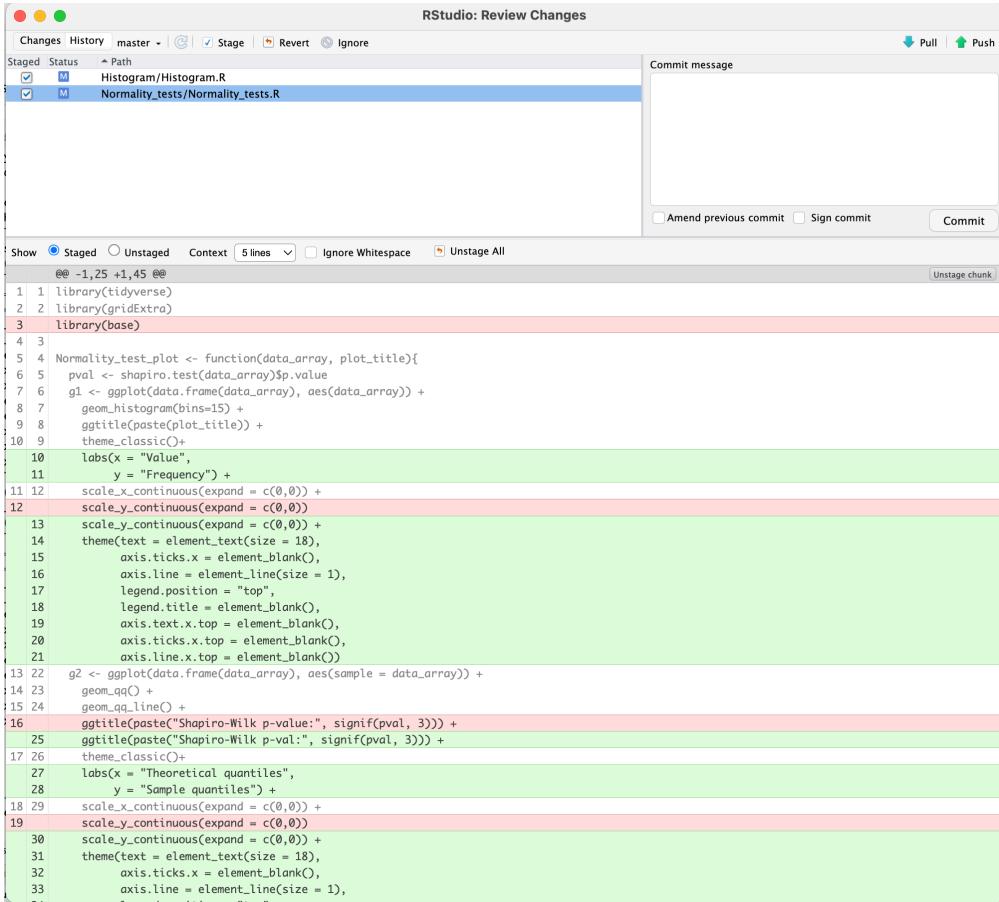
```
library(tidyverse)
library(gridExtra)
Normality_test_plot <- function(data_array, plot_title){
  pval <- shapiro.test(data_array)$p.value
  g1 <- ggplot(data.frame(data_array), aes(data_array)) +
    geom_histogram(bins=15) +
    ggtitle(paste(plot_title)) +
    theme_classic()+
    labs(x = "Value",
         y = "Frequency") +
    scale_x_continuous(expand = c(0,0)) +
    scale_y_continuous(expand = c(0,0)) +
    theme(text = element_text(size = 18),
          axis.ticks.x = element_blank(),
          axis.line = element_line(size = 1),
          legend.position = "top",
          legend.title = element_blank(),
          axis.text.x.top = element_blank(),
          axis.ticks.x.top = element_blank(),
          axis.line.x.top = element_blank())
  g2 <- ggplot(data.frame(data_array), aes(sample = data_array)) +
    geom_qq() +
    geom_qq_line() +
    ggtitle(paste("Shapiro-Wilk p-val:", signif(pval, 3))) +
    theme_classic()+
    labs(x = "Theoretical quantiles",
         y = "Sample quantiles") +
    scale_x_continuous(expand = c(0,0)) +
    scale_y_continuous(expand = c(0,0)) +
    theme(text = element_text(size = 18),
          axis.ticks.x = element_blank(),
          axis.line = element_line(size = 1),
          legend.position = "top",
          legend.title = element_blank(),
          axis.text.x.top = element_blank(),
          axis.ticks.x.top = element_blank(),
          axis.line.x.top = element_blank())
  gridExtra::grid.arrange(g1, g2, ncol=2)
}
# Example
data_array <- rnorm(100)
Normality_test_plot(data_array, "Normal distribution")

```

ファイルを保存すると、右上のGit paneが更新されて、 `Normality_test.R`の更新が追跡されていることがわかります。あとはチェックボックスをチェックしてステージにあげます。



コミットボタンを押すと



RStudio: Review Changes

Changes History master - Stage Revert Ignore

Staged Status Path

M Histogram/Histogram.R

M Normality_tests/Normality_tests.R

Show Staged Unstaged Context 5 lines Ignore Whitespace Unstage All

@@ -1,25 +1,45 @@

```
1 1 library(tidyverse)
2 2 library(gridExtra)
3 library(base)
4 3
5 4 Normality_test_plot <- function(data_array, plot_title){
6 5   pval <- shapiro.test(data_array)$p.value
7 6   g1 <- ggplot(data.frame(data_array), aes(data_array)) +
8 7     geom_histogram(bins=15) +
9 8     ggtitle(paste(plot_title)) +
10 9     theme_classic()+
11 10   labs(x = "Value",
12 11     y = "Frequency") +
13 12   scale_x_continuous(expand = c(0,0)) +
14 13   scale_y_continuous(expand = c(0,0)) +
15 14   theme(text = element_text(size = 18),
16 15     axis.ticks.x = element_blank(),
17 16     axis.line = element_line(size = 1),
18 17     legend.position = "top",
19 18     legend.title = element_blank(),
20 19     axis.text.x.top = element_blank(),
21 20     axis.ticks.x.top = element_blank(),
22 21     axis.line.x.top = element_blank())
23 22 g2 <- ggplot(data.frame(data_array), aes(sample = data_array)) +
24 23   geom_qq() +
25 24   geom_qq_line() +
26 25   ggtitle(paste("Shapiro-Wilk p-value:", signif(pval, 3))) +
27 26   ggtitle(paste("Shapiro-Wilk p-val:", signif(pval, 3))) +
28 27   theme_classic()+
29 28   labs(x = "Theoretical quantiles",
30 29     y = "Sample quantiles") +
31 30   scale_x_continuous(expand = c(0,0)) +
32 31   scale_y_continuous(expand = c(0,0)) +
33 32   theme(text = element_text(size = 18),
34 33     axis.ticks.x = element_blank(),
35 34     axis.line = element_line(size = 1),
```

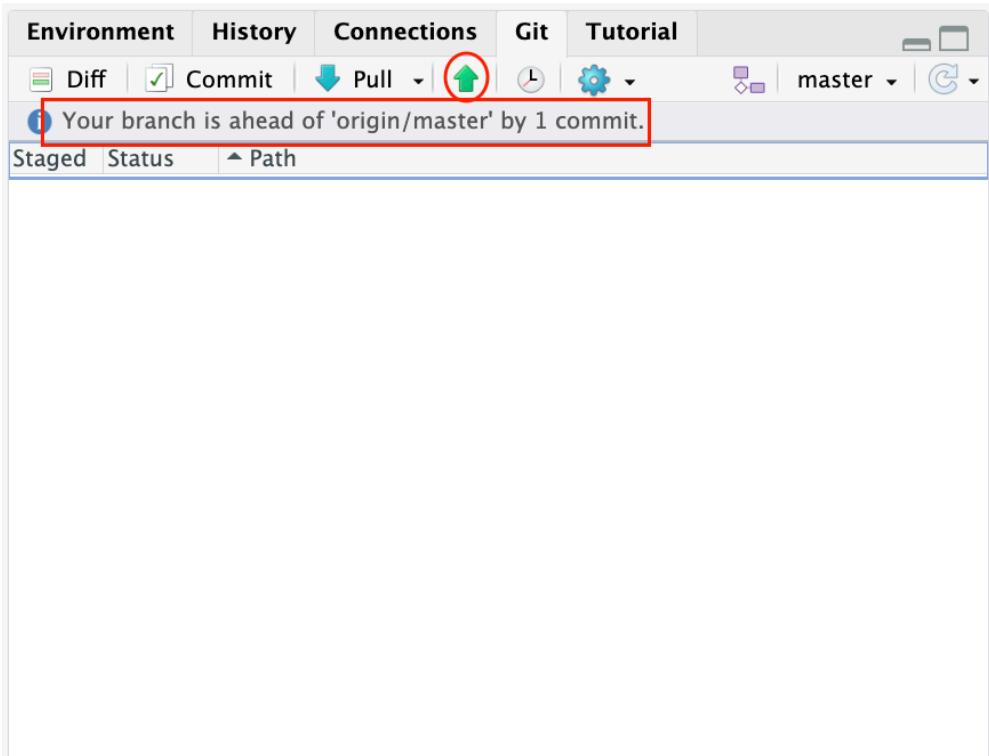
以上のようにどのような変更がなされたのかを示してくれます。

ここで適当にコミットメッセージを打ったコミットしましょう。

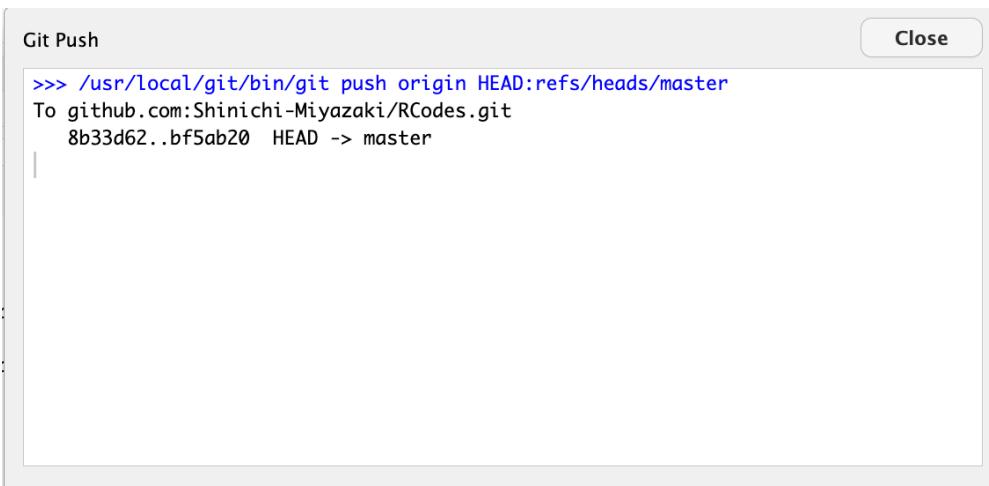
最後にプッシュ

コミットしただけでは、リモートのリポジトリは更新されません。忘れずにプッシュしましょう。

コミットがうまくいっていれば、Git paneに下の赤い四角で囲ったようなメッセージが出ていると思います。これはコミットしたけどリモートに反映されていないものが1塊ありますよ というような意味です。



赤丸で囲った上矢印がPushのボタンなので、こちらを押しましょう。



こんな感じで、出ていればOKです。

お疲れ様です。これでコードを最新版にして、編集して更新という一連の流れができました！。試しにリモートで、先ほどの変更が保存されているかみてみましょう。

まずはリモートで先ほどのリポジトリに行ってみます。

R Codes Private

master 1 Branch 0 Tags

Shinichi-Miyazaki modify plot appearance bfb5ab20 - 5 minutes ago 8 Commits

- .idea Initial commit 2 months ago
- FoQ_trace_plot add info 2 weeks ago
- Histogram modify plot appearance 5 minutes ago
- Normality_tests modify plot appearance 5 minutes ago
- .gitignore add 3 weeks ago
- RCodes.Rproj test commit last month
- README.md Change readme 2 months ago

README

About R codeの保存用レポジトリ

- Readme
- Activity
- 0 stars
- 1 watching
- 0 forks

Releases No releases published Create a new release

Packages No packages published Publish your first package

右の方のCommitを押してみましょう。

Commits

master

Commits on Jul 9, 2024

modify plot appearance bfb5ab20

Shinichi-Miyazaki committed 6 minutes ago

Commits on Jun 25, 2024

add info 8b33d62

先ほどのコミットがきちんと記録されています。中身を見てみると

modify plot appearance

change plot axis
change labels

master

Shinichi-Miyazaki committed 6 minutes ago

Showing 2 changed files with 25 additions and 4 deletions.

Filter changed files

Histogram/Histogram.R

```

1 52 52 @@ -62,3 +52,4 @@
2 53 53     axis.ticks.x.top = element_blank(),
3 54 54     axis.line.x.top = element_blank()
4 55 +

```

Normality_tests/Normality_tests.R

```

1 28 ... @@ -1,25 +1,46 @@
2 1  library(tidyverse)
3 2  library(gridExtra)
4 3 - library(base)
5 4
6 5  Normality_test_plot <- function(data_array, plot_title){
7 6    pval <- shapiro.test(data_array)$p.value
8 7    g1 <- ggplot(data.frame(data_array), aes(data_array)) +
9 8      geom_histogram(bins=15) +
10 9        optitle(paste(plot_title)) +
11 10       theme_classic()+
12 11       labs(x = "Value",
13 12         y = "Frequency") +
14 13       scale_x_continuous(expand = c(0,0)) +
15 14       scale_y_continuous(expand = c(0,0))

```

このようにどのような変更がされたのかをみることができます。

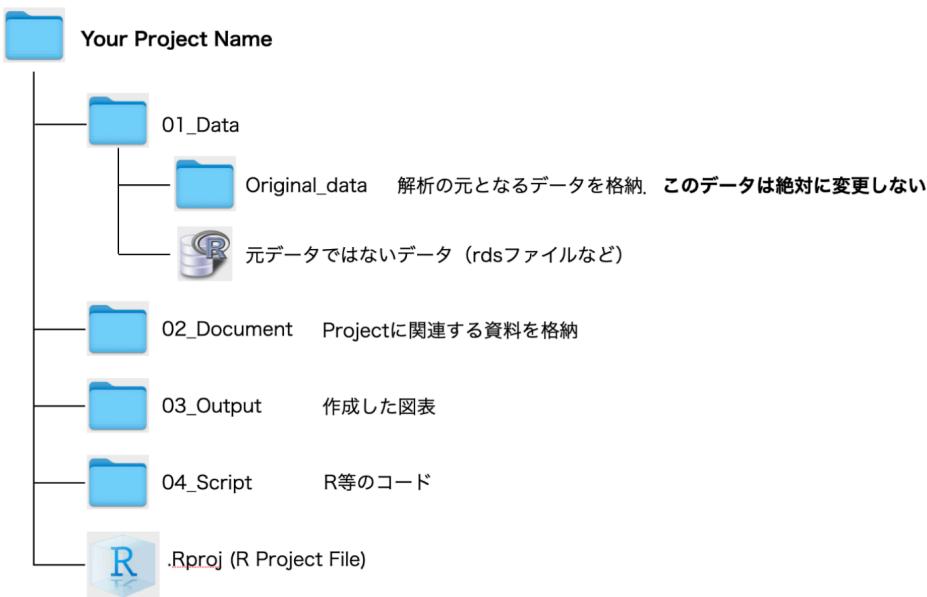
特定のコードでうまく行っていたのに、次の日に少しいじったらうまくいかなくなったりといったことが生じたとしても、この機能があればどんな変更を自分が施したのかわかるので解決の糸口になります。また以前のバージョンに戻すのも簡単にできます。

少しは便利さが伝わるでしょうか？

§2-3 RStudio特有の問題 .gitignoreによる追跡の除外

ここでは少しRStudioに特有の問題について書いていきます。 RStudioのプロジェクトは大変便利な機能です。特にワーキングディレクトリが固定されているのが大変わかりやすいです。 ただRStudioを普通に使っていてGitHubを導入すると少し困ったことがあります。

時折みるRStudioのフォルダ構成に以下のような構成があります。



要するにRProjectのフォルダ内にデータも入れる構成です。 この構成のメリットとして、RProjectは開いた段階でカレントディレクトリがプロジェクトのディレクトリになるので、Dataを相対パスで指定しやすいという点があります。 またデータとRコードを近くに置いておくことで、どちらも迷子にならないという点もメリットです。

一方でプロジェクト自体をGitHubで管理すると以下の2点の問題が起こります。

1. Githubは大きいサイズ (>100MB) のファイルをサポートしていないので、大きい生データだった場合にエラーが起る。
2. 生データが外部に出る可能性は0.1%でも避けたいので、とにかくアップロードしたくない。

これを避けるためには、Gitに対して「このファイルは追跡しないでくださいね」というのを教える必要があります。それが.gitignoreです。

.gitignoreはリポジトリを作成する際に一緒に作成するかどうか聞かれるので、必要だと思ったら次からリポジトリを作る際に設定しておくといいと思います。

	Name	Size	Modified
..	~/Documents/GitHub/RCodes		
<input type="checkbox"/>	README.md	131 B	Jun 13, 2024, 9:59 PM
<input type="checkbox"/>	RCodes.Rproj	205 B	Jul 8, 2024, 11:43 PM
<input type="checkbox"/>	Normality_tests		
<input type="checkbox"/>	Histogram		
<input type="checkbox"/>	FoQ_trace_plot		
<input type="checkbox"/>	Data		
<input type="checkbox"/>	.Rhistory	10.9 KB	Jul 8, 2024, 11:43 PM
<input type="checkbox"/>	.RData	6.3 KB	Jun 25, 2024, 9:17 PM
<input type="checkbox"/>	.gitignore	45 B	Jul 9, 2024, 9:25 PM

.gitignoreでgitの追跡から外すディレクトリやファイルを指定するには以下のような方法があります。

1. フォルダとその中身を無視する。 フォルダ名/ と記載します。

```

1 .Rproj.user
2 .Rhistory
3 .RData
4 .Ruserdata
5 Data/

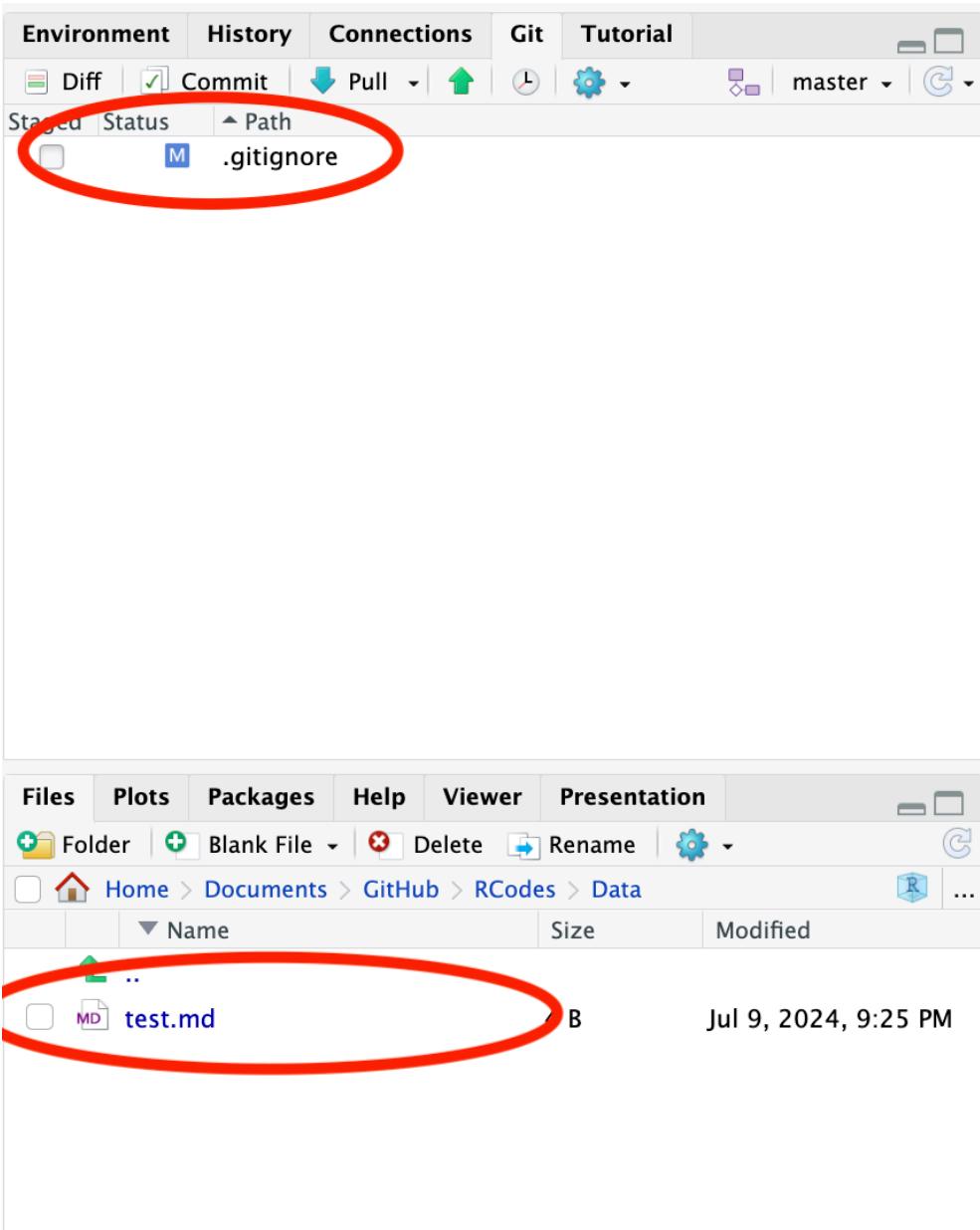
```

この例ではData フォルダと内部のファイルを全て無視します。

2. 特定の拡張子を無視する。 例えばcsvファイルをすべて無視する場合には .csv と記載します。上記の例では.Rhistoryという履歴ファイルなどが無視されています。

他にもたぶん書き方が色々あるので、必要に応じて調べてみてください。

さて今回私はDataフォルダ内にtest.mdというファイルを作ってみました。通常であればGit pane内に変更履歴が出ているはずです。



上記の通り、下のファイルPane

ではtest.mdが現れていますが、Git Paneではtest.mdが現れていないのがわかります。

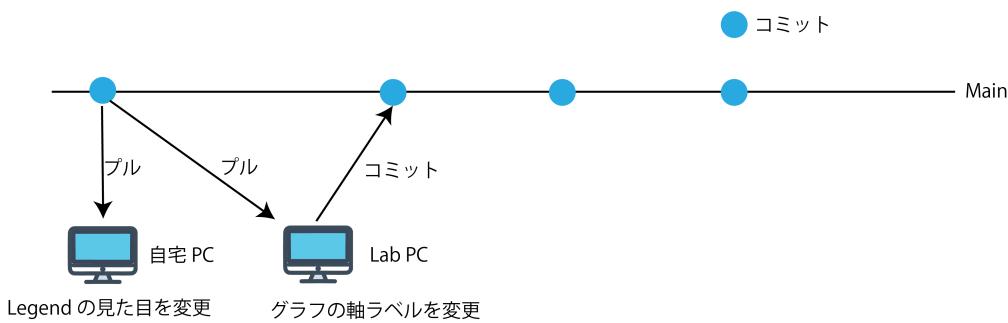
うまくDataディレクトリの内部を追跡しないようにできています。

みなさんも.gitignoreをうまく使って、安全にコードやファイルの変更履歴を追跡しましょう！

§2-4 初めてのブランチ 家とラボの両方でコードを書く

ここではブランチについて説明します。正直ブランチは一人GitHubではあまり使わないのですが、いくつかのPCを併用して開発を行う際には便利かもしれません。

ブランチがあると何が嬉しいのか。



ブランチというのは本来並行作業を円滑に行うためにあります。

例えばですが、箱ひげ図を作ってくれるスクリプトを書いているとします。あるとき、自宅のPCで開発をしようとして、いつものようにリモートからプルして、凡例 (Legend) の見た目を変更したとしましょう。その時に、時間がなくて最後まで終わらず、家を出なくてはいけなかったとします。

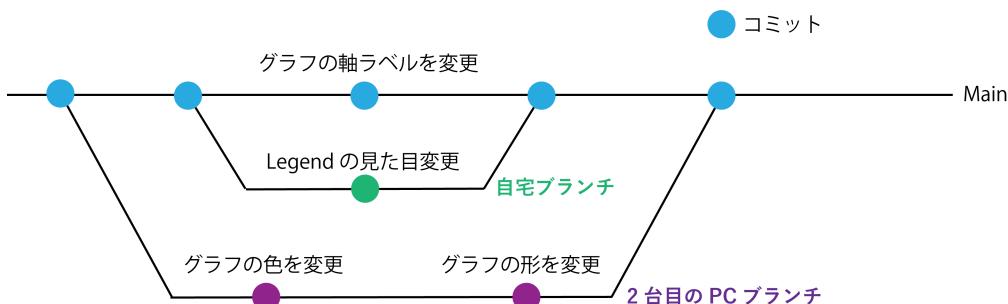
こうなると自宅PCのローカルには途中まで更新したコードがありますが、そのコードはまだリモートには反映されていないことになります。

その後、ラボについてから今度はグラフの軸ラベルを変更したくなつたとしましょう。またプルして変更しますが、この際にプルされるのは自宅でプルしたのと同じコミット (下の図の青丸) になります。ラボではたくさん時間があったのでなんとか軸ラベルを直してコミットしたとします。この時点でリモートにはラボのPCで行った軸ラベルの変更しか記録されていません。

さてこの場合、家に帰ってLegendの続きを直してコミットした場合、何が起こるでしょうか。通常Gitは前回のプルした内容からの変更履歴 (差分) を記録します。しかしこのケースではコミットしてプッシュする際に、差分 (自宅PCでの作業記録) のほかにも差分があることになります。今回のケースのようにコードの違う部分だけが更新されていればそんなに問題になりませんが、仮に同じ部分を変更していると問題になります。(コンフリクトと言います。後に解説します。)

今回の問題は色々な原因があります。極端に言えば作業ごとに必ずコミットしてプッシュしていれば問題は起こりません。(実際に私はこれに近い運用をしています) これが個人開発でブランチがいらないかもしれない所以です。

しかしいつかの端末で同時に開発を行いたい場合には、そもそもいきません。そのような場合にはブランチを使って解決することが可能です。簡単に言えばコミット先をいくつか別に作ってあげることで解決します。



ブランチを使った場合、上記のようになります。2台目のPCや自宅のPCで作業する際には、必要なブランチを作成してその中で変更を行なってコミットしていきます。いい感じに開発がひと段落したら、元のMainにマージ (コードの最新版同士を合わせること) を行なってMainのコードを最新版に更新します。

実際にブランチを使ってみる。

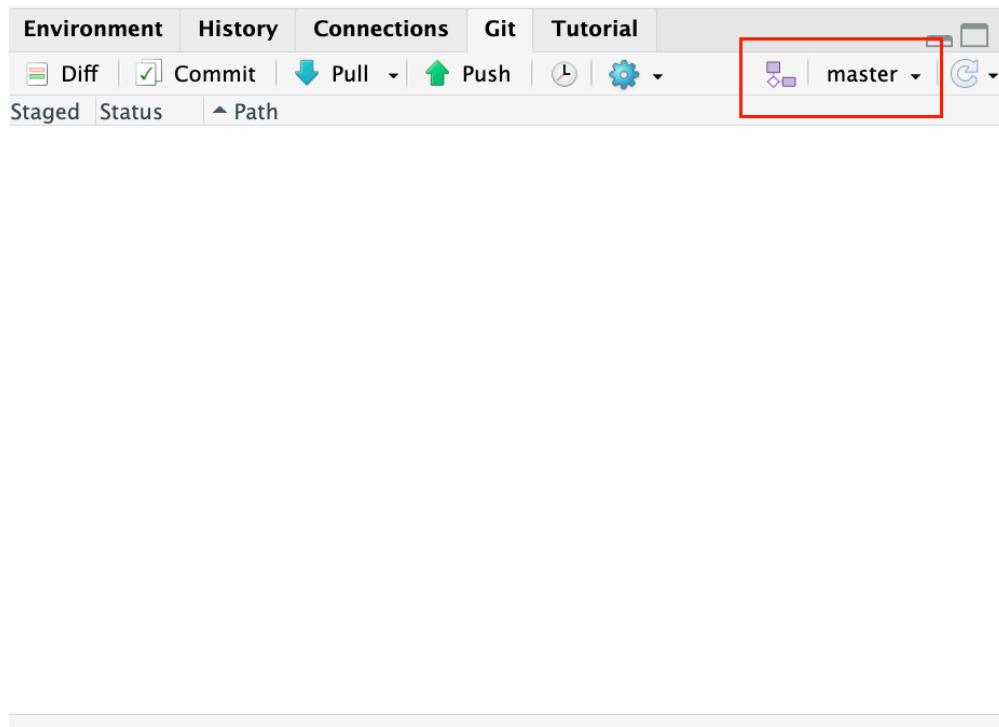
それでは実際にブランチを使って作業してみます。

RStudioではGit paneの右上の方にブランチ関連がまとまっています。

右の"master"と書いてあるのは今自分がいるブランチです。

特別なことをいなければ"Main"か"Master"にいるとと思います。

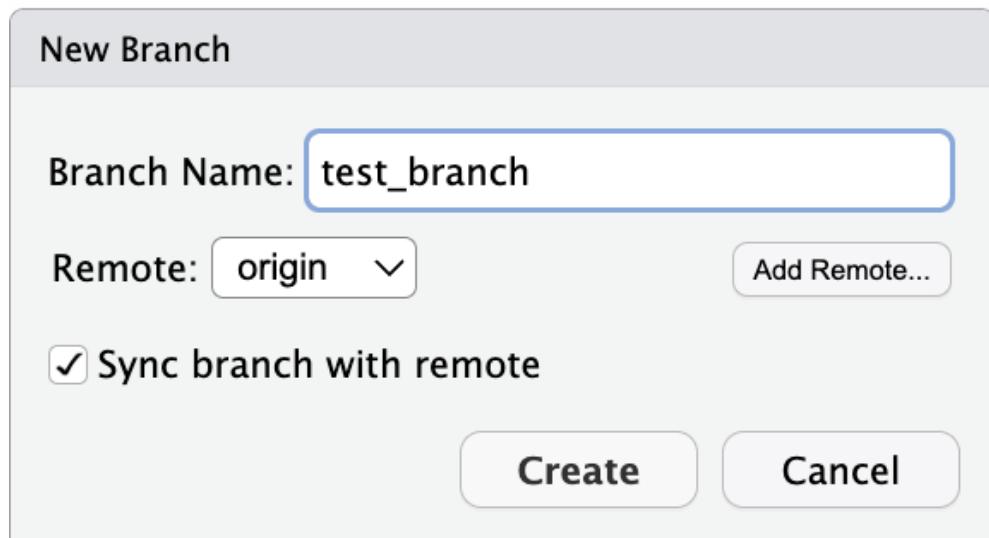
(MainとMasterがあるのは、元々Masterという名前だったデフォルトを、最近になってMainに変えたからです。要はどちらでも大丈夫です。)



1. 新しいブランチを作ってみる。

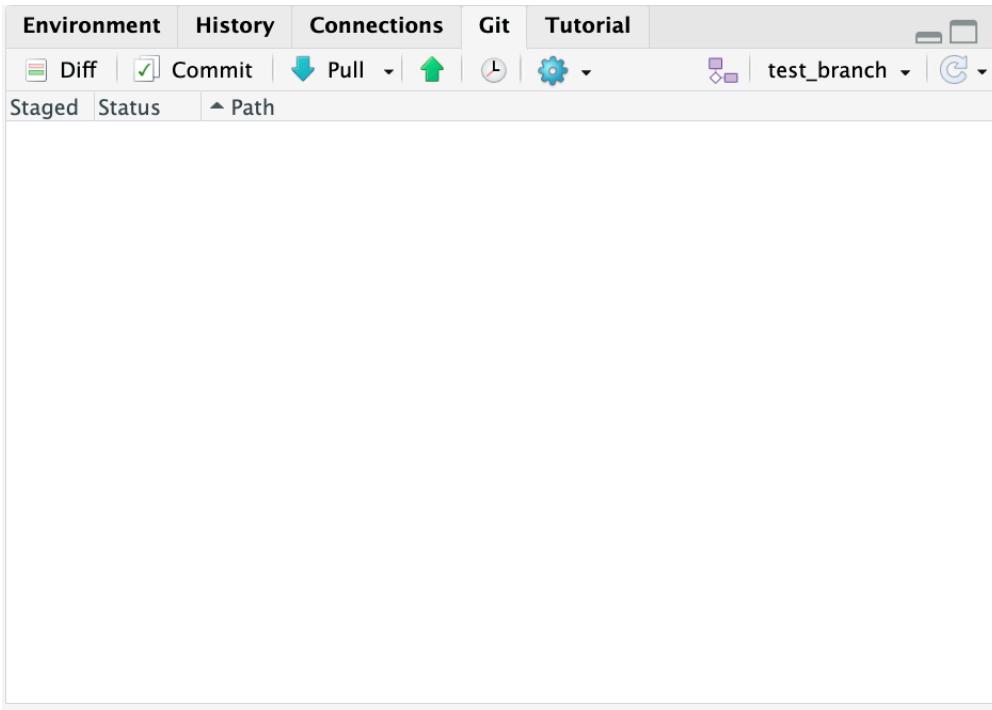
ブランチを作る際には先ほどの画像の"master"の左側にあるアイコンをクリックしましょう。

クリックすると以下のようないい像が出てくると思います。



適当にブランチ名を書いたら、

Create というボタンを押してみましょう。

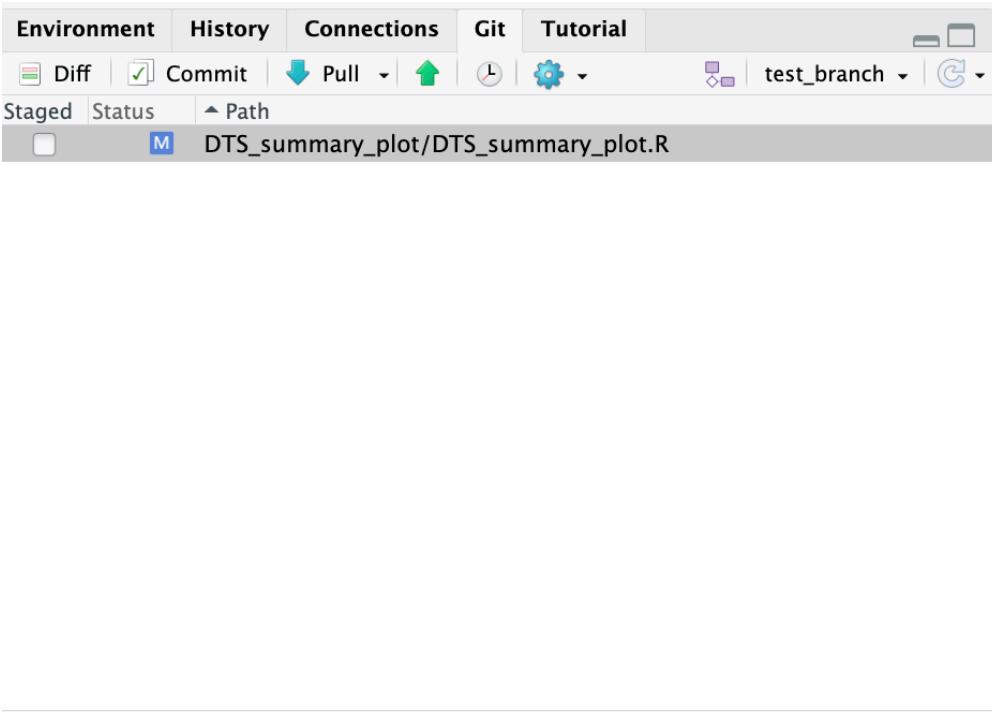


うまくいくと先ほどまで"master"と書いてあったところが先ほど指定した名前に変わると思います。
これでブランチの作成と移動(チェックアウトと言います)が完了です。

2. ブランチで変更してコミットしてみる。

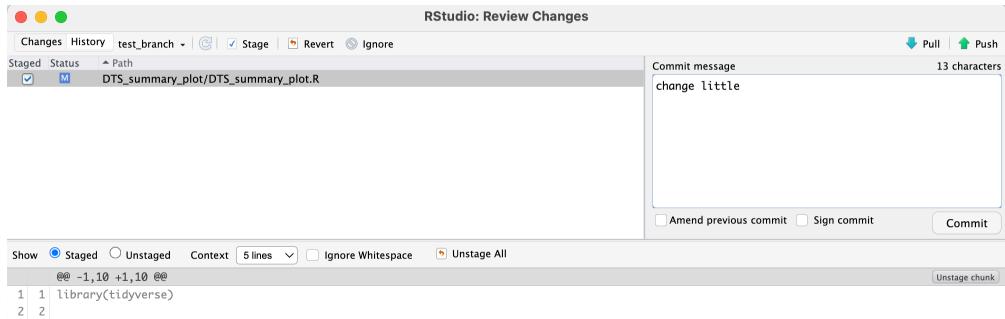
それでは適当にファイルを変更してみましょう。この辺りの流れは以前と同じです。

ファイルを変更したらGit Panelに変更したファイルが出てきます。



そしたらStagedというところにチェックをして、コミットを押します。コミットメッセージもきちんと書きましょう

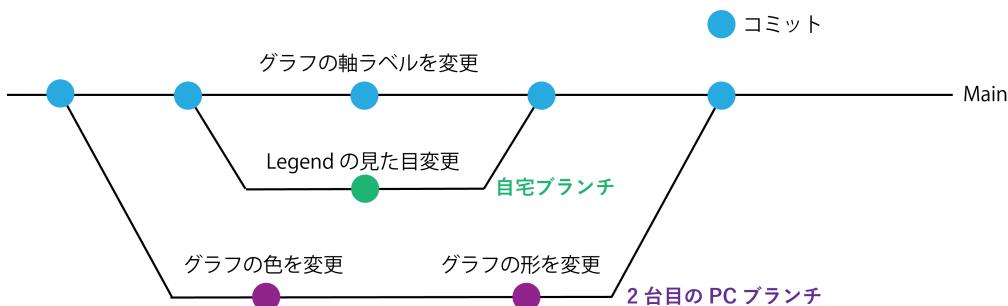
(写真のように適当ではダメです！)



コミットが終わったらPushを押して終了です。お疲れ様です。普段の作業と変わらないで簡単ですね！

3. マージする。

ここからが普段と少し異なる部分です。



この図でいうところの、自宅ブランチからMainへ戻す部分はマージ と呼ばれます。ここではMainの最新版と自宅ブランチの最新版を比較して、両方の最新のところを取ってきたものをMainの次の最新版として記録する ということが行われています。

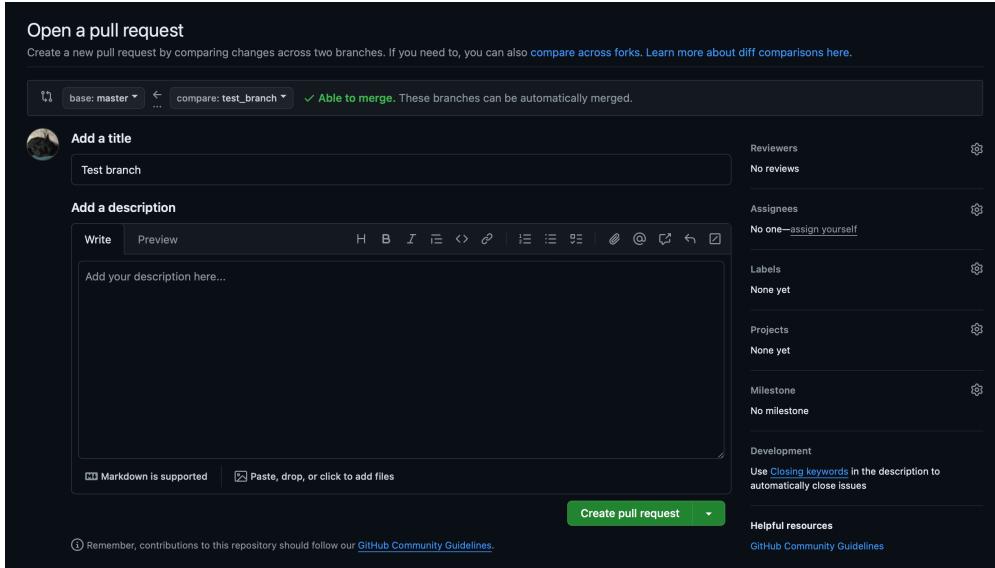
これをやってみましょう。マージはGitHubのwebページでできます（RStudioではできないようです。）

先ほどブランチを作ったりしたリポジトリをGitHubからみてみると以下のように表示されていると思います。

The screenshot shows a GitHub repository named 'RCodes' (Private). It displays a pull request from 'test_branch' to 'master'. The pull request message says 'test_branch had recent pushes 2 minutes ago' and has a 'Compare & pull request' button. Below it, the repository stats show 'master', '2 Branches', and '0 Tags'. There are three commits listed: 'Shinichi change fig size' (02432a6 · last week · 17 Commits), '.idea Initial commit' (2 months ago), and 'DTS_summary_plot add summary plot function' (3 weeks ago). Navigation buttons include 'Go to file', 'Add file', and 'Code'.

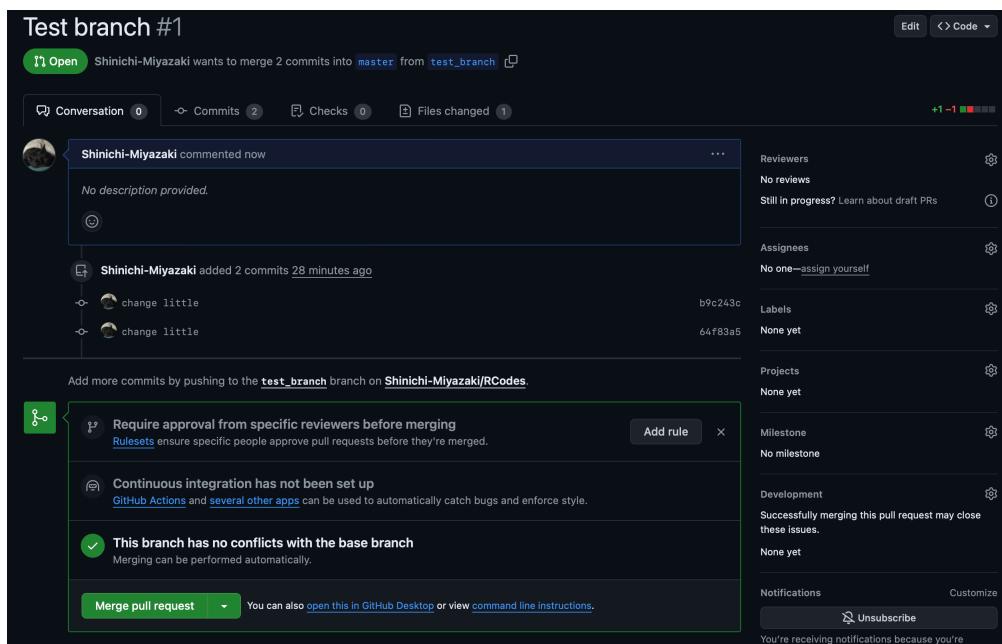
右上のCompare & Pull request というボタンを押しましょう。（Pull requestというのはMergeのためのリクエストのようなものです。本来は共同開発で使う機能ですので、シチュエーションとしては“こんな変更したけどマージしていいかい？”というリクエストをオーナーに送っている というような感じです。今回の場合にはリクエストする側もオーナーも自分なので一人芝居）

以下のようにプルリクエストの画面に遷移します。 ひとまず何も記載しないでCreate Pull Requestを押してみましょう。



う。

すると以下のようないいなになります。



ここではブランチの内容とMain (Master) の内容をマージしても良いかをチェックしてくれて、その後にマージの緑色のボタンが出てきます。特に問題なければマージを押して完了です。

以上がブランチを作成してコミットし、マージを完了するまでの流れです。

§2-5 コンフリクトに対応してみる。

ここではコンフリクトへの対応を学びます。

せっかくなのでここまで学んだことを試しながら説明していきたいと思います。

以下記載内容 適当なスクリプト (あやめで箱ひげなど) を用意して、コミット、ブランチ作成、コンフリクトなどを図解しながら進めてみる。

§2-5 VSCodeをGitHubと連携してみる。

§3 共同開発編

§3-1

Appendix1 パスとディレクトリ

パスとディレクトリ

多分間違っていますが、住所に例えてみます。住所は 日本/茨城/つくば/天王台/1-1-1 みたいに階層構造になっています。日本は茨城を含んでいて、茨城はつくばを含んでいます。こうした時に、パスは住所全体でディレクトリは「日本」とか「つくば」とかの階層の名前を指していると考えると良いかもしれません。

例えば私のPCのデスクトップのパス(住所)をみてみると

```
/Users/miyazakishinichi/Desktop
```

だそうです³。

パスはファイルにも定義できて、デスクトップ上にあるtest1.csvというファイルのパスであれば

```
/Users/miyazakishinichi/Desktop/test1.csv
```

となります。

ディレクトリはさっきの例で言うと、 Users miyazakishinichi Desktop が該当します。test.csvはファイルを内包できないのでディレクトリではないです。

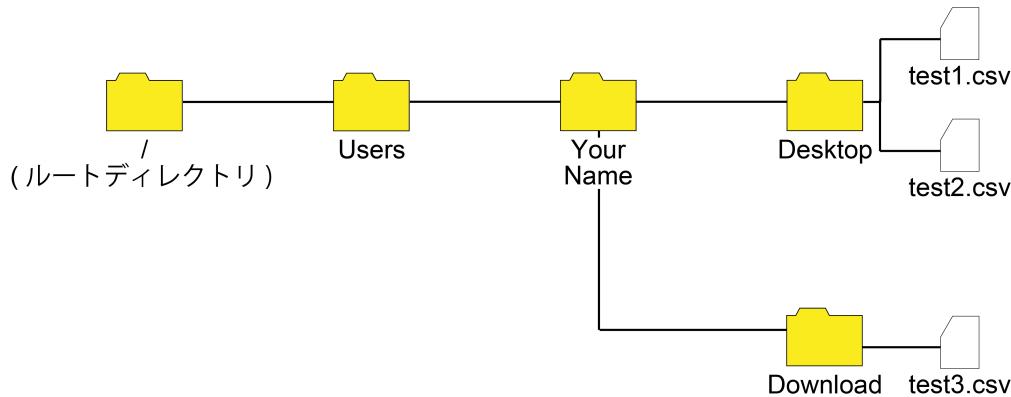
絶対パスと相対パス

パスには実は2種類あって、絶対パスと相対パスがあります。絶対パスは上記したように一番上のディレクトリ(ルートディレクトリといい"/"で表します。)から順に書いていく方法で、普通の住所のような感じです。これのメリットは、自分が今どこにいようが書き方が変わらない点です。自分がニューヨークにいようが土佐にいようが筑波大学の住所は上記のままです。相対パスは今自分がいる場所(カレントディレクトリと言います)から目的のところまでの行き方を書いたパスです。

パスの書き方では

- .(カレントディレクトリ)

- .. (一つ上の階層のディレクトリ) を示しています。今こんな感じのフォルダ構成になっているとしましょう。



そして今あなたがDesktopディレクトリにいる^{^4}とします。この時にtest1.csvの相対パスはどうなるかというと

`./test1.csv`

となります。`."`はカレントディレクトリ (=Desktop) を示すので、その下のtest1.csvだけ指定してあげれば住所としては十分なわけです。

すなわちtest1.csvのパスは2種類あって

`/Users/miyazakishinichi/Desktop/test1.csv` (絶対パス)
`./test1.csv` (Desktopにいるときの相対パス)

となります。

ここまで読んでお気づきの方もいると思いますが、上記の例だけでは、今いるディレクトリ以下のファイルしか、相対パスでは指定できない と思うかもしれません。そこで出てくるのが`.."` (一つ上の階層のディレクトリ) です。

例えば今Desktopにいるとして、test3.csvの相対パスを指定したい時には以下のように書きます。

`../Download/test3.csv`

最初の`.."`が`"一つ上の階層のディレクトリ"`と言う意味なので、Desktopの上の階層であるYournameを指定しています。YournameはDownloadとDesktopを含んでるので、そのうちのDownloadを指定してあげて、その中のtest3.csvを指定する と言うのが上記の意味になります。

「いる」とは何か

大体のコンピュータ作業において、「今自分がどこにいるか」と言うのは大事です。普段はあまり意識しないかもしれませんのが、git使ったりR, pythonを使っていくなら知っておいたほうがいいと思います。

今どこにいるか はいくつかの方法で知ることができます。

ターミナルやGitBashでは以下のコマンドです。

```
pwd
```

RStudioを使っているならコンソールで以下を打ってみましょう。

```
getwd()
```

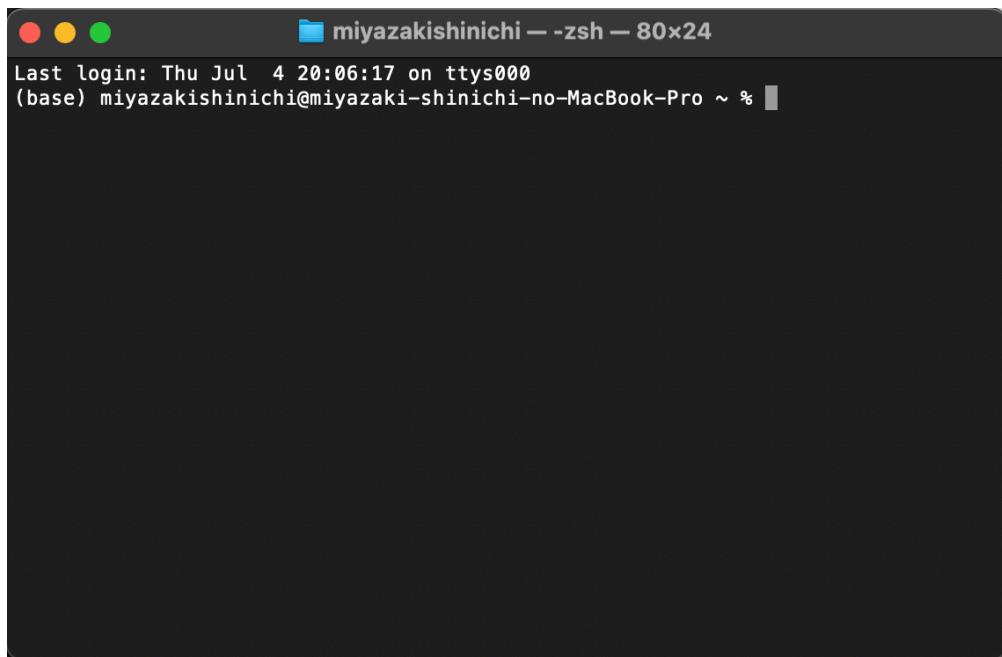
PythonではIDEのコンソールで以下を打ってみましょう。

```
import os  
os.getcwd()
```

これで「今どこにいるか」がわかるようになりました。この「今どこにいるか」をカレントディレクトリと言います。

ホームディレクトリとルートディレクトリ

もう一つの概念としてホームディレクトリがあります、これはエディタなどを開いた際に最初に「いる」ディレクトリのことです。例えばターミナル(GitBash)を新しく開くと以下のようないい出ると思います。



```
miyazakishinichi -- zsh -- 80x24  
Last login: Thu Jul 4 20:06:17 on ttys000  
(base) miyazakishinichi@miyazaki-shinichi-no-MacBook-Pro ~ %
```

この状態で今いる場所を尋ねると

```
Last login: Thu Jul  4 20:06:17 on ttys000
(base) miyazakishinichi@miyazaki-shinichi-no-MacBook-Pro ~ % pwd
/Users/miyazakishinichi
(base) miyazakishinichi@miyazaki-shinichi-no-MacBook-Pro ~ %
```

このようにUsers/miyazakishinichiにいますよ と答えてくれます。この最初に開いた場所をホームディレクトリ と言います。またパスの書き方ではホームディレクトリを"~"で表します。 ですので例えばホームディレクトリの下に Desktopがあるなら

```
~/Desktop
```

でデスクトップのパスを表すことができます。

もう一つの概念としてルートディレクトリがあります。これはパソコン内で最も上位に位置する階層のことです。パスの書き方では"/"で表します。 これが便利なのはディレクトリ移動の時です。ディレクトリ移動はターミナルやGitBashで

▶ cd (行きたいパス)

と書きます。 それでは練習として、

1. ターミナルを開き
2. デスクトップに移動する

▶ cd /Users/Yourname/Desktop

3. デスクトップからDownloadフォルダ内のどこかのフォルダに移動してみる。

▶ cd ../Download/testDir

4. ルートディレクトリに戻る。

▶ cd /

5. もう一度デスクトップに移動

▶ cd /Users/Yourname/Desktop

6. ホームディレクトリに戻る。

▶ cd ~

できましたでしょうか？ホームディレクトリがルートディレクトリの人は多分同じ場所に戻ると思います。

Appendix2 そもそもGitとGithubとは？

(最初に読むには難しいので飛ばして良いです)

Gitは、ソフトウェア開発におけるバージョン管理システムの一つで、プロジェクトのソースコードの変更履歴を管理するため使用されます。GitHubは、Gitのリポジトリをホスティングするウェブベースのサービスです。プロジェクトのコラボレーションとコード共有に特化しており、開発者がプロジェクトを公開または非公開で管理できるプラットフォームを提供します。GitHubは、プルリクエストやイシュー トラッキングなどの機能を通じて、チーム内のコミュニケーションとコードレビューのプロセスを簡素化します。

GitとGitHubの組み合わせは、ソフトウェア開発プロジェクトの効率的な管理と協力を可能にします。Gitがバージョン管理と分散作業のサポートを提供する一方で、GitHubはこれらのプロセスをよりアクセスしやすく、より協力的なものにします。

GitやGitHubを使用することで、このプロセスが大幅に簡単になります。これらのツールを使うことで、以下のような利点があります：

1. **変更履歴の全記録**：過去のあらゆるバージョンから特定の変更点を簡単に確認できます。
2. **チームでの協力**：複数人で同じプロジェクトに取り組む際に、互いの作業を上書きすることなく、効率的に作業を進めることができます。
3. **ブランチ機能**：新機能の開発やバグ修正をメインのプロジェクトから分離して作業することができ、安定したバージョンを保ちながら開発を進めることができます。
4. **マージ機能**：異なるブランチの変更を統合することができ、スムーズにプロジェクトを一つにまとめることができます。
5. **レビュープロセス**：GitHubのプルリクエスト機能を使用することで、コードの変更を他の人がレビューし、フィードバックを提供する前にマージすることができます。

これらの機能により、個人でもチームでも、より効率的に、より安全にプロジェクトを進めることができます。GitとGitHubは、現代のソフトウェア開発において不可欠なツールとなっています。

Appendix3 Markdown

Markdownは、プレーンテキストで記述した文書をHTMLに変換できる軽量マークアップ言語です。

特徴

シンプルで読みやすい 記述が簡単 HTMLよりも直感的に記述できる 様々なプラットフォームで利用可能 用途

ブログ記事 READMEファイル チートシート 資料作成 コミュニケーションツール

基本的な記法

見出し

```
# これは1つ目の見出し  
## これは2つ目の見出し  
### これは3つ目の見出し
```

太字

****太字****

斜体

** 斜体**

箇条書き

- * 箇条書き1
- * 箇条書き2
- * 箇条書き3

番号付きリスト

1. 番号付きリスト1
2. 番号付きリスト2
3. 番号付きリスト3

リンク

[リンクテキスト](<https://www.example.com>)

画像

![画像の説明](<https://www.example.com/image.jpg>)

Markdownのメリット

シンプルで読みやすい Markdownは、プレーンテキストベースなので、シンプルで読みやすいのが特徴です。HTMLと比べて記述量も少なく、初心者でも簡単に習得することができます。

記述が簡単 Markdownは、装飾記号と呼ばれる記号を使って文章を装飾します。装飾記号は直感的に理解できるものが多いので、記述が簡単です。

HTMLよりも直感的に記述できる Markdownは、HTMLよりも直感的に記述できるのが特徴です。HTMLでは、タグを使って文章を装飾する必要がありますが、Markdownでは装飾記号を使って装飾するので、タグを覚える必要がありません。

様々なプラットフォームで利用可能 Markdownは、様々なプラットフォームで利用可能です。ブログサービス、Wiki、エディタなど、多くのツールでMarkdownに対応しています。

Markdownのデメリット

表現力が制限される Markdownは、HTMLと比べて表現力が制限されます。複雑なレイアウトや装飾には、HTMLの方が適しています。

プレビュー機能がないエディタもある Markdownエディタの中には、プレビュー機能がないものがあります。プレビュー機能がないと、記述内容を確認しながら作業することができません。

まとめ

Markdownは、シンプルで読みやすく、記述が簡単な軽量マークアップ言語です。ブログ記事やREADMEファイル、チートシートなど、様々な用途に利用することができます。

Markdownを習得することで、文章をより効果的に表現することができます。ぜひ、Markdownを活用してみてください。