The first thing to know is that git pull does too much stuff. Well, for some people (me), it does too much; others like that it does this much; but in fact, it does two jobs, each of which has its own separate Git command:

1. git pull runs git fetch. Most, but not all, of the arguments you give *to* git pull are passed directly to git fetch. So git pull means *run git fetch* and git pull origin somebranch means *run git fetch origin somebranch*.
2. Assuming the first step succeeds, git pull runs a second Git command.

The reason to have a step 2 at all is simple enough: git fetch *obtains new commits* from some other Git repository, stuffing those new commits into your own repository where you now have access to them. But then it *stops*. You *have access to* the new commits, but *nothing is actually using* the new commits yet. To *use* the new commits, you need a second step.

Initially, that second step was always git merge. The git merge command is pretty big and complicated but it has a meaning that's pretty simple to describe: ***Merge* means *combine work*.** Git will attempt to take work you have done, if you have done any, and work they have done, if they have done any, and combine the work, using simple and stupid automated rules. These rules have no clue as to how or why you did the work, or what anything you changed *means*. They just work based on "lines" in diffs.

There are, however, four possibilities here:

- Perhaps you did no work and they did no work. You got no new commits. There's literally nothing to do, and git merge does nothing.
- Perhaps you did some work and they did nothing; you got no new commits; there's nothing to do and git merge does nothing again.
- Perhaps you did no work and they did do some work. You got some new commits. Combining your lack-of-work with their actual work is easy and git merge will take a shortcut if you allow it.
- Perhaps you and they both did work. You have new commits *and* you got new commits from them, and git merge has to use its simple-and-stupid rules to combine the work. Git cannot take any shortcuts here and you will get a full-blown merge.

The shortcut that Git *may* be able to take is to simply check out their latest commit while dragging your branch name forward. The git merge command calls this a *fast-forward merge*, although there's no actual *merging* involved. This kind of not-really-a-merge is trivial, and normally extremely safe: the only thing that *can* go wrong is if their latest commit doesn't actually function properly. (In that case, you can go back to the older version that does.) So a "fast forward" merge is particularly friendly: there's no complicated line-by-line merging rules that can go awry. Many people like this kind of "merge".

Sometimes the shortcut is not possible, and sometimes some people don't *want* Git to take the shortcut (for reasons we won't cover here to keep this answer short, or short for me anyway). There is a way to tell git merge *do not take the shortcut, even if you can*.

So, for git merge alone, that gives us three possibilities:

- nothing to do (and git merge is always willing to do nothing);
- fast-forward is possible, but maybe Git shouldn't do it; and
- fast-forward is not possible, which means this merge isn't trivial.

The git merge command has options to tell it what to do in all but the "nothing to do" case:

- (no flags): do a fast-forward if possible, and if not, attempt a real merge.
- --ff-only: do a fast-forward if that's possible. If not, give an error stating that fast-forward is not possible; do not attempt a merge.
- --no-ff: even if a fast-forward is possible, don't use the shortcut: attempt a full merge in every case (except of course the "nothing to do" case).

The git pull command accepts all of these flags and will pass them on to git merge, should you choose to have git pull run git merge as its step 2.

But wait, there's more

Not everyone wants Git to do merges. Suppose you have made one or two new commits, which we'll call I and J, and your git fetch from origin brings in two new commits that *they* made since you started, which we will call K and L. That gives you a set of commits that, if you were to draw them, might look like this:

```
       I--J   <-- your-branch
      /
...--G--H   <-- main
      \
       K--L   <-- origin/main
```

You can *fast-forward* your main to match their origin/main:

```
       I--J   <-- your-branch
      /
...--G--H
      \
       K--L   <-- main, origin/main
```

And, whether or not you do that, you can *merge* your commit J with their commit L to produce a new *merge commit* M:

```
       I--J
      /    \
...--G--H      M   <-- your-branch (HEAD)
      \    /
       K--L   <-- origin/main
```

But some people prefer to *rebase* their commits—in this case I and J—so that they come *after* commit L, so that the picture now looks like this:

```
       I--J   [abandoned]
      /
...--G--H--K--L   <-- origin/main
            \
             I'-J'  <-- your-branch
```

Here, we have *copied* commits I and J to new-and-improved commits I' and J'. These commits make the same *changes* to L that I-J made to H, but these commits have different big-ugly-hash-IDs and look like you made them *after* the origin guys made their K-L commits.

The git pull command can do this kind of rebasing:

```
git switch your-branch
git pull --rebase origin main
```

does this all in one shot, by running git fetch to get their commits, then running git rebase with the right arguments to make Git copy I-J to I'-J' as shown above. Once the rebase is done—remember that, like git merge, it may have merge conflicts that you have to solve first—Git will move the branch name your-branch to select the last copied commit: J' in this example.

Not very long after git pull was written, this --rebase was added to it. And since many people want this sort of thing to happen *automatically*, git pull gained the ability to *default* to using --rebase. You configured your branch to do this (by setting branch.*branch*.rebase to true) and git pull would do a rebase for you. (Note that the commit on which your rebase occurs now depends on two things: the *upstream* setting of the branch, and some of the arguments you can pass to git pull. I've kept things explicit in this example so that we do not have to worry about smaller details, but in practice, you do.)

This brings us to 2006 or 2008 or so

At this point in Git's development, we have:

- git fetch: obtains new commits from somewhere else (an "upstream" or origin repository for instance), often updating origin/* style remote-tracking names;
- git merge: does nothing, or a fast-forward, or a true merge, of some specified commit or the branch's upstream;
- git rebase: copies some set of existing commits to new-and-improved commits, using a specified commit or the branch's upstream, then abandons the original commits in favor of the copies; and
- git pull: using the branch's upstream or explicit arguments, run git fetch and then run either git merge or git rebase.

Because git merge can take --ff-only or --no-ff arguments, git pull must be able to pass these to git merge *if* we're using git merge.

As time goes on, more options start appearing, such as auto-stashing, rebase's "fork point", and so on. Also, it turns out that many people *want* rebasing to be *their default* for git pull, so Git acquires a new configuration option, branch.autoSetupRebase. When set to remote or always, this does what many of these folks want (though there are actually four settings today; I don't remember if it had four back then and have not bothered to check).

Time continues marching on and we reach the 2020s

By now—some time between 2020 and 2022—it has become clear that git pull does the *wrong thing* for many, maybe even most, people who are new to Git. My personal recommendation has been to *avoid* git pull. Just don't use it: run git fetch first, then look at what git fetch said. Then, if git fetch did a lot, maybe use git log next. And *then*, once you're sure whether you want git merge with whatever options, or git rebase also with whatever options, *run that command*. **If you use this option, you are in full control. You dictate what happens, rather than getting some surprise from Git.** I like this option: it's simple! You do need to run at least two commands, of course. But you *get to* run additional commands *between* those two, and that can be useful.

Still, if a git pull brings in new commits that *can* be merged under git merge --ff-only, that often turns out to be what I want: do that fast-forward, or else stop and let me look around and decide whether I want a rebase, a merge, or whatever else I might want.[1] And that often turns out to be what others want as well, and now git pull, run with no arguments at all, can be told do that directly:

```
git config --global pull.ff only
```

achieves this.

Meanwhile, the other two git config --global commands in the hint you show in your question make the second command be merge or rebase. So *now*, in 2022, it's easy to tell git pull to do what *I* would want it to do. Furthermore, it seems that the Git maintainers have come around to my point of view: that git pull *without* some forethought is *bad*, and newbies should not use it. So they've set up git pull so that it now *requires* that you pick one of these three options, if you want to run it with no arguments.[2]

So, you need to pick one. **The *old* default *was* git config pull.rebase false, but that was a bad default. I do not recommend it. I *do* recommend git config pull.ff only** (though I still don't actually use it due to 15+ years of habits).

[1]One real-world example: I encounter some bug that's a problem for me. I make a change to the code that I know is wrong, but lets *me* get *my* work done. I commit this horrible hack. I then wait for the upstream to make changes. They do and I bring in the new commits. If they've *fixed* the bug, I want to *drop* my fix, not merge or rebase it. If they *have not* fixed the bug, I want to rebase my hack (which may or may not need some tweaking). The "have they fixed the bug" test requires something git pull cannot test on its own.

[2]Note that running git pull *with* arguments is not supposed to generate this kind of complaint. I still don't run it much, so I'm not quite sure what the bug was, but in the first round or two of implementation of the new feature, there was a bug where git pull would complain inappropriately. I believe it is fixed in 2.35 and am almost positive it's fixed in 2.36, which should be out any time now.