Q1
Gitとはなにか？

A1
バージョン管理を行うためのシステムのひとつ

Gitを導入することで以下２つのメリットが得られる
①簡単にバージョン管理を行うことができるようになる
②複数人での共同作業、チーム開発がやりやすくなる

Q2
バージョン管理とはなにか？

A2
ファイルを以前の状態に戻せるように、
ファイルの変更の履歴を保存・管理しておく仕組みのこと
バージョン管理は、Gitのようなバージョン管理システムによって行う

バージョン管理システムを用いてバージョン管理を行うと、
必要に応じてファイルを過去の状態に戻すことができるだけでなく、
「いつ、誰が、どのファイルに、どのような変更を加えたのか」を
記録・閲覧することもできる

代表的なバージョン管理システムとしては、
分散型のGitや、集中型のSubversionなどがある

Q3
バージョン管理システムを使わずに、
プログラムなどのテキストファイルを編集する場合、
どのような問題が生じるか？

A3
バージョン管理をせずに、
１つのファイルを変更・上書き保存する場合、以下のような問題が発生する
・以前削除した箇所がやはり必要になったとき、手作業で戻す必要がある
・間違った変更を加えてしまったときも、編集前の状態に手作業で戻す必要がある
・手作業で戻すので時間がかかる上、正しく復元できない場合もある

この問題を解決するバージョン管理の方法のひとつとして、
「ファイルの編集前にコピーをとる」という方法がある
例えば、「20211220_app.py」「20220106_app.py」... 、というように

しかしこのバージョン管理の方法でも以下のような問題が発生する
・変更が積み重なっていくうちに、ファイルの数が膨大になり、管理ができなく

なってしまう
・複数人で作業する場合、誰かが命名ルールを間違えてコピーしてしまう

こういった問題はGitのようなバージョン管理システムを使うことで解決できる

Q4
ワークツリーとはなにか？

A4
Gitなどのバージョン管理システムの管理下に置かれたディレクトリのこと
「作業ディレクトリ」「ワーキングツリー」とも呼ばれる

「git init」コマンドを実行したときの、
カレントディレクトリがワークツリーに設定され、
ワークツリー内のファイルやディレクトリがバージョン管理の対象になる

Gitにおいては、ファイルの編集作業はワークツリー内で行い、
ワークツリーのファイルをステージングエリアに追加（add）した上で、
ステージングエリアのスナップショットを、
ローカルリポジトリに保存（commit）することでバージョン管理を行う

ワークツリーとなったディレクトリには、
ローカルリポジトリの実体である「.git」ディレクトリが配置される

Q5
ステージングエリアとはなにか？

A5
ローカルリポジトリにコミットされる前に、一旦ファイルが配置される領域のこと
「インデックス（index）」とも呼ばれる

ワークツリーにあるファイルは、ローカルリポジトリにコミットする前に、
いったんステージングエリアと呼ばれる領域に配置することになっている

なのでファイルは、以下の順番で移動していく
ワークツリー → ステージングエリア → ローカルリポジトリ

ファイルを、ワークツリーからステージングエリアに移すのが「git add」
ステージングエリアからローカルリポジトリに移すのが「git commit」

ステージングエリアの実体は、「.git」というディレクトリの内部にある

ステージングエリアを用いることで、
コミットの内容を局所化しやすくなり、バージョン管理がしやすくなる

Q6
リポジトリとはなにか？

A6
ステージングエリアのスナップショットを記録する領域のこと
コミットを記録する領域とも言える

リポジトリには以下の２種類がある
・ローカルリポジトリ
ユーザーひとりひとりが利用するリポジトリ、手元のマシンそれぞれに配置
ローカルリポジトリの実体は、
ワークツリーに配置される「.git」というディレクトリ

・リモートリポジトリ
複数人で共有するリポジトリ、専用のサーバーに配置したり、
GitHubのようなホスティングサービスを利用して用意する

リポジトリにはコミットが記録されているので、
リポジトリにあるコミットの内容を確認したり、
指定したコミットの状態にワークツリーやステージングエリアを戻すこともできる

Q7
addとは？

A7
ワークツリーのファイルやディレクトリを、ステージングエリアに追加すること

addすることによってはじめて、ワークツリーのファイルやディレクトリを
Gitによるバージョン管理下に置くことができる

「git add ファイル」を実行すると指定したファイルを、
ワークツリーからステージングエリアに追加できる

Q8
commit（コミット）とは？

A8
コミットは以下の２つの意味で用いられる
・ステージングエリアにあるファイルのスナップショットを、ローカルリポジトリに登録すること
・ローカルリポジトリに登録されたそれぞれのスナップショットのこと

「git commit -m "コミットメッセージ"」のコマンドでコミットができる

Q9
ステージングエリアのメリットは？

A9
ステージングエリアがあることによって、コミットの内容を局所化できる

バージョン管理においては、あとからコミット履歴を見たときにわかりやすいように、
１つのコミットには１つの修正しか含めないことが推奨される

例えば、「バグ修正」と「機能追加」という２つの修正を行なっているときに、
それらをまとめて１つのコミットにすると、あとからバージョンごとの差分を確認する際に、
どこが「バグ修正」でどこが「機能追加」なのかわかりづらくなってしまう

１つのコミットには１つの修正を実施しやすくするために、
コミットしたい内容だけをステージングエリアまず入れてから
リポジトリにコミットするという仕組みが取り入れられている

Q10
「GUI」と「CLI」とはそれぞれなにか？

A10
「GUI」：グラフィカルユーザーインターフェース（Graphical User Interface）の略
「CLI」：コマンドラインインターフェース（Command Line Interface）の略

WindowsやmacOSで使われているような、
画面上のアイコンをマウスなどを用いてクリックすることで操作するインターフェースがGUI
Linuxで使われているような、キーボードなどからコマンドを入力し、
文字列として結果を出力するインターフェースがCLI

CLIは直感的な操作はできないものの、以下のようなメリットがある
・自動化がしやすい（すべての操作を文字列で表現できるため）
・慣れればマウス操作よりも作業効率が高い
・GUIよりも変更されづらいので一度覚えればずっと使える

Q11
Gitをはじめる際に、
ユーザー名とメールアドレスを
設定する必要があるのはなぜか？

A11
Gitを使っているユーザーが誰かを明確にするため
共同開発をする際には、誰がどんな変更を加えたか？を知る必要がある

「git log」というコマンドを実行すると、
コミットした人のユーザー名とメールアドレスを確認できる

Q12
Gitを初期化する

A12
```
git init
```

「init」は「initialize（初期化する）」の略

このコマンドによって以下２つが行われる
①カレントディレクトリがワークツリーとして設定される
ここで設定されたワークツリー内に、
Gitでバージョン管理したいファイルを入れる

②カレントディレクトリに「.git」というディレクトリが作成される
このディレクトリがステージングエリアやローカルリポジトリの実体

Q13
ステージングエリアに「example.txt」を追加する

A13
```
git add example.txt
```

「git add ファイル」というコマンドを実行すれば、
指定したファイルをステージングエリアに追加できる

「git add .」というコマンドを実行すれば、
「.」はカレントディレクトリの意味なので、
ワークツリーのすべてのファイルをステージングエリアに追加できる

Q14
ステージングエリアに入っているファイルのリストを表示する

A14
```
git ls-files
```

「ls-files」は「list files」の略

ステージングエリア内のファイルをリストアップする、という意味

ワークツリーの内容がすべてaddされているなら、
ワークツリーのファイルのリストを表示する「ls」コマンドと結果が一致する

Q15
ステージングエリアにあるファイルやディレクトリを、
ローカルリポジトリにコミットする
ただし、コミットメッセージは「Message」とする

A15
```
git commit -m "Message"
```

「-m」は「message」の略

コミットする際に必要な、コミットメッセージを入力するためのオプション

コミットメッセージは、どのような変更を加えたのかわかりやすいものにする

「-m」オプションを指定しないで単に「git commit」すると、
Vimなどの設定されたエディタが開いて、
エディタ上でコミットメッセージの入力ができるようになる

Q16
コミットの履歴を表示する

A16
```
git log
```

以下のような項目が表示される
・コミットID（SHA-1 ハッシュ）
・コミットしたユーザーの名前とメールアドレス
・コミット日時
・コミットメッセージ

例えば、以下のように表示される
```
commit 02a59c97cab6a7gjohd17b7a275c2b1ceba4059
Author: username <username@example.com>
Date:   Tue Jun 29 14:06:50 2021 +0900

    Commit Message
```

Q17
各コミットの統計情報とともに、

コミットの履歴を表示する

A17
`git log --stat`

「`--stat`」は「`statistics`（統計）」の略
ファイルの統計情報も込みで表示する、という意味

「`--stat`」オプションをつけると、コミットごとに
どのファイルに何行の追加・削除があったかという
統計情報を含めてコミット履歴を表示できる

Q18
ワークツリーとステージングエリアの差分を表示する

A18
`git diff`

「`diff`」は「`difference`（違い、差異）」の略
２つのファイルの差分を表示するコマンド

「`git diff`」とするとワークツリーとステージングエリアの差分を表示する
「`git diff` コミット1 コミット2」というように2つのコミットを引数に指定すれば、
コミット間の差分を表示することもできる

「`git diff`」コマンドでの差分の表示には、
`Git`の管理下に置かれていないファイル（`add`していないファイル）は含まれないことに注意

Q19
「`git diff`」を実行すると以下のように出力された
この出力はどのような意味か？

```
diff --git a/example.txt b/example.txt
index 912a645..1121bb3 100644
--- a/example.txt
+++ b/example.txt
@@ -1,3 +1,3 @@
-
+Hello

 World
\ No newline at end of file
```

A19
「git diff」コマンドによって、
ワークツリーとステージングエリアの差分が表示されている

どのような差分があるかは以下のように読み取れる
・「diff --git a/example.txt b/example.txt」
二時点（ワークツリーとステージングエリア）でのexmaple.txtの内容を比較している
ワークツリーが「b/example.txt」、ステージングエリアが「a/example.txt」

・「@@ -1,3 +1,3 @@」
この下に書かれている差分をどのように解釈すべきかが書かれている
この記述の下で、
「-」のファイルの1行目からはじめて3行だけ表示（-1,3）、
「+」のファイルの1行目からはじめて3行だけ表示（+1,3）していることを示す
「-」のファイルとは「--- a/example.txt」なので「a/example.txt（ステージングエリア）」を指し、
「+」のファイルとは「+++ b/example.txt」なので「b/example.txt（ワークツリー）」を指す
具体的な差分がこの下に書かれている

・「@@ -1,3 +1,3 @@」より下
具体的な差分が書かれている
「-」は空行が削除されたこと、
「+Hello」は「Hello」が追加されたことを示す
残り２行は変更がないので、
「+」や「-」をつけずそのまま書かれている

Q20
変更されたファイルを検出してステージングエリアに追加し、
それらをコミットするという動作を一つのコマンドで実行する
ただし、コミットメッセージは「Message」とする

A20
```
git commit -a -m "Message"
```

「-a」は「add」の略
コミットと同時に、ステージングエリアへの追加（add）も行う、
という意味のオプション

「-m」は「message」の略
コミットする際に必要な、コミットメッセージを入力するためのオプション
コミットメッセージは、どのような変更を加えたのかわかりやすいものにする

Q21
ステージングエリアとHEADの差分を表示する

A21
```
git diff --cached
```

```
git diff --stagedでもOK
```

・ワークツリーとステージングエリアの差分
```
git diff
```
・ステージングエリアとHEADの差分
```
git diff --cached
```
・ワークツリーとHEADの差分
```
git diff HEAD
```

Q22
ワークツリーとHEADの差分を表示する

A22
```
git diff HEAD
```

・ワークツリーとステージングエリアの差分

```
git diff
```

・ステージングエリアとHEADの差分

```
git diff --cached
```

・ワークツリーとHEADの差分

```
git diff HEAD
```

Q23
ワークツリー内に、バージョン管理の対象に
なっていないファイルはないか？を調べる

A23
```
git status
```

「git status」は以下３つの情報を調べるためのコマンド
・ワークツリー内に、バージョン管理の対象になっていないファイルはないか？
・ワークツリー内に、ステージングエリアにaddされていない変更はないか？

・ステージングエリア内に、まだローカルリポジトリにコミットされていない変更はないか？

Q24
ワークツリー内に、ステージングエリアに
addされていない変更はないか？を調べる

A24
```
git status
```

「git status」は以下３つの情報を調べるためのコマンド
・ワークツリー内に、バージョン管理の対象になっていないファイルはないか？
・ワークツリー内に、ステージングエリアにaddされていない変更はないか？
・ステージングエリア内に、まだローカルリポジトリにコミットされていない変更はないか？

Q25
ステージングエリア内に、まだローカルリポジトリに
コミットされていない変更はないか？を調べる

A25
```
git status
```

「git status」は以下３つの情報を調べるためのコマンド
・ワークツリー内に、バージョン管理の対象になっていないファイルはないか？
・ワークツリー内に、ステージングエリアにaddされていない変更はないか？
・ステージングエリア内に、まだローカルリポジトリにコミットされていない変更はないか？

Q26
ワークツリーにあるファイルのうち、
すでにバージョン管理されているすべてのファイルの変更を
ステージングエリアに追加する

A26
```
git add -u
```

「-u」は「update」の略

「git add -u」では新しく作成したファイルはaddされないことに注意

Q27
「file」という名前のファイルをGitの管理下から外し、
ステージングエリアから削除する

A27
`git rm file --cached`

このコマンドを実行すると、「`file`」という名前のファイルが
ステージングエリアから削除され、`Git`の管理下から外される

ディレクトリを`Git`管理下から外すには「`-r`」オプションをつける

ワークツリーからも削除したい場合は「`git rm file`」というように、
「`--cached`」オプションを外す

Q28
「`dir`」という名前のディレクトリを`Git`の管理下から外し、
ステージングエリアとワークツリーから削除する

A28
`git rm -r dir`

このコマンドを実行すると、「`dir`」という名前のディレクトリが
ステージングエリアとワークツリーから削除され、`Git`の管理下から外される

「`-r`」は「`recursive`」の略で「再帰的な」の意味
再帰的にディレクトリを削除する

ワークツリーから削除したくない場合は
「`git rm -r --cached dir`」というように、「`--cached`」オプションをつける

まだコミットされていない変更がワークツリーにある場合はエラーになる
強制的に削除したい場合は「`-f`」オプションをつけて実行する
「`-f`」は「`force removal`（強制削除）」の略

Q29
「`.gitignore`」とはどのようなファイルか？

A29
バージョン管理したくないファイルを指定するためのファイル

このファイル内に書いたファイルは「`git status`」を実行しても、
`Git`の管理下に置かれていないファイルとして表示されなくなる

「`.gitignore`」自体もバージョン管理するのが基本

Q30
「.gitignore」という名前のファイルの内容を確認したところ
以下のような記述があった
これはどのような意味か？

```
.DS_STORE
*.log
!example.log
```

A30
「.gitignore」は、バージョン管理したくないファイルを指定するためのファイル

3行指定されているがそれぞれ以下の意味
・1行目「.DS_STORE」
「.DS_STORE」という名前のファイルをバージョン管理から外す

・2行目「*.log」
拡張子が「.log」のファイルをバージョン管理から外す
「*」はワイルドカードと呼ばれ、任意の文字列を表す

・3行目「!example.log」
ただし、「example.log」という名前のファイルはバージョン管理する
「!」は否定の記号なので、バージョン管理から外さない例外のファイルを指定できる

Q31
ワークツリーとステージングエリアにおいて
「file1」というファイルの名前を「file2」に変更する

A31
```
git mv file1 file2
```

「mv」は「move」の略で、
ファイルの移動や名前の変更をするためのコマンド

２つの引数にどちらもファイルを指定すると名前の変更になる
Linuxのコマンドの「mvコマンド」を使ってしまうと、
Gitの管理下から外れてしまうので、
Gitで管理しているファイルは「git mv」コマンドで操作すると良い

基本的にLinuxの「mvコマンド」と使い方は同じ

Q32

ワークツリーとステージングエリアにおいて
「file」というファイルを「dir」というディレクトリに移動する

A32
```
git mv file dir
```

「mv」は「move」の略で、
ファイルの移動や名前の変更をするためのコマンド

2つ目の引数にディレクトリを指定すると、
名前の変更ではなくファイルやディレクトリの移動になる

ファイルの移動に、Linuxのコマンドの「mvコマンド」を使ってしまうと、
Gitの管理下から外れてしまうので、
Gitで管理しているファイルは「git mv」コマンドで移動すると良い

基本的にLinuxの「mvコマンド」と使い方は同じ

Q33
SHA-1 ハッシュとはなにか？

A33
コミットの名前として機能するユニークな文字列のこと
「コミットID」とよばれることもある

「git log」を実行すると表示される
「ca82a6dff817ec66f44342007202690a93763949」のような文字列がSHA-1ハッシュ
先頭部分だけを取り出して「ca82a6d」というような省略形で書かれることもある

SHA-1 ハッシュを指定することで、どのコミットかを特定することができる

Q34
SHA-1 ハッシュが「03ebe2a」のコミットと、
「83hefo8」のコミットの差分を表示する

A34
```
git diff 03ebe2a 83hefo8
```

「git diff ハッシュ値① ハッシュ値②」を実行すると、
指定したSHA-1ハッシュ値のコミットの差分を確認できる

「git diff」コマンドは引数を二つ渡すと、
二つの引数の差分を表示してくれる

Q35
ブランチとは？

A35
ブランチとは、枝分かれした、それぞれのコミットの履歴のこと
「branch」は「木の枝」の意味
ブランチの実体は、一連のコミットの先頭を指すポインタ

ブランチを導入すると以下のようなメリットがある
①チーム開発がやりやすくなる
②コミット履歴がわかりやすくなる

ソフトウェア開発では、新機能を開発しながら、
既存のバグを修正するなど、並行して作業を行うことが多い
このような場合、他の作業に影響を受けない・与えない仕組みが必要になる
そのために用いるのがブランチ

開発の目的ごとにブランチを分けることで、
それぞれ独立して開発が可能になるのでチーム開発がやりやすくなる
さらに、コミット履歴が開発の目的別に分かれて見やすくなる

例えば、
①リリース版のブランチ
②バグ修正のブランチ
③機能追加のブランチ
という３つのブランチを用意すれば、
いつでもリリースできる状態（①）を維持しつつ、
バグ修正のための変更（②）や、新機能追加のための変更（③）も加えられる

ブランチはマージすることによって、
それぞれの最新コミットの内容を１つにまとめることができる

Q36
既存のブランチの一覧を表示する

A36
`git branch`

このコマンドの出力では、
HEADがあるブランチの横に「*」マークがついている

「`git branch` ブランチ名」として実行すれば、
引数に指定した名前のブランチを新規作成できる

Q37
master（main）ブランチとはなにか？

A37
Gitリポジトリを作成したときに自動的に作成されるブランチ

今までは「master」がデフォルトの名前だったが、
人権運動を背景に、今後は「main」に変わっていく
GitHubではすでにmainになっている（2021年8月時点）

master（main）ブランチを、
プロジェクトのメインのブランチ（trunk）として使うことが一般的
メインのブランチでは、いつでもリリースできる状態でバージョンを管理する

Q38
・トランク（trunk）
・トピックブランチ
とはそれぞれなにか？

A38
・トランク（trunk）
プロジェクトのメインとなるブランチで、基本的には常にリリースできる状態にしておくもの

「trunk」は「木の幹（みき）」の意味
メインライン（mainline）、メインブランチとも言う
デフォルトで用意されているmaster（main）ブランチをトランクとして使うことが一般的

・トピックブランチ
機能追加やバグ修正など、あるトピックに関する作業を行うために作成するブランチ

例えば、追加したい機能が大きくて１回のコミットでは対応できない場合、
メインのブランチにコミットしてしまうと、メインのブランチが半端な状態になり、
正しく動作しなくなる恐れがある
このような場合は、機能追加用のトピックブランチを作成して、
そこで開発を行うことで、メインブランチを守りつつ開発を進められるようになる

トピックブランチで開発を終えたら、メインブランチにマージするようにする

Q39
マージとは？

A39
マージとは、２つのブランチを１つに統合すること
より正確に言うと、
２つのブランチそれぞれの最新コミットの内容を統合する、
新しいコミットを現在のブランチに作ること

例えば、新機能追加のブランチでの開発が完成し、正しく動作するようになったら、
機能追加のブランチとリリース版のブランチをマージして統合する

現在作業中のブランチと、`add-func`ブランチをマージするには、以下のように実行する
```
git merge add-func
```

マージが正しく完了したら、
機能追加のブランチのようなトピックブランチは不要になることが多いので、
その場合はトピックブランチを削除する

変更箇所が重複していない場合は、自動でマージが行われるが、
重複している場合はコンフリクトが発生する
コンフリクトが発生したらワークツリーを手動で修正した上で
「`git add`」と「`git commit`」を行う

Q40
「`bug-fix`」という名前のブランチを新規作成する

A40
```
git branch bug-fix
```

ブランチの一覧を表示したいなら引数を渡さずに、
「`git branch`」を実行する

新規作成したブランチにすぐに`checkout`したいなら
「`git checkout -b bug-fix`」とすれば、
「`git branch bug-fix`」と「`git checkout bug-fix`」を
同時に実行することができる

Q41
`master`ブランチから「`bug-fix`」という既存のブランチに切り替える

A41

```
git checkout bug-fix
```

「git checkout」は本来、HEADを移動するコマンド
引数にブランチを指定するとブランチにHEADを移動できる
つまり、ブランチを切り替えることができる

新規作成したブランチにすぐにcheckoutしたいなら
「git checkout -b bug-fix」とすれば、
「git branch bug-fix」と「git checkout bug-fix」を
同時に実行することができる

Q42
今作業をしているmasterブランチと、
「bug-fix」ブランチをマージする

A42
```
git merge add-func
```

masterブランチにHEADがある状態で
「git merge add-func」というコマンドを実行すると、
masterブランチにadd-funcブランチの変更が取り込まれる

つまり、add-funcブランチの先頭のコミットの内容と、
masterブランチの先頭のコミットの内容が組み合わさった、
新しいコミットがmasterブランチに追加される

変更箇所が重複していない場合は、
「git merge」によって自動でマージが行われるが、
重複している場合はコンフリクトが発生する

コンフリクトが発生したらワークツリーを手動で修正した上で

「git add」と「git commit」を行う

Q43
「bug-fix」という名前の既存のブランチを削除する
ただし、「bug-fix」ブランチの最新コミットは、
他のブランチにマージされているものとする

A43
```
git branch -d bug-fix
```

削除するブランチにマージされていないコミットがある場合は、
この書き方では削除できないことに注意

最新コミットがマージされていないブランチを強制的に削除するには、
「-d」オプションの代わりに、「-D」オプションを付けて以下のように実行する
git branch -D 削除するブランチ名

Q44
マージをしようとしたらコンフリクトが発生した
この場合、マージを完了させるためにやるべきことは？

A44
①ワークツリーのファイルを手動で編集してコンフリクトを解消する
②ワークツリーの内容をaddしてcommitする

マージは以下の流れで行われる
①取り込むブランチの最新コミットの内容と、取り込まれるブランチの最新のコミットの内容が組み合わさった内容が、ワークツリーに反映される
②内容の組み合わせは自動で行われる、コンフリクトが発生して自動で出来ない場合は、手動でワークツリーを編集する
③組み合わせが完了したワークツリーの内容を、ステージングエリアに追加して、ローカルリポジトリにコミットする

Q45
「oldbranch」という名前の既存のブランチを、
「newbranch」という名前に変更する

A45
```
git branch -m oldbranch newbranch
```

「-m」は「move」の略

Linuxにおいて、「move」の略である「mvコマンド」は、
名前の変更を行うためのコマンド

Q46
「bug-fix」というブランチを新規作成した上で、
masterブランチからこのブランチに切り替える

A46
```
git checkout -b bug-fix
```

「-b」オプションをつけると、
・ブランチの新規作成（git branch）
・作成したブランチへHEADを移動（git checkout）
を１つのコマンドで実行できる

Q47
HEADとはなにか？

A47
HEADとは、現在参照しているコミットのこと
HEADの実体は、現在のコミットへのポインタ

HEADが移動することで、参照するコミットが変更（チェックアウト）される

HEADが指し示すコミットは新たに追加されていくコミットの親になる

HEADがブランチのポインタを指している場合にコミットをすることで
ブランチを伸ばしていくことができる

HEADがどのブランチにもない状態を、
detached HEAD（切り離されたHEAD）という

Q48
チェックアウトとはなにか？

A48
チェックアウトとは、HEADを移動したり、
ワークツリー・ステージングエリアのファイルを置き換えること

例えば、「機能追加がひと段落したから、バグ修正に移ろう」というような場合、
新機能追加のブランチから、バグ修正のブランチにチェックアウトする

HEADをbug-fixブランチに移動する場合、以下のコマンドを実行する
```
git checkout bug-fix
```

HEADをSHA-1ハッシュ値が「03ebe2a」のコミットに
チェックアウトするには、以下のコマンドを実行する
```
git checkout 03ebe2a
```

HEADを移動すると、ワークツリーとステージングエリアの内容が
HEADが指すコミットの内容に置き換わる

Q49
SHA-1 ハッシュが「03ebe2a」のコミットに、
HEADを移動する

A49
```
git checkout 03ebe2a
```

「git checkout コミット」というようにコミットを指定すると
指定したコミットにHEADが移動する

このコマンドを実行すると、
ワークツリーにはHEADの移動先のコミットの内容がコピーされる
これを利用すれば以前のコミットの内容を確認できる

Q50
SHA-1 ハッシュが「03ebe2a」のコミットに含まれる
「file」という名前のファイルの内容を、
ワークツリーとステージングエリアの「file」にコピーする

A50
```
git checkout 03ebe2a -- file
```

「git checkout コミット -- ファイル名」というようにすると、
指定したコミットの指定したファイルの内容を、
ワークツリーとステージングエリアにコピーすることができる

この場合はHEADは移動しないことに注意

Q51
detached HEADとはなにか？

A51
HEADがどのブランチにもない状態のこと

「git checkout <コミット>」というようにして、
ブランチがあるコミット以外にHEADを移動するとdetached HEADになる

detached HEADの状態でコミットすると、子にコミットが追加されるが、
そのコミットはどのブランチにも紐づいていないコミットになってしまうので、
基本的にアクセスできないし、いずれ削除されてしまう

Q52
SHA-1ハッシュ値が「03ebe2a」のコミットに、
HEADとmasterブランチを移動する
ただし、現在HEADはmasterブランチを指しているものとし、
指定したコミットの内容を、
ワークツリーとステージングエリアにコピーするものとする

A52
```
git reset --hard 03ebe2a
```

「git reset」はHEADの位置を移動する機能
HEADがブランチを指している場合には、ブランチごと移動するのが特徴

「git reset --soft コミット」とするとワークツリーとステージングエリアはそのまま
「git reset コミット」とするとステージングエリアにHEADの内容をコピーする
「git reset --hard コミット」とするとワークツリーとステージングエリアにHEADの内容をコピーする

「git reset」はHEADを移動する機能のほかにも、
ワークツリーやステージングエリアにHEADの内容をコピーする機能を持つ

Q53
HEADのコミットでの変更を取り消すコミットを追加する

A53
git revert HEAD

指定したコミットが行った変更を打ち消すコミットを追加するコマンド
「revert」は「元に戻す」の意味

「git revert 打ち消すコミット」と実行すれば、
指定したコミットを打ち消すコミットが追加される

リモートリポジトリにあるコミットをresetで削除してしまうと、
そのリモートリポジトリを上流に設定している別の開発者にも影響が出てしまい
コンフリクトが発生してしまう可能性が非常に高い

revertならコミット削除せずに新しいコミットを追加するだけなので、
他の開発者への影響を最小限にして変更を打ち消すことができる

また、revertならコミットを取り消したことが履歴としても残るので、

コミットの取り消しを行ったことを履歴として残したい場合にも役立つ

Q54
fast-forward merge（早送りマージ）とはなにか？

A54
新たなコミットを作らずに、
取り込むブランチ（HEADが指すブランチ）を、
取り込まれるブランチの位置に移動するだけのマージ

例えば、HEADがmasterブランチを指している場合に
「git merge bug-fix」コマンドを実行して、
fast-forward merge（早送りマージ）が行われた場合、
masterブランチをbug-fixブランチが指すコミットに移動する

通常の3-way merge（3方向マージ）とは異なり、
新たなコミットを追加しないのが特徴

fast-forward merge（早送りマージ）は、
取り込むブランチが指すコミットが、
取り込まれるブランチの祖先になっている場合のみ可能

この条件が満たされている状態で「git merge」コマンドを実行すると、
自動的にfast-forward merge（早送りマージ）になる（※設定にもよります）

fast-forward merge（早送りマージ）できるがしたくない場合は、
「git merge --no-ff」というコマンドを実行する
「--no-ff」は「no fast-forward」の略

Q55
現在HEADのあるブランチにbug-fixブランチを3方向マージする
ただし、fast-foward mergeをしようと思えばできるものとする

A55
```
git merge --no-ff bug-fix
```

「--no-ff」は「no fast-forward（早送りマージしない）」の意味

fast-forward merge（早送りマージ）は、取り込むブランチの先端が、
取り込まれるブランチの祖先になっている場合のみ可能

fast-forward merge（早送りマージ）ができる場合には、
「git merge」コマンドはデフォルトで早送りマージをするようになっている

Q56
「git log」コマンドのヘルプを確認する

A56
```
git help log
```

具体的なコマンドのヘルプを見たいときには、
「git help コマンド」というように引数に具体的なコマンドを入れる

Git全般のヘルプを見たい時には、

「git help」というように引数を指定せずに実行する

Q57
「example.txt」に変更を行なっている
コミットの履歴を表示する

A57
```
git log example.txt
```

「git log ファイル」というように引数にファイルを指定すると、
指定したファイルに変更を行っているコミットの履歴だけを表示することができる

「git log example.txt」というコマンドを実行した場合、
example.txtに変更を加えていないコミットは表示されない

その他オプションを指定する時には、
「git log --graph --oneline ファイル」というように、
オプションの後にファイルを指定する

Q58
コミット履歴を視覚的にわかりやすく表示する

A58
```
git log --graph
```

「--graph」オプションをつけると、
ブランチの分岐やマージがわかりやすいように
視覚的にコミットの履歴を表示することができる

Q59
コミット履歴を一行でシンプルに表示する

A59
```
git log --oneline
```

「git log --oneline」というように「--oneline」オプションを付けて実行すると、
コミット履歴が以下のように一行で表示されるようになる

```
83hefo8 "Commit Message"
```

Q60
現在HEADがあるブランチからたどれるコミットだけでなく、
その他すべてのブランチからたどれるコミットを表示する

A60
`git log --all`

「`git log`」というコマンドを実行すると、
現在`HEAD`が指すブランチからたどれるコミットしか表示されない

「`git log --all`」というように「`--all`」オプションをつけて実行すると、
すべてのブランチからたどれるコミットが表示されるようになる

Q61
`SHA-1` ハッシュの値が「`ca82a6d`」のコミットに
「`newbranch`」という名前のブランチを作成する

A61
`git branch newbranch ca82a6d`

新規ブランチを`HEAD`が指すコミット以外のコミットに作成したい場合、
「`git branch` ブランチ名 コミット」というように実行すると、
指定したコミットに任意の名前のブランチを新規作成できる

Q62
`HEAD`が指すコミットと、
`HEAD`が指すコミットの親のコミットの差を確認する

A62
以下のどちらか
`git diff HEAD^ HEAD`
`git diff HEAD~1 HEAD`

`HEAD`の１つ前は「`HEAD^`」「`HEAD^1`」「`HEAD~1`」と表現する
`HEAD`の２つ前は「`HEAD^^`」「`HEAD^1^1`」「`HEAD~2`」と表現する

キャレット（^）は、何番目の親かを指定する記号
１番目の親なら「`^1`」、２番目の親なら「`^2`」
「`^1`」は省略して「`^`」と書いてもいい

チルダ（~）は、何世代前かを指定する記号
１世代前なら「`~1`」、２世代前なら「`~2`」

「`HEAD^^`」は「`HEAD`の１番目の親の１番目の親」の意味なので、
`HEAD`の２つ前のコミットを指す

Q63

タグとはなにか？

A63
コミットをわかりやすくする名前をつける機能のこと

コミットは通常、以下のようなSHA-1 ハッシュで識別される
「ca82a6dff817ec66f44342007202690a93763949」

SHA-1 ハッシュを覚えるのは大変なので、
タグをつかってコミットに名前をつけて、
その名前でコミットを指定できるようにすると便利

主なタグの種類として、軽量タグと注釈付きタグがある
軽量タグは、名前だけをつけることができる
注釈付きタグには、名前のほかにメッセージをつけることができる

Q64
HEADが指しているコミットに
「tagname」という名前の軽量タグをつける

A64
```
git tag tagname
```

SHA-1 ハッシュを覚えるのは大変なので、
タグをつかってコミットに名前をつけて、
その名前でコミットを指定できるようにすると便利

コミットを指定しないとHEADが指すコミットにタグを設定するが、
「git tag tagname ca82a6d」というようにコミットを指定して実行すると、
指定したコミットにタグをつけることができる

Q65
設定されているタグの一覧を表示する

A65
```
git tag
```

注釈付きタグで設定したメッセージを含めて表示したい場合には、
「-n」オプションをつける

「-n」オプションは「number」の略
「-n2」とするとメッセージを2行だけ表示する、というように
表示するメッセージの行数を指定することができる

Q66
SHA-1 ハッシュの値が「ca82a6d」のコミットに
「tagname」という名前の軽量タグをつける

A66
```
git tag tagname ca82a6d
```

「git tag タグの名前 コミット」というように実行すれば、
指定したコミットに指定したタグ名をつけることができる

Q67
HEADが指しているコミットに注釈付きタグをつける
ただし以下の値を設定するものとする
名前：v1.0.2
メッセージ：脆弱性の修正

A67
```
git tag -a v1.0.2 -m ”脆弱性の修正”
```

「-a」オプションをつけると注釈付きタグをつけることができる
「-a」オプションは「annotate（注釈をつける）」の略

注釈付きタグにつけるメッセージは「-m」オプションで指定できる

HEADが指すコミット以外に注釈付きタグをつけたい場合は、
「git tag -a タグの名前 -m ”メッセージ” コミット」というように、
最後にコミットを指定する

Q68
注釈付きタグのメッセージが見えるように
設定されているタグの一覧を表示する
ただし、メッセージは2行だけ表示するものとする

A68
```
git tag -n2
```

注釈付きタグで設定したメッセージを含めて表示したい場合には、
「-n」オプションをつける

「-n」オプションは「number」の略
「-n2」とするとメッセージを2行だけ表示する、というように
表示するメッセージの行数を指定することができる

「git tag -n」というように「-n」オプションの後に数字をつけないと、

メッセージは1行だけ表示される

Q69
「tagname」という名前のタグを削除する

A69
```
git tag -d tagname
```

引数に指定した名前のタグを削除することができる

「-d」は「delete（削除する）」の略

Q70
バージョン番号とはなにか？

A70
ファイルなどを変更していく際に、変更の段階を表すための番号のこと
「Python 3.9.5」「macOS Big Sur 11.3.1」みたいな
バージョンを付与することで、ファイルの変更の段階に応じてファイルを管理できるようになる

変更を重ねていくにつれて、以下のようにバージョンは上がっていく
3.9.5→3.9.6→3.10.0→3.10.1→4.0.0

Q71
バージョン番号の一般的な設定ルールは？

A71
「X.Y.Z」というように、X,Y,Zという３つの数値で表現する方法が一般的
各数値はそれぞれ以下の意味を持つ
X：メジャーバージョン
見た目や操作性が変わる大きな変更や、下位互換性のない変更などでインクリメント
Y：マイナーバージョン
新機能の追加や既存機能の削除・修正などでインクリメント
Z：パッチバージョン
バグや脆弱性の修正などの軽微な変更でインクリメント

具体的な一例としては以下のようにバージョンを設定する
1.0.0 初めてのリリース
1.0.1 バグ修正
1.0.2 誤字の修正
1.1.0 新機能の追加
2.0.0 UI全体の刷新と新機能の追加

Q72
スタッシュ（stash）とはなにか？

A72
変更の一時的な退避場所のこと

HEADを現在作業中のブランチから、他のブランチやコミットに移動する際には、
ワークツリーの変更をコミットするか、スタッシュに退避させる必要がある

まだコミットはしたくないが、現在の作業を捨てたくない場合に使える

ワークツリーの変更をスタッシュに入れるには以下のコマンド
「git stash」

Q73
ワークツリーとステージングエリアの変更をスタッシュに入れる

A73
git stash

スタッシュ（stash）とは変更の一時的な退避場所のこと

HEADを現在作業中のブランチから、他のブランチやコミットに移動する際には、
ワークツリーの変更をコミットするか、スタッシュに退避させる必要がある

まだコミットはしたくないが、現在の作業を捨てたくない場合に使える

ワークツリーの変更をスタッシュに入っている変更を
ワークツリーに反映させるには以下のコマンド
「git stash pop」

「pop」からわかるように、
stashは後入れ先出し（スタック）の構造になっている

Q74
スタッシュに退避させた変更のリストを表示する

A74
git stash list

stashに入っている変更の一覧を表示することができる

stashは先入れ後出し（スタック）のデータ構造を持つので、

「git stash pop」としてスタッシュに入れた変更を取り出すと、
最後にスタッシュに入れた変更が取り出される

「git stash list」コマンドの出力では、
最後に追加したものが一番上に出力される

Q75
最後にスタッシュに入れた変更を
ワークツリーに反映させた上で、スタッシュから削除する
ただし、ステージングエリアの内容は反映させないものとする

A75
```
git stash pop
```

stashは先入れ後出し（スタック）のデータ構造を持つので、
「git stash pop」としてスタッシュに入れた変更を取り出すと、
最後にスタッシュに入れた変更が取り出され、ワークツリーに反映される
ステージングエリアの内容は破棄されることに注意

「git stash list」コマンドの出力では、
一番上に出力される変更が取り出される

ステージングエリアの内容もstashに入れたときの状態に戻すには
「git stash pop --index」というように「--index」オプションを指定する

Q76
最後にスタッシュに入れた変更を
ワークツリーとステージングエリアに反映させた上で、スタッシュから削除する

A76
```
git stash pop --index
```

stashは先入れ後出し（スタック）のデータ構造を持つので、
「git stash pop --index」としてスタッシュに入れた変更を取り出すと、
最後にスタッシュに入れた変更が取り出され、
ワークツリーとステージングエリアのに反映される

「git stash list」コマンドの出力では、
一番上に出力される変更が取り出される

ステージングエリアの内容はstashに入れたときの状態に戻さない場合には
「git stash pop」というように「--index」オプションを指定しない

Q77

スタッシュに入っている変更をすべて削除する

A77
```
git stash clear
```

「git stash clear」を実行することで、
スタッシュに入っている変更をすべて削除することができる

Q78
Gitの操作履歴を表示する

A78
```
git reflog
```

「git reflog」を実行すると、Gitの操作履歴を表示することができる
「reflog」は「Reference logs（ログの参照）」の意味

このコマンドの出力は以下のようになる
```
5dd8520 (HEAD -> main) HEAD@{0}: reset: moving to HEAD
5dd8520 (HEAD -> main) HEAD@{1}: checkout: moving from bug-fix
to main
```

「HEAD@{0}」は一番最近行った操作、
「HEAD@{1}」は「HEAD@{0}」の1つ前に行った操作を指す

「git reset --hard HEAD@{1}」というように、
「git reflog」コマンドで表示された作業履歴を指定すれば、
「HEAD@{1}」の作業を終えた状態に戻ることができる
つまり、一番最近に行った操作を取り消すことができる

Q79
一番最近に行った操作を取り消す

A79
```
git reset --hard HEAD@{1}
```

「git reflog」を実行すると、Gitの操作履歴を表示することができる「reflog」は「Reference logs（ログの参照）」の意味

このコマンドの出力は以下のようになる
```
5dd8520 (HEAD -> main) HEAD@{0}: reset: moving to HEAD
5dd8520 (HEAD -> main) HEAD@{1}: checkout: moving from bug-fix
to main
```

「HEAD@{0}」は一番最近行った操作、
「HEAD@{1}」は「HEAD@{0}」の1つ前に行った操作を指す

「git reset --hard HEAD@{1}」というように、
「git reflog」コマンドで表示された作業履歴を指定すれば、
「HEAD@{1}」の作業を終えた状態に戻ることができる
つまり、一番最近に行った操作を取り消すことができる

Q80
「file」という名前のファイルにバグのある行が見つかった
このバグのある行を追加した犯人を調べる

A80
```
git blame file
```

「git blame ファイル名」というように実行すると、
指定したファイルの各行が最後に変更されたコミットと、
誰がいつ変更したのかを確認することができる

「blame」は「〜のせいにする、とがめる」の意味
犯人捜しのためのコマンド

「git blame ファイル名」を実行すると、以下のような出力が得られる
```
c82206wc (username 2021-08-06 14:01:47 +0900 1) Hello
2f25ed2q (username 2021-08-06 12:15:43 +0900 2) World
```

指定したファイルの各行ごとに以下のような情報が表示される
・行が最後に変更されたコミット
・変更したユーザー
・変更をコミットした日時
・行の具体的な記述内容

Q81
リモートリポジトリとローカルリポジトリはそれぞれなにか？

A81
ファイルやディレクトリの状態を記録する領域であるリポジトリには、以下の２種類がある
・ローカルリポジトリ
ユーザーひとりひとりが利用するリポジトリ、手元のマシンそれぞれに配置
ユーザーごとに用意する
・リモートリポジトリ
複数人で共有するリポジトリ、専用のサーバーに配置
プロジェクトごとに用意する

メンバーそれぞれがローカルリポジトリを持つことは、以下のようなメリットがある
・他のメンバーへの影響を小さく出来る
・インターネットがなくても変更をコミットできる

Q82
バージョン管理システムにおいて、
「集中型」と「分散型」とはそれぞれなにか？

A82
・集中型
リポジトリは専用のサーバーに配置されたリモートリポジトリ１つだけで、
メンバー全員がその１つのリポジトリを共有する形式を集中型という

集中型では、メンバーごとにはリポジトリを持たないので、
シンプルでわかりやすい反面、以下のようなデメリットがある
・自分のコミットが他のメンバーに強く影響してしまう
・インターネットがないとコミットできない
・アクセス権の設定が必要なため、不特定多数による開発に向かない

Subversionは集中型のバージョン管理システム

・分散型
専用のサーバーに配置されたリモートリポジトリだけでなく、
メンバーそれぞれが手元のマシンにローカルリポジトリを持つ形式を分散型という

集中型よりも少し仕組みが複雑になるのがデメリットだが、
分散型は集中型と比較した時に、以下のようなメリットがある
・他のメンバーへの影響が少ないので、ローカルリポジトリでいろいろ試せる
・インターネットがなくてもローカルリポジトリにコミットできる
・アクセス権の設定が不要なので、ソーシャルコーディングがしやすい

Gitは分散型のバージョン管理システム

Q83
GitHubとはなにか？

A83
Gitリポジトリをインターネット上に配置できる、ホスティングサービスのこと

GitHub登場以前は、リモートリポジトリを配置するサーバーを、各自が用意する必要があった

現在は、リモートリポジトリはGitHub上に配置することが一般的

GitHubは、単にリモートリポジトリのホスティングサービスとしての機能だけではなく、
Pull Requestなどの機能によって、複数の開発者がコラボレーションできるようにしており、
ソーシャルコーディングを広めたきっかけになったと言われている

GitHubの類似サービスとしては、BitbucketやGitLabなどがある

Q84
ソーシャルコーディングとはなにか？

A84
ソースコードを全世界に共有して、他のユーザーとコラボレーションしながら開発をしていくこと

GitHubの登場によって広まった概念で、
現在は大量のソフトウェアがソーシャルコーディングで開発されている

GitHubが持つPullRequestという機能によって、
不特定多数の開発者がコラボレーションした開発がしやすくなった

Ruby on Rails / Bootstrap / Node.js / 各種ライブラリなど、具体例を挙げればきりがない

Q85
URLが「https://github.com/username/example.git」という
リモートリポジトリを手元のマシンにコピーする

A85
```
git clone https://github.com/username/example.git
```

「clone」は「クローン人間」などにも使われるように、
まったく同じものを複製するという意味

「git clone リモートリポジトリのURL」とすることで、
指定したリモートリポジトリをローカルリポジトリとしてcloneできる

「git clone」コマンドでリモートリポジトリをコピーした場合、
コピー元のリモートリポジトリを追跡するように自動で設定される

Q86
pushとはなにか？

A86
ローカルリポジトリのブランチのコミット履歴を、
リモートリポジトリにも反映すること

「git push リモートリポジトリ名 ローカルのブランチ名:リモートのブランチ名」
というように実行することで、指定したローカルリポジトリのブランチを、
指定したリモートリポジトリのブランチ名で、
指定したリモートリポジトリにpushすることができる

Q87
現在作業中のローカルリポジトリに設定されている
リモートリポジトリを表示する

A87
git remote

push/fetch/pullの操作は、
このコマンドで表示されたリモートリポジトリに対して行うことができる

リモートリポジトリのデフォルトの名前は「origin」

リモートリポジトリは複数設定することもできる

Q88
ローカルリポジトリの「local-name」ブランチの内容を
「origin」という名前のリモートリポジトリにpushして、
リモートリポジトリに「remote-name」ブランチを作成する

A88
git push origin local-name:remote-name

「git push リモートリポジトリ名 ローカルのブランチ名:リモートのブランチ名」
というように実行することで、指定したローカルリポジトリのブランチを、
指定したリモートリポジトリのブランチ名で、
指定したリモートリポジトリにpushすることができる

なので、「git push origin local-name:remote-name」とすると、
ローカルリポジトリにある「local-name」ブランチを
「remote-name」という名前のブランチとして、
「origin」という名前のリモートリポジトリにpushすることができる

「git push origin add-feature」のように「:（コロン）」を使わずに書いた場合は、
「:（コロン）」の後ろのリモートのブランチ名が省略されたことになり、
その場合はローカルと同じ名前のブランチがリモートにも作成される

Q89
リモート追跡ブランチとはなにか？

A89
ローカルリポジトリにある、リモートリポジトリのブランチの位置を記憶するブランチのこと

リモート追跡ブランチにHEADを置いてコミットしても、
リモート追跡ブランチを伸ばすことはできないことに注意

あくまでもリモートリポジトリのブランチの位置を知るためのブランチ

リモート追跡ブランチは「remotes/origin/main」というように、
「remotes/リモートリポジトリ名/ブランチ名」の形で表示される

また、リモート追跡ブランチは、
「remotes」を省略して「origin/main」というように、
「リモートリポジトリ名/ブランチ名」の形で表示されることもある

Q90
リモート追跡ブランチも含めてブランチの一覧を表示する

A90
```
git branch -a
```

「-a」は「all」の略で、
（リモート追跡ブランチも含めた）すべてのブランチを表示するという意味

リモート追跡ブランチは「remotes/origin/main」というように、
「remotes/リモートリポジトリ名/ブランチ名」の形で表示される

また、リモート追跡ブランチは、
「remotes」を省略して「origin/main」というように、
「リモートリポジトリ名/ブランチ名」の形で表示されることもある

リモート追跡ブランチだけを表示するには、
「git branch -r」というように「-r」オプションをつける
「-r」は「remotes」の略

Q91
Pull Request（プルリクエスト）とはなにか？

A91
トピックブランチの最新のコミットを、
メインブランチにマージするように、リポジトリの管理者に依頼する機能
プルリクエスト、プルリク、マージリクエストとも呼ばれる

OSSの開発などにおいては、
誰でも勝手にメインブランチにマージできるようにしてしまうと、
バグのあるコードをメインブランチに含めてしまうリスクが非常に高い

なので、勝手にマージできるようにはせず、
トピックブランチの最新コミットを、リポジトリの管理者にチェックしてもらった上で、
リポジトリの管理者の権限でマージするかを決める仕組みを導入している

それぞれの開発者がトピックブランチで行ったプロジェクトのコードの変更を、
メインブランチにマージするようにリポジトリの管理者に依頼する機能がPull Request

Pull Requestによって、承認された変更のみが反映されるようになるので、
技術力もさまざまな見ず知らずの人でも、共同で開発できるようになる

また、OSS以外の企業のソフトウェア開発でも使われる
エンジニアのレベルにばらつきがある企業や、
ひとつのバグが非常に大きな損害になる会社では、
経験の少ない開発者が致命的な変更を加えないように
責任者にPull Requestを送った上で、責任者の判断でマージするようにすれば
安全にソフトウェアを開発することができるようになる

Q92
Pull Requestに対してレビューコメントをしたあとに、
レビューコメントを送る際に出てくる3つの選択肢
「Comment」「Approve」「Request Change」とは
それぞれどのように使い分けるか？

A92
・Comment（コメント）
変更を明確には承認せず、さらなる変更をリクエストすることもなく、
おおまかなフィードバックだけを残したい場合は、[Comment] を選択する
・Approve（承認する）
フィードバックを提出して、Pull Request で提案された変更を
マージすることを承認するには、[Approve] を選択する

・Request Changes（変更を要求する）
Pull Request をマージする前に対処すべき問題を
フィードバックするには、[Request changes] を選択する

ただし、Pull Requestの作成者はCommentしかできない

Q93
ローカルリポジトリの「add-feature」ブランチの内容を
「origin」という名前のリモートリポジトリにpushして、
リモートリポジトリにも「add-feature」ブランチを作成する
ただし、リモートリポジトリの「add-feature」ブランチを
ローカルリポジトリの「add-feature」ブランチの
上流ブランチとして設定するものとする

A93
```
git push -u origin add-feature
```

「git push リモートリポジトリ名 ローカルのブランチ名:リモートのブランチ名」
というように実行することで、指定したローカルリポジトリのブランチを、
指定したリモートリポジトリのブランチ名で、
指定したリモートリポジトリにpushすることができる

なので、「git push origin add-feature:add-feature」とすると、
ローカルリポジトリにある「add-feature」ブランチを
「add-feature」という名前のブランチとして、
「origin」という名前のリモートリポジトリにpushすることができる

「git push origin add-feature」のように「:（コロン）」を使わずに書いた場合は、
「:（コロン）」の後ろのリモートのブランチ名が省略されたことになり、
その場合はローカルと同じ名前のブランチがリモートにも作成される

「-u」は「set upstream（上流を設定する）」の略

このオプションを含めることで上流ブランチを設定することができるので、
以降は単に「git push」と実行すれば良くなる

Q94
上流ブランチとはなにか？

A94
ローカルリポジトリのブランチに対応する、
リモートリポジトリのブランチのこと

上流ブランチを設定することで、「git push」や「git pull」などのコマンドで
引数にリモートリポジトリを指定しないでもよくなる

例えば、ローカルリポジトリの「add-feature」ブランチの上流ブランチに
リモートリポジトリの「add-feature」ブランチを設定すると、
「git push origin add-feature」というように
引数にリモートリポジトリを指定することなく、
「git push add-feature」というコマンドで十分になる

「git push」というようにブランチの引数も省略すると、
上流ブランチが設定されているすべてのブランチをpushする

Q95
fetchとは？

A95
リモートリポジトリのブランチのコミット履歴を、
ローカルリポジトリに反映すること

fetchしただけでは、ローカルリポジトリのブランチは移動しない

pullとは異なり、マージをせずに変更をローカルリポジトリに取り込むだけなので、

pullよりも安全にリモートリポジトリの変更を取り入れることができる

Q96
「origin」という名前のリモートリポジトリにある
「add-feature」ブランチをfetchする

A96
git fetch origin add-feature

「git fetch リモートリポジトリ名 リモートのブランチ名」
というように実行することで、指定したリモートリポジトリのブランチを、
現在作業中のローカルリポジトリにfetchすることができる

「git fetch origin」というように、
リモートのブランチ名の引数を省略すると、
originのすべてのブランチをfetchする

「git fetch」というように、引数をすべて省略すると、
ローカルのHEADが指すブランチの上流ブランチからfetchする

Q97

「origin」という名前のリモートリポジトリの
すべてのブランチのコミット履歴をfetchする

A97
```
git fetch origin
```

「git fetch リモートリポジトリ名 リモートのブランチ名」
というように実行することで、指定したリモートリポジトリのブランチを、
現在作業中のローカルリポジトリにfetchすることができる

「git fetch origin」というように、
リモートのブランチ名の引数を省略すると、
originのすべてのブランチをfetchする

「git fetch」というように、引数をすべて省略すると、
ローカルのHEADが指すブランチの上流ブランチからfetchする

Q98
ローカルリポジトリの既存のブランチの一覧を表示する
ただし、設定されている上流ブランチも含めて表示するものとする

A98
```
git branch -vv
```

「-v」は「verbose（詳細な）」の略

「git branch -v」はブランチの一覧を詳細に表示
「git branch -vv」はブランチの一覧をより詳細に表示

「git branch -v」を実行するとブランチ名だけでなく、
ブランチが指すコミットのSHA-1ハッシュ値とコミットメッセージを表示する

「git branch -vv」を実行すると「git branch -v」の出力に加えて
上流ブランチも含めて表示してくれる

Q99
pullとはなにか？

A99
fetch + merge のこと

コマンドでいうと、「git pull origin main」を実行すると、
「git fetch origin main」からの「git merge FETCH_HEAD」が実行される

リモートリポジトリのブランチを、
ローカルリポジトリに取り込む際に生じる一連の流れを、
ひとつの操作で行えるようにしている

Q100
「origin」という名前のリモートリポジトリの「main」ブランチの内容を、
ローカルリポジトリの「main」ブランチにpullする

A100
git pull origin main

「git pull リモートリポジトリ名 リモートのブランチ名」
というように実行することで、指定したリモートリポジトリのブランチを
現在作業中のローカルリポジトリのブランチにpullすることができる

例えば「git pull origin main」を実行すると、
「git fetch origin main」からの「git merge FETCH_HEAD」が実行される

FETCH__HEADとは、fetchしてきたブランチと同じコミットを指す、
ローカルリポジトリにあるポインタのことなので、
この場合は「origin/main」と同じコミットを指していることになる

Q101
FETCH_HEADとはなにか？

A101
「git fetch リモートリポジトリ名 リモートのブランチ名」というコマンドを実行した際に
引数に指定したブランチと同じコミットを指す、ローカルリポジトリにあるポインタのこと

「git pull origin main」を実行すると、
「git fetch origin main」からの「git merge FETCH_HEAD」が実行される
この場合、FETCH_HEADというポインタは、
「origin/main」ブランチと同じコミットを指している

Q102
「origin」という名前のリモートリポジトリにある
「add-feature」ブランチを削除する

A102

```
git push --delete origin add-feature
```
もしくは
```
git push origin :add-feature
```

マージが完了して不要になったリモートリポジトリのブランチは削除する
削除したブランチのリモート追跡ブランチもローカルリポジトリから削除される

自分以外の開発者がリモートリポジトリのブランチを削除した場合は、
自分のマシンのローカルリポジトリのリモート追跡ブランチは削除されないことに注意

Q103
「origin」という名前のリモートリポジトリでは
すでに削除されているブランチのリモート追跡ブランチを
ローカルリポジトリからも削除する

A103
```
git fetch --prune
```
もしくは
```
git remote prune origin
```

「prune」は「刈り取る」の意味
不要になったリモート追跡ブランチを刈り取る、という意味

リモートリポジトリからブランチが削除されたブランチの
リモート追跡ブランチがローカルリポジトリに残ってしまうことがあるので
その場合はこのコマンドを使って削除を行うと便利

Q104
ローカルリポジトリの「add-feature」ブランチの内容を
「origin」という名前のリモートリポジトリにpushして、
リモートリポジトリにも「add-feature」ブランチを作成する
ただし、ローカルリポジトリの「add-feature」ブランチを
現在HEADが指しているものとする

A104
```
git push origin HEAD
```

「git push リモートリポジトリ名 ローカルのブランチ名:リモートのブランチ名」
というように実行することで、指定したローカルリポジトリのブランチを、
指定したリモートリポジトリのブランチ名で、
指定したリモートリポジトリにpushすることができる

ブランチの代わりにHEADを引数に指定すると、
HEADが指すブランチを指定したことになる

指定したブランチと同名のブランチがリモートリポジトリにない場合は
自動的に作成される
リモートとローカルのブランチ名は一致させるのが一般的

「-u」オプションを含めれば、上流ブランチを設定することができる

Q105
ローカルリポジトリの「add-feature」ブランチの内容を
「origin」という名前のリモートリポジトリにpushする

ただし、ローカルリポジトリの「add-feature」ブランチは
「origin」という名前のリモートリポジトリに
上流ブランチが設定されており、
ローカルリポジトリのHEADは、
ローカルリポジトリの「add-feature」ブランチを指しているとする

A105
git push

「git push リモートリポジトリ名 ローカルのブランチ名:リモートのブランチ名」
というように実行することで、指定したローカルリポジトリのブランチを、
指定したリモートリポジトリのブランチ名で、
指定したリモートリポジトリにpushすることができる

「git push add-feature」というように、リモートリポジトリ名を省略すると、
設定されている上流ブランチのあるリポジトリを指定したことになる

「git push」というように、ローカルとリモートのブランチ名を省略すると、
ローカルのHEADが指すブランチを指定したことになる

Q106
ローカルリポジトリのHEADが指しているブランチの
上流ブランチからfetchする

A106
git fetch

「git fetch リモートリポジトリ名 リモートのブランチ名」

というように実行することで、指定したリモートリポジトリのブランチを、

現在作業中のローカルリポジトリにfetchすることができる

「git fetch origin」というように、

リモートのブランチ名の引数を省略すると、

originのすべてのブランチをfetchする

「git fetch」というように、引数をすべて省略すると、

ローカルのHEADが指すブランチの上流ブランチからfetchする

Q107
ローカルリポジトリのHEADが指しているブランチの

上流ブランチからpullする

A107
git pull

「git pull リモートリポジトリ名 リモートのブランチ名」
というように実行することで、指定したリモートリポジトリのブランチを
現在作業中のローカルリポジトリのブランチにpullすることができる

例えば「git pull origin main」を実行すると、
「git fetch origin main」からの「git merge FETCH_HEAD」が実行される

FETCH__HEADとは、fetchしてきたブランチと同じコミットを指す、
ローカルリポジトリにあるポインタのことなので、
この場合は「origin/main」と同じコミットを指していることになる

「git pull」というように引数を省略すると、
「git fetch」の引数が省略されたことになる

Q108
リモートリポジトリ「origin」の「add-feature」ブランチを、
ローカルリポジトリの「add-feature」ブランチの
上流ブランチに設定する

ただし、ローカルリポジトリのHEADは
「add-feature」ブランチを指しているものとする

A108
git branch -u origin/add-feature

「-u」は「set upstream to」の略

ローカルのHEADが「add-feature」以外を指している場合は、
「git branch add-feature -u origin/add-feature」とする

Q109
OSSとはなにか？

A109
Open Source Softwareの略称

ソースコードが広く一般に公開されており、
誰もが自由に利用・修正・再配布することのできるソフトウェアのこと

GitHub上では多くのOSSの開発が行われている
GitもOSSのひとつ

Q110
GitHubにおいて、以下の用語の意味は？
・コラボレータ（Collaborator）
・コントリビュータ（Contributor）

A110
・コラボレータ（Collaborator）
プロジェクトのコアな開発者のこと
プロジェクトのリポジトリでpushやmergeなど
リポジトリの変更を行う権限を持っている

・コントリビュータ（Contributor）
プロジェクトの外部から貢献している開発者のこと
プロジェクトのリポジトリの変更権限は持っておらず、
ForkやPull Requestを使ってリポジトリを変更する

Q111
forkとはなにか？

A111
他人のリモートリポジトリを、自分のリモートリポジトリにコピーすること

forkして開発を行うことで、
OSSなど他者のリポジトリに機能を追加する・バグ修正するなどの貢献ができるようになる

例えば、OSSプロジェクトへの貢献にする場合以下のような手順で開発が行われれる
①OSSプロジェクトのリポジトリをforkして自分のリモートリポジトリにコピー
②forkしたリモートリポジトリをcloneしてローカルリポジトリにコピー
③ローカルリポジトリ内でトピックブランチを切って開発する
④開発完了したらトピックブランチをマージしてリモートリポジトリにpush
⑤自分のリモートリポジトリからOSSのリポジトリにPull Request
⑥OSSの管理者がPull Requestを承認したらマージ完了

forkはコピー元リポジトリへの貢献が前提にあり、
forkしたことはコピー元リポジトリに通知されるようになっている

Q112
rebaseとはなにか？
また、どのような用途で使われるか？

A112
ブランチの根元を移動するコマンド

rebaseは以下二つの用途で使われる
・コミット履歴をわかりやすくしたい
複数の開発者が複数のブランチを切って開発していると、
マージが入り乱れてコミット履歴がわかりづらくなる場合がある
そんなときにrebaseを用いてコミット履歴をわかりやすくすることもある

・別のブランチの内容を取り込みたい
トピックブランチで開発を行っている間に、
他の開発者によってメインブランチが伸びたとする
その場合、メインブランチの内容をトピックブランチに取り入れるのに使える

この場合においては実はrebaseを使わなくても、
トピックブランチにメインブランチの内容を取り込む形で
マージすればメインブランチを伸ばさずにメインブランチの内容を取り込める

Q113
rebaseはどのような仕組みで行われるか？

A113
cherry-pickを繰り返すことで行われる

なので、rebaseで根元が移動されたブランチのコミットは、
元のブランチにあったコミットとは別の新しいコミットになる

別のコミットになるので、すでにpushしているブランチを
rebaseしてしまうとそれ以降pushできなくなってしまう

また、rebaseを行うとcherry-pickが行われた数だけ
コンフリクトが発生する可能性がある

Q114
cherry-pickとはなにか？

A114
指定したコミットの内容を取り込む機能のこと

「cherry-picking」とは本来詭弁のテクニックで、
自分に有利な証拠だけを並べ立てることを指す

これに由来するgitの「cherry-pick」は、
必要なコミットだけを「いいとこ取り」する機能

「git cherry-pick コミット」を実行すれば、
指定したコミットのに内容を現在のブランチに追加することができる

自動解決できない場合は、コンフリクトを手動で直す必要がある

Q115
新たなコミットは追加せずに、ワークツリーに
「main」ブランチが指すコミットの内容を取り込む

A115
```
git cherry-pick -n main
```

「-n」は「no commit」の意味

「git cherry-pick main」と実行すると、
mainブランチが指すコミットの内容を取り込んだ
新しいコミットが自動的に生成されてしまう

新しいコミットを作成せず、ワークツリーに内容を取り込むには
「-n」オプションを利用する

Q116

「add-feature」ブランチの根元を、
「main」ブランチの指すコミットにrebaseする
ただし、HEADは「add-feature」ブランチを指しているものとする

A116
git rebase main

「git rebase コミット」というように実行すると、
現在HEADが指すブランチの根元を、引数で指定したコミットに移動することができる

引数にブランチが指定された場合は、
そのブランチが指すコミットに根本が移動するので
「git rebase main」とすると、
mainブランチが指すコミットに根元を移動する

Q117
「add-feature」ブランチの根元を、
「main」ブランチの指すコミットに移動（rebase）しようとしたところ、
「file」という名前のファイルにコンフリクトが発生して
手動で「file」を修正しなければいけなくなった
「file」を修正して、コンフリクトを解消した後に
実行するべきコマンドはなにか？
ただし、HEADは「add-feature」ブランチを指しているものとする

A117
git add file
からの
git rebase --continue

「git rebase main」を実行すると、
HEADが指すブランチの根元をmainブランチが指すコミットに移動（rebase）できる

rebaseはcherry-pickを繰り返し行うことでブランチの根元を移動するので
コンフリクトが発生する場合もある

その場合は、コンフリクトが発生したファイルを手動で修正した上で
「git add file」で変更をステージングエリアにaddした上で、
「git rebase --continue」でコミットを追加することができる

「git rebase --continue」を実行するとコミットメッセージの入力を求められる

Q118
「add-feature」ブランチの根元を、
「main」ブランチの指すコミットに移動（rebase）しようとしたところ、
「file」という名前のファイルにコンフリクトが発生した
コンフリクトの内容を見るとrebaseするブランチを
間違えていることが判明したのでrebaseをキャンセルしたい
このとき、実行するべきコマンドはなにか？

A118
```
git rebase --abort
```

「git rebase main」を実行すると、
HEADが指すブランチの根元をmainブランチが指すコミットに移動（rebase）できる

rebaseはcherry-pickを繰り返し行うことでブランチの根元を移動するので
コンフリクトが発生する場合もある

コンフリクト発生したところで間違いに気づき、
rebaseをキャンセルしたい場合は、
「git rebase --abort」を実行する
「abort」は「中止する」という意味

Q119
mergeと比較した時のrebaseのデメリットはなにか？

A119
rebaseには以下のようなデメリットもあるため、
トピックブランチにメインブランチを取り込む形で
マージする方がrebaseよりも良いことも多い
・rebaseは他ブランチの内容を取り込んだことが履歴として残らない
・コンフリクトを解消する回数が多くなりがち
・一度pushしたトピックブランチをrebaseするとpushできなくなる
・「git push -f」で無理やりpushするとリモートリポジトリを共有する他の開発者に影響がでる

ただし、GitHubで行うソーシャルコーディングでは、
rebaseしてからプルリクエストを送ることがマナーとされる

これはコミット履歴をきれいにするという目的と、
プルリクエストを受け取る側がコンフリクトを解消するのでなく、
プルリクエストを送る側がコンフリクトを解消するようにするという目的のため

Q120

rebaseの使いどころはどのような場面か？

A120
・マージの手間よりもコミット履歴のきれいさを優先したい
・まだrebaseするブランチをリモートにpushしていない
・pushしている場合は、リモートリポジトリを他の開発者と共有していない

Q121
既存のローカルリポジトリに
「origin」という名前のリモートリポジトリを追加する
ただし追加するリモートリポジトリのURLは、
「https://github.com/username/example.git」とする

A121
```
git remote add origin https://github.com/username/example.git
```

「git remote add リモートリポジトリ名 リモートリポジトリのURL」
とすることで、現在のローカルリポジトリにリモートリポジトリを設定できる

OSS貢献の際に、Fork元のリポジトリを追加する際によく使われる

Q122
ローカルリポジトリの「add-feature」ブランチの内容を
「origin」という名前のリモートリポジトリに強制的にpushする

A122
```
git push -f origin add-feature
```

「-f」は「forced（強制的な）」の意味

「-f」オプションを使うと、リモートリポジトリの履歴に関係なく、
ローカルリポジトリの履歴を優先してpushすることができる

すでにpushしているブランチをrebaseした後に使うことが多い

他の開発者に多大な迷惑をかけてしまうので、
他の開発者と共有しているブランチでは強制pushをしてはいけない