Git Version Control Quiz (100 Questions)

DevOps Learning Module

This quiz covers fundamental concepts related to the Git version control system. Choose the best answer for each question.

1. What command is used to initialize a new, empty Git repository?

   A. `git new`

   B. `git start`

   C. `git init`

   D. `git create`

   Answer: C
   Explanation: `git init` creates a new `.git` subdirectory in your current directory, which contains all the necessary repository files.

2. What command is used to check the status of your working directory and staging area?

   A. `git check`

   B. `git status`

   C. `git changes`

   D. `git diff`

   Answer: B
   Explanation: `git status` is one of the most common commands. It shows which files are modified, which are staged, and which are untracked.

3. What command is used to stage a file (add it to the index) for the next commit?

   A. `git commit`

   B. `git add`

   C. `git stage`

   D. `git include`

   Answer: B
   Explanation: `git add <file>` adds the file's current content to the "staging area" (or "index"), preparing it to be included in the next commit.

4. How do you stage *all* modified and new untracked files in the current directory?

   A. `git add .` or `git add -A`

B. `git add *`

C. `git add -all`

D. `git stage all`

Answer: A
Explanation: `git add .` stages all new and modified files in the current directory. `git add -A` (or `-all`) stages all changes in the entire repository.

5. What command is used to record the staged changes into the repository's history?

A. `git save`

B. `git store`

C. `git commit`

D. `git push`

Answer: C
Explanation: `git commit` takes all the changes from the staging area, creates a new "snapshot" (commit), and saves it to the local repository history.

6. How do you provide a commit message directly on the command line?

A. `git commit -message "My message"`

B. `git commit -m "My message"`

C. `git commit -msg "My message"`

D. `git commit "My message"`

Answer: B
Explanation: The `-m` flag allows you to provide a short commit message without opening the text editor.

7. What command is used to show the history of commits in the repository?

A. `git history`

B. `git log`

C. `git commits`

D. `git list`

Answer: B
Explanation: `git log` displays the commit history, showing the commit hash, author, date, and commit message for each commit.

8. What command is used to show the differences between your working directory and the staging area?

A. `git diff`

B. `git diff -staged`

C. `git diff HEAD`

D. `git status`

Answer: A
Explanation: Running `git diff` with no arguments shows the changes you've made to tracked files that you have *not yet* staged (added to the index).

9. What command is used to show the differences between the staging area and the last commit?

   A. `git diff`

   B. `git diff -staged` (or `-cached`)

   C. `git diff HEAD`

   D. `git diff -all`

   Answer: B
   Explanation: `git diff -staged` shows the changes that *are* in the staging area and will be part of the next commit.

10. What command is used to "unstage" a file (remove it from the staging area) but keep the changes in the working directory?

    A. `git unstage <file>`

    B. `git reset HEAD <file>`

    C. `git checkout - <file>`

    D. `git rm -cached <file>`

    Answer: B
    Explanation: `git reset HEAD <file>` (or just `git reset <file>`) is the command to unstage a file.

11. What command is used to discard all changes in a specific file in your working directory, reverting it to the version from the last commit?

    A. `git undo <file>`

    B. `git reset -hard <file>`

    C. `git checkout - <file>`

    D. `git clean <file>`

    Answer: C
    Explanation: This command is dangerous as it overwrites your local changes. It's used to "check out" the version of the file from the index (or HEAD).

12. What command is used to create a new branch?

    A. `git branch <branch-name>`

    B. `git new-branch <branch-name>`

    C. `git create <branch-name>`

D. `git checkout -n <branch-name>`

Answer: A
Explanation: `git branch <branch-name>` creates a new pointer (branch) to the current commit. It does *not* switch you to that branch.

13. What command is used to switch to a different branch?

    A. `git switch <branch-name>`

    B. `git checkout <branch-name>`

    C. `git move <branch-name>`

    D. Both A and B.

Answer: D
Explanation: `git checkout <branch-name>` is the classic command. The newer `git switch <branch-name>` command was introduced to separate "switching branches" from "checking out files".

14. What command is used to create a new branch AND switch to it in one step?

    A. `git branch -n <branch-name>`

    B. `git checkout -b <branch-name>`

    C. `git new <branch-name>`

    D. `git switch -c <branch-name>`

Answer: B
Explanation: `git checkout -b <branch-name>` is the traditional shortcut. The modern equivalent is `git switch -c <branch-name>`.

15. What command is used to combine the history of one branch into your current branch?

    A. `git combine <branch-name>`

    B. `git merge <branch-name>`

    C. `git rebase <branch-name>`

    D. Both B and C.

Answer: D
Explanation: `git merge` and `git rebase` are the two primary ways to integrate changes from one branch into another.

16. What is the result of a `git merge`?

    A. It rewrites the commit history to be linear.

    B. It creates a new "merge commit" that joins the two branches.

    C. It deletes the other branch.

    D. It squashes all commits into one.

Answer: B
Explanation: A `git merge` (specifically a non-fast-forward merge) creates a new commit that has two parent commits, preserving the history of both branches.

17. What is the result of a `git rebase`?

    A. It creates a new "merge commit".

    B. It moves the *entire* current branch to begin on the tip of the target branch, creating a linear history.

    C. It is the same as a merge.

    D. It deletes the target branch.

    Answer: B
    Explanation: `git rebase` rewrites commit history. It takes all the commits from your branch and replays them, one by one, on top of the target branch, resulting in a "cleaner," linear history.

18. Why should you generally *not* rebase a branch that is shared with other people (e.g., `main`)?

    A. It is slower than merging.

    B. It can cause merge conflicts.

    C. Because it rewrites history, it will cause major problems for anyone who has already pulled the old history.

    D. It is not a feature of GitHub.

    Answer: C
    Explanation: This is the golden rule of rebasing. Only rebase your local, private branches. Rebasing a public, shared branch rewrites its history, which will conflict with everyone else's copy.

19. What command is used to connect your local repository to a remote repository (like on GitHub)?

    A. `git remote add <name> <url>`

    B. `git connect <name> <url>`

    C. `git link <name> <url>`

    D. `git setup remote <url>`

    Answer: A
    Explanation: This command adds a new remote, giving it a short "name" (like `origin`) that points to the specified `url`.

20. What is the default "name" for the remote repository you cloned from?

    A. `remote`

    B. `github`

C. `main`

D. `origin`

Answer: D
Explanation: By convention, when you `git clone` a repository, Git automatically sets up a remote named `origin` pointing back to the URL you cloned from.

21. What command is used to download changes from a remote repository's branch into your local repository?

A. `git fetch <remote>`

B. `git pull <remote>`

C. `git download <remote>`

D. `git sync <remote>`

Answer: A
Explanation: `git fetch` downloads all the new commits, branches, and tags from the remote, but it does *not* merge them into your local working branch.

22. What command is used to download changes from a remote repository AND merge them into your current branch?

A. `git fetch`

B. `git pull`

C. `git sync`

D. `git update`

Answer: B
Explanation: `git pull` is essentially a `git fetch` followed immediately by a `git merge` (or `rebase`).

23. What command is used to upload your local commits to a remote repository?

A. `git upload <remote> <branch>`

B. `git send <remote> <branch>`

C. `git push <remote> <branch>`

D. `git sync <remote> <branch>`

Answer: C
Explanation: `git push` (e.g., `git push origin main`) uploads your branch's commits to the remote repository.

24. What is the `.gitignore` file for?

A. A list of files that Git should delete.

B. A list of files and patterns that Git should intentionally ignore (not track).

C. A list of files that have errors.

D. A file that configures Git settings.

Answer: B
Explanation: You add patterns like `node_modules/`, `*.log`, or `.env` to `.gitignore` to prevent build artifacts, log files, and secrets from being committed.

25. What command is used to create a copy of an existing remote repository on your local machine?

    A. `git copy <url>`

    B. `git duplicate <url>`

    C. `git clone <url>`

    D. `git init -remote <url>`

    Answer: C
    Explanation: `git clone` downloads the entire repository, creates a local copy, and automatically sets up `origin` to point back to the source.

26. What is a "merge conflict"?

    A. A bug in Git.

    B. When two branches have commits that are out of order.

    C. When Git is unable to automatically merge two branches because they both changed the same line of a file.

    D. When two people try to push at the same time.

    Answer: C
    Explanation: A merge conflict occurs when Git doesn't know which change to keep. It stops the merge and inserts markers (`«««<`, `=====`, `»»»>`) in the file for you to resolve manually.

27. After you have manually resolved a merge conflict in a file, what is the *next* step?

    A. `git merge -continue`

    B. `git add <file>`

    C. `git commit`

    D. `git rebase -continue`

    Answer: B
    Explanation: After editing the file to fix the conflict, you must run `git add <file>` to mark the conflict as resolved and stage the newly merged content.

28. After resolving all conflicts and adding all files, what command completes the merge?

    A. `git merge -finish`

    B. `git commit`

    C. `git rebase -continue`

D. `git push`

Answer: B
Explanation: Running `git commit` (with no other arguments) will create the new merge commit, completing the process.

29. What is `git stash` used for?

    A. To delete all your local changes.

    B. To commit your changes.

    C. To temporarily save your uncommitted local changes (in the working directory and staging area) so you can switch branches.

    D. To hide a commit.

    Answer: C
    Explanation: If you're in the middle of work but need to switch branches, you can `git stash` your changes. This saves them and reverts your directory to `HEAD`, leaving it clean.

30. How do you retrieve and re-apply the changes you just saved with `git stash`?

    A. `git stash apply`

    B. `git stash pop`

    C. `git stash get`

    D. Both A and B.

    Answer: D
    Explanation: `git stash apply` re-applies the changes but leaves them in the stash. `git stash pop` re-applies the changes AND removes them from the stash list.

31. What is a "tag" in Git?

    A. A type of branch.

    B. A text label in a commit message.

    C. A permanent, human-readable pointer to a specific commit, often used to mark a release (e.g., `v1.0.0`).

    D. A way to group commits.

    Answer: C
    Explanation: Unlike a branch (which moves), a tag is a fixed pointer to a single commit, used to mark important points in history like a release version.

32. How do you create an "annotated" tag?

    A. `git tag -a v1.0 -m "My version 1.0"`

    B. `git tag v1.0`

    C. `git tag -annotated v1.0`

D. `git tag -m "My version 1.0" v1.0`

Answer: A
Explanation: An annotated tag (`-a`) is a full Git object. It stores a message, author, and date, and is generally recommended for releases.

33. Does `git push` automatically push your tags to the remote?

    A. Yes, always.

    B. No, you must push tags explicitly with `git push -tags`.

    C. Only if they are annotated tags.

    D. Only if you push the `main` branch.

    Answer: B
    Explanation: Tags are not pushed by default. You must use `git push origin <tagname>` for a single tag, or `git push -tags` to push all tags.

34. What is `git log -oneline`?

    A. It shows only the most recent commit.

    B. It shows the log in a compact, single-line-per-commit format.

    C. It shows the log for a single file.

    D. It opens the log in a text editor.

    Answer: B
    Explanation: This is a very useful command for getting a quick overview of the commit history, showing just the commit hash (shortened) and the commit message.

35. What is `git log -graph`?

    A. It displays a graphical (GUI) version of the log.

    B. It shows a bar graph of commits per author.

    C. It draws an ASCII art graph showing the branching and merging history.

    D. It outputs the log in JSON format.

    Answer: C
    Explanation: This is extremely useful for visualizing how branches were created, merged, and rebased. It's often combined with `-oneline -decorate`.

36. What is `git rm <file>`?

    A. It removes the file from the staging area, but not the working directory.

    B. It removes the file from the working directory and stages that deletion.

    C. It discards changes to the file.

    D. It renames the file.

Answer: B
Explanation: This command is like running `rm <file>` in the terminal, and then running `git add <file>` to stage the deletion.

37. What is `git rm -cached <file>`?

    A. It removes the file from the working directory, but not the staging area.

    B. It removes the file from version control (the staging area), but leaves it in the working directory.

    C. It deletes the file from the last commit.

    D. It caches the file for later.

    Answer: B
    Explanation: This is used when you've accidentally tracked a file (like a log file) and want to stop tracking it (and add it to `.gitignore`), but you don't want to delete the local copy.

38. What does `git commit -amend` do?

    A. It creates a brand new commit.

    B. It lets you edit the commit message of the *previous* commit.

    C. It lets you add more staged changes to the *previous* commit, rewriting it.

    D. Both B and C.

    Answer: D
    Explanation: This command is a powerful way to fix a mistake in your last commit. It replaces the previous commit with a new one that includes any new staged changes and/or a new commit message.

39. What is `git rebase -i` (interactive rebase)?

    A. A command that opens a text editor, allowing you to reorder, squash, edit, or delete recent commits.

    B. A command to merge two branches.

    C. A command to run rebase with a GUI.

    D. A command to integrate Hiera data.

    Answer: A
    Explanation: Interactive rebase is a powerful history-editing tool. It's often used to "clean up" a feature branch's history before merging it.

40. In an interactive rebase, what does the "squash" command do?

    A. It deletes the commit.

    B. It pauses the rebase at that commit.

    C. It melts the commit into the commit immediately preceding it, combining their changes and messages.

D. It splits the commit into two.

Answer: C
Explanation: "Squashing" is used to combine multiple small "work-in-progress" commits into a single, meaningful commit.

41. What is `HEAD` in Git?

    A. The first commit in the repository.

    B. The most recent commit on the `main` branch.

    C. A pointer to the branch or commit you are currently checked out to.

    D. A pointer to the staging area.

    Answer: C
    Explanation: `HEAD` is a symbolic reference that points to your current location. When you switch branches, `HEAD` moves to point to that branch's tip.

42. What does `HEAD~1` (or `HEAD^`) refer to?

    A. The staging area.

    B. The first commit.

    C. The commit *before* `HEAD` (the parent of `HEAD`).

    D. The next commit (which doesn't exist yet).

    Answer: C
    Explanation: The tilde (`~`) and caret (`^`) notations are used for navigating history. `HEAD~1` means "the first parent of HEAD." `HEAD~2` means "the grandparent of HEAD."

43. What is `git reset -soft HEAD~1`?

    A. It deletes the last commit and all its changes.

    B. It "undoes" the last commit, but leaves all the changes from that commit in the staging area.

    C. It "undoes" the last commit, but leaves all the changes in the working directory (unstaged).

    D. It moves the `HEAD` pointer, but leaves the staging area and working directory as they were.

    Answer: B
    Explanation: This is a very useful command. It rewinds the commit history by one, but "keeps" all the changes from that commit staged and ready to be re-committed.

44. What is `git reset -mixed HEAD~1` (or just `git reset HEAD~1`)?

    A. It deletes the last commit and all its changes.

    B. It "undoes" the last commit, but leaves all the changes from that commit in the staging area.

C. It "undoes" the last commit, but leaves all the changes in the working directory (unstaged).

D. It is not a valid command.

Answer: C
Explanation: This is the default reset mode. It "undoes" the last commit and also "unstages" all the changes, leaving them as modified files in your working directory.

45. What is `git reset -hard HEAD~`?

A. It "undoes" the last commit and permanently deletes all changes from that commit.

B. It "undoes" the last commit, but leaves all the changes from that commit in the staging area.

C. It "undoes" the last commit, but leaves all the changes in the working directory (unstaged).

D. It is a very safe command.

Answer: A
Explanation: This is a very dangerous command. It not only removes the last commit but also erases all the changes (from the staging area AND working directory) associated with it.

46. What command is used to find which commit introduced a bug?

A. `git blame`

B. `git search -bug`

C. `git bisect`

D. `git find-bug`

Answer: C
Explanation: `git bisect` is a powerful tool that performs a binary search on your commit history. You give it a "good" commit and a "bad" commit, and it helps you find the exact commit that introduced the problem.

47. What command is used to show who last modified each line of a file?

A. `git blame <file>`

B. `git author <file>`

C. `git log <file>`

D. `git who <file>`

Answer: A
Explanation: `git blame` annotates every line in a file with the commit hash, author, and date of the last change. It's used to find the context of a specific line of code.

48. What is a "remote-tracking" branch (like `origin/main`)?

A. The same as your local `main` branch.

B. A read-only pointer that shows you where the `main` branch was on the remote (`origin`) the last time you fetched.

C. A branch that you cannot see.

D. A branch that is used to send tracking data to GitHub.

Answer: B
Explanation: You don't edit this branch. When you `git fetch`, Git updates `origin/main` to point to the latest commit from the remote. You then merge `origin/main` into your local `main`.

49. What is `git checkout -t origin/feature`?

A. It creates a new tag.

B. It deletes the remote branch.

C. It creates a new local branch named `feature` that "tracks" the remote branch `origin/feature`.

D. It runs tests on the branch.

Answer: C
Explanation: This is a common shortcut. It creates a local `feature` branch and sets it up to "track" the remote one, so `git pull` and `git push` work without extra arguments.

50. What is a "detached HEAD" state?

A. When your `HEAD` pointer is pointing directly to a commit, not to a branch.

B. When your `.git` directory is corrupted.

C. When you are not on any branch.

D. Both A and C are correct.

Answer: D
Explanation: This happens if you `git checkout` a commit hash or a tag. You are in a "detached HEAD" state. Any new commits you make will be lost unless you create a new branch.

51. What command is used to apply a specific commit from one branch to another?

A. `git copy-commit <hash>`

B. `git apply <hash>`

C. `git rebase -onto <hash>`

D. `git cherry-pick <hash>`

Answer: D
Explanation: `git cherry-pick` takes the changes from a single commit and applies them as a new commit on your current branch.

52. What is `git revert <hash>`?

A. It deletes the specified commit.

B. It creates a *new* commit that is the exact opposite of the specified commit, effectively undoing it.

C. It resets your `HEAD` to that commit.

D. It re-applies that commit.

Answer: B
Explanation: This is the "safe" way to undo a commit in a shared history. Instead of rewriting history (like `git reset`), `git revert` adds a new commit that reverses the changes.

53. What is a "submodule" in Git?

A. A branch that is inside another branch.

B. A way to embed another Git repository inside your main repository.

C. A part of a `.git` directory.

D. A small commit.

Answer: B
Explanation: Submodules allow you to keep one Git repo as a subdirectory of another Git repo, while keeping their histories separate.

54. What is `git config -global`?

A. It sets a configuration value for *all* repositories on your system (for your user).

B. It sets a configuration value for *only* the current repository.

C. It shows all configuration values.

D. It resets the configuration.

Answer: A
Explanation: This is used for settings you want to apply everywhere, like your `user.name` and `user.email`. The settings are saved in `~/.gitconfig`.

55. How do you set your email address for all repositories?

A. `git config -global user.email "you@example.com"`

B. `git config -global email "you@example.com"`

C. `git set email "you@example.com"`

D. `git config user.email "you@example.com"`

Answer: A
Explanation: This, along with `user.name`, is one of the first two commands you should run after installing Git.

56. What is a "hook" in Git?

A. A merge conflict.

B. A script in the `.git/hooks` directory that runs automatically at certain points (e.g., pre-commit, post-commit).

C. A way to link repositories.

D. A web service.

Answer: B
Explanation: Hooks are scripts that "hook into" Git's workflow. A `pre-commit` hook, for example, could run a linter and block the commit if it fails.

57. What is `git reflog`?

A. The same as `git log`.

B. A log of all remote references.

C. A log of *all* movements of `HEAD`, including commits, checkouts, and resets.

D. A log of all references to your repository.

Answer: C
Explanation: This is Git's "safety net." If you "lose" a commit (e.g., with `git reset -hard`), you can find it in the `git reflog` and restore it.

58. What is `git clean -fd`?

A. A command to clean up old commits.

B. A command to delete all untracked files and directories.

C. A command to fix (f) and debug (d) the repository.

D. A command to fetch and download.

Answer: B
Explanation: This is a very destructive command. `-f` (force) is required, and `-d` (directories) tells it to also remove untracked directories.

59. What is a "fast-forward" merge?

A. A merge that creates a merge commit.

B. A merge where the target branch's tip is a direct ancestor of the current branch's tip. Git just moves the pointer.

C. A merge that happens on the server.

D. A merge that skips testing.

Answer: B
Explanation: If you are on `main` and merge `feature`, and `main` has no new commits, Git will just move the `main` pointer to `feature`'s tip. No "merge commit" is needed.

60. How do you force `git merge` to create a merge commit, even if it's a fast-forward?

A. `git merge -no-ff`

B. `git merge -force`

C. `git merge -create-commit`

D. You cannot do this.

Answer: A
Explanation: The `-no-ff` (no fast-forward) flag forces Git to create a merge commit. This is often used to keep a record of when a feature branch was merged.

61. What is a "Pull Request" (PR)?

A. It is a `git pull` command.

B. It is a core Git command.

C. A feature of hosting platforms (like GitHub/GitLab) to propose and discuss changes before they are merged.

D. A request to download a repository.

Answer: C
Explanation: A Pull Request (or Merge Request) is not a Git feature, but a web-based feature. It's a formal way to say, "I've pushed a branch, please review my changes and merge them."

62. What is a "Fork"?

A. A merge conflict.

B. A feature of Git to split a branch.

C. A feature of hosting platforms (like GitHub) to create a personal, server-side copy of someone else's repository.

D. A `git clone` command.

Answer: C
Explanation: Forking is how you contribute to open-source projects. You "fork" the project (creating a copy in your account), `clone` your fork, push changes to your fork, and then open a Pull Request.

63. What is the `git remote -v` command used for?

A. To remove a remote.

B. To rename a remote.

C. To show a verbose list of all remotes and their URLs.

D. To validate a remote's URL.

Answer: C
Explanation: The `-v` (verbose) flag shows you the "name" of the remote (e.g., `origin`) and the URLs it uses for (fetch) and (push).

64. What does `git pull -rebase` do?

A. It runs `git fetch` followed by `git rebase` (instead of `git merge`).

B. It runs `git pull` and then starts an interactive rebase.

C. It is not a valid command.

D. It rewrites the remote repository's history.

Answer: A
Explanation: This is a very common workflow. It pulls the remote changes and "replays" your local, un-pushed commits *on top* of them, maintaining a linear history.

65. What is `git show <hash>`?

A. It shows the log for that commit.

B. It shows the metadata and the "diff" (changes) for a specific commit.

C. It checks out that commit.

D. It shows the author of the commit.

Answer: B
Explanation: `git show` is the easiest way to see the details (author, date, message) and the exact changes introduced by a single commit.

66. What is the difference between `git remote prune` and `git fetch -prune`?

A. There is no difference.

B. `remote prune` is safer.

C. `fetch -prune` deletes remote-tracking branches that no longer exist on the remote *before* fetching.

D. `remote prune` deletes local branches.

Answer: C
Explanation: If a branch is deleted on the remote, `git fetch -prune` (or `git remote prune origin`) will remove your local `origin/deleted-branch` remote-tracking branch.

67. How do you delete a local branch?

A. `git branch -d <branch-name>`

B. `git branch -D <branch-name>`

C. `git delete <branch-name>`

D. Both A and B.

Answer: D
Explanation: `-d` (delete) is the "safe" delete; it won't delete the branch if it has unmerged changes. `-D` (force delete) will delete it regardless.

68. How do you delete a remote branch?

A. `git push origin -delete <branch-name>`

B. `git branch -d -r <branch-name>`

C. `git remote delete <branch-name>`

D. `git push origin :<branch-name>`

Answer: A
Explanation: This is the modern, easy-to-read command. (D is the older, more cryptic syntax for the same action).

69. What is "Git LFS"?

    A. Git "Large File Storage": An extension for versioning large binary files.

    B. Git "Lightweight File System".

    C. Git "Local File Storage".

    D. A hosting provider.

    Answer: A
    Explanation: Git is bad at handling large binary files. LFS stores these files on a separate server and places small "pointer" files in the Git repository instead.

70. What is `git mv`?

    A. A command to move a commit.

    B. A command to rename a file or move it to a new directory.

    C. A command to move a branch.

    D. A command to change a remote's URL.

    Answer: B
    Explanation: `git mv old.txt new.txt` is equivalent to running `mv old.txt new.txt`, `git rm old.txt`, and `git add new.txt` all in one step.

71. What is the "index" in Git?

    A. The `.git` directory.

    B. Another name for the "staging area".

    C. The history of all commits.

    D. A file containing a list of all branches.

    Answer: B
    Explanation: The "index" and "staging area" are interchangeable terms. It's the area where you build up your next commit.

72. What is the `git filter-branch` command used for?

    A. To filter log output.

    B. A complex, powerful (and slow) command to rewrite the entire repository history.

    C. To filter which branches are fetched.

    D. To find a specific commit.

    Answer: B
    Explanation: This is a "heavy-duty" command for tasks like removing a large file or

a password from *every* commit in the history. (The newer, faster alternative is `git filter-repo`).

73. What is a "bare" repository?

    A. A repository with only one branch.

    B. A repository that has no commits.

    C. A repository that has no "working directory" (no checked-out files), just the `.git` data.

    D. A repository with no `.gitignore`.

    Answer: C
    Explanation: Bare repositories (created with `git init -bare`) are used on servers. You can't work in them; you can only `push` to them and `fetch` from them.

74. What is `git revert HEAD`?

    A. It deletes the last commit.

    B. It creates a new commit that undoes the changes from the last commit.

    C. It discards all local changes.

    D. It resets the `HEAD` pointer.

    Answer: B
    Explanation: This command creates a new commit that reverses the last commit, and is a safe way to "undo" in a shared history.

75. What is "GitFlow"?

    A. A core Git command.

    B. A Git GUI client.

    C. A popular, but complex, branching model using `main`, `develop`, `feature`, `release`, and `hotfix` branches.

    D. A continuous integration tool.

    Answer: C
    Explanation: GitFlow is a strict, formalized workflow. It's often contrasted with simpler workflows like "GitHub Flow" (which just uses `main` and feature branches).

76. What is `git remote set-url`?

    A. To set the URL for your Git provider.

    B. To change the URL of an existing remote.

    C. To create a new remote.

    D. To validate a remote's URL.

Answer: B
Explanation: If your repository moves (e.g., from HTTP to SSH), you use `git remote set-url origin <new-url>` to update the remote's URL.

77. What is `git log -p`?

    A. It shows the log for a specific "path".

    B. It shows the log in "patch" (diff) format, including the changes for each commit.

    C. It "prepares" the log for pushing.

    D. It shows a "pretty" log.

    Answer: B
    Explanation: The `-p` (patch) flag is very useful. It shows the log, but also includes the full `diff` of the changes introduced by each commit.

78. What is `git diff <commit1>..<commit2>`?

    A. It shows all commits between the two commits.

    B. It shows the changes (a "diff") between the two commits.

    C. It merges the two commits.

    D. It is not valid syntax.

    Answer: B
    Explanation: This command shows the *difference* between the state of the repository at `commit1` and the state at `commit2`.

79. What is `git add -p`?

    A. It adds all files in a "path".

    B. It "patches" the commit.

    C. It interactively "patches" the add, allowing you to stage parts of a file (hunks) instead of the whole file.

    D. It "prepares" the files to be added.

    Answer: C
    Explanation: This is an extremely useful command. It walks you through every "hunk" (chunk of changes) in your modified files and asks you (y/n/s) if you want to stage it.

80. What is a "commit hash"?

    A. A short, 7-character ID for a commit.

    B. A 40-character SHA-1 checksum that uniquely identifies a commit.

    C. A password for a commit.

    D. A type of merge conflict.

Answer: B
Explanation: The SHA-1 hash is a unique ID for a commit, generated from its content (the changes, the parent, the author, the date, etc.).

81. What is `git grep`?

    A. It's the same as the regular `grep` command.

    B. It's a command to search for a string or regex in all *tracked* files in your repository.

    C. It's a command to search your commit messages.

    D. It's a command to search your `.gitignore` file.

    Answer: B
    Explanation: `git grep` is a very fast and powerful way to search your project's codebase, because it only searches files that are tracked by Git.

82. How do you search your *commit messages* for a string?

    A. `git grep -messages "foo"`

    B. `git log -grep="foo"`

    C. `git search -message "foo"`

    D. `git log "foo"`

    Answer: B
    Explanation: The `-grep` flag for `git log` filters the log to show only commits where the commit message matches the provided string.

83. What is `git shortlog`?

    A. A command that shows a very short log (`-oneline`).

    B. A command that summarizes `git log` output, grouping commits by author.

    C. A command to shorten a commit message.

    D. A command to log a short message.

    Answer: B
    Explanation: This is a great command for seeing who has been contributing to a project. `git shortlog -s -n` is a common way to see a list of contributors, sorted by commit count.

84. What is `git archive`?

    A. A command to move old commits to an archive.

    B. A command to create a `.zip` or `.tar.gz` file of the repository's contents.

    C. A command to set a repository to read-only.

    D. A command to manage `git lfs`.

Answer: B
Explanation: `git archive -o latest.zip HEAD` will create a `.zip` file of the current commit, *without* the `.git` directory. This is great for creating release artifacts.

85. What is the difference between `git fetch` and `git pull`?

    A. They are exactly the same.

    B. `git pull` is safer.

    C. `git fetch` only downloads new data; `git pull` downloads AND merges it.

    D. `git fetch` only works on the `main` branch.

    Answer: C
    Explanation: `git fetch` updates your remote-tracking branches (like `origin/main`) but doesn't touch your local `main`. `git pull` does a `fetch` *and* a `merge`.

86. What is `git remote show <remote-name>`?

    A. It shows the URL of the remote.

    B. It shows all remotes.

    C. It shows detailed information about a remote, including its URLs and which branches are tracked.

    D. It shows the remote's `.git` config.

    Answer: C
    Explanation: This command provides a detailed summary of a remote, showing the fetch/push URLs and which remote branches are being tracked.

87. What is a "merge squash"?

    A. A merge that fails.

    B. A merge that takes all commits from a feature branch and combines (squashes) them into a *single* new commit on the target branch.

    C. A merge that deletes the feature branch.

    D. A type of interactive rebase.

    Answer: B
    Explanation: When you merge a branch with `git merge -squash`, Git stages all the changes, but doesn't create a merge commit. This allows you to create a single, clean commit.

88. What is "cherry-picking"?

    A. Selecting the best commits for a release.

    B. Using `git cherry-pick` to apply a single commit from one branch onto another.

    C. A type of merge conflict.

    D. Deleting bad commits.

Answer: B
Explanation: If you have a bug fix on a feature branch that you also need on `main`, you can "cherry-pick" that one commit over without merging the whole branch.

89. What is `git worktree`?

    A. A file that lists all files in the repository.

    B. A command that lets you have multiple working directories (for different branches) linked to a single `.git` repository.

    C. A graphical tool for viewing the commit tree.

    D. A command to clean the working directory.

    Answer: B
    Explanation: This is an advanced feature. It lets you check out `main` in one directory and `feature-A` in another directory *at the same time*, without stashing or re-cloning.

90. What is the `git bisect start` command used for?

    A. To start a new repository.

    B. To start a `git daemon`.

    C. To begin a "binary search" session to find a bug.

    D. To split a commit.

    Answer: C
    Explanation: After `git bisect start`, you give it a `git bisect good <hash>` and `git bisect bad <hash>`. Git then checks out a commit in the middle for you to test.

91. What is `git branch -vv`?

    A. It creates two branches.

    B. It is "very verbose" and shows local branches, their tracked remote branches, and their ahead/behind status.

    C. It is not a valid command.

    D. It deletes a branch very forcefully.

    Answer: B
    Explanation: This is a very useful command to see the "upstream" tracking relationship for all your local branches and if they are in sync with the remote.

92. What is `git shortlog -s -n`?

    A. It shows a summary (`-s`) of commits, sorted numerically (`-n`) by author.

    B. It shows a short log of new (`-n`) and "squashed" (`-s`) commits.

    C. It's not a valid command.

    D. It shows a log of the last `-n` commits.

Answer: A
Explanation: This is the go-to command for generating a "Contributors" list, as it shows a count of commits next to each author's name.

93. What does `git fetch origin -prune` do?

    A. It prunes (deletes) the `origin` remote.

    B. It deletes (prunes) local branches that have been merged.

    C. It deletes remote-tracking branches (like `origin/old-feature`) if they no longer exist on the remote.

    D. It prunes the reflog.

    Answer: C
    Explanation: This is a housekeeping command. It ensures your list of remote-tracking branches stays in sync with the branches that *actually* exist on the remote.

94. What is the `git merge-base` command?

    A. A tool for merging `main` into your branch.

    B. A command that finds the best common ancestor commit between two branches.

    C. A command to set the default merge strategy.

    D. A command to start a merge.

    Answer: B
    Explanation: This is a "plumbing" command often used in scripts. It tells you "where" two branches diverged from, which is the base commit used for a 3-way merge.

95. What is `gitk`?

    A. A command-line tool for logging.

    B. A built-in, lightweight graphical tool for viewing repository history.

    C. A Git hosting provider.

    D. The kernel of Git.

    Answer: B
    Explanation: `gitk` is a simple, Tcl/Tk-based GUI that provides a graphical view of the commit log, branches, and diffs.

96. What is the `.git/config` file?

    A. The global configuration file.

    B. The system-wide configuration file.

    C. The *local*, repository-specific configuration file.

    D. The file that stores your `.gitignore` rules.

Answer: C
Explanation: This file contains configuration settings (like remotes) that apply *only* to the current repository, overriding global settings in `~/.gitconfig`.

97. What is `git log -all`?

    A. It shows the log for all files.

    B. It shows the log for all authors.

    C. It shows the log for all branches, not just the current one.

    D. It shows all configuration settings.

    Answer: C
    Explanation: By default, `git log` only shows the history of the branch you're on. `-all` shows the history of *all* branches and tags.

98. What is `git log -pretty=format:"%h - %an, %ar :  %s"`?

    A. A command to format the `git log` output.

    B. A command to find a commit by format.

    C. A command to reformat a commit message.

    D. A command to set the global log format.

    Answer: A
    Explanation: The `-pretty=format` string allows you to completely customize the log output. (`%h` = short hash, `%an` = author name, `%ar` = author date relative, `%s` = subject).

99. What is "GitHub Actions"?

    A. A set of Git commands.

    B. A CI/CD (Continuous Integration / Continuous Deployment) platform built into GitHub.

    C. A graphical Git client.

    D. A tool for managing permissions.

    Answer: B
    Explanation: GitHub Actions is a platform for automating workflows, such as running tests when you push a commit or deploying to production when you create a tag.

100. What is `git gui`?

    A. A built-in, lightweight graphical tool primarily used for staging changes and making commits.

    B. A command to view the commit graph.

    C. A web-based interface.

    D. A command to configure the GUI.

Answer: A
Explanation: `git gui` is another simple, Tcl/Tk-based GUI (like `gitk`) that is focused on the "committing" part of the workflow: staging, unstaging, and committing.