git$_{well}$*Documentation*

Release 0.2.1

Jon Crall

Feb 23, 2024

CONTENTS

Basic

CHAPTER

ONE

GIT_WELL PACKAGE

1.1 Subpackages

1.1.1 git_well.demo package

1.1.1.1 Module contents

git_well.demo.**make_dummy_git_repo**()

git_well.demo.**make_dummy_git_repo_with_orphans**()

1.2 Submodules

1.2.1 git_well.__main__ module

1.2.2 git_well._utils module

git_well._utils.**rich_print**(**args*, ***kwargs*)

git_well._utils.**find_merged_branches**(*repo*, *main_branch='main'*)

git_well._utils.**confirm**(*msg*)

git_well._utils.**choice_prompt**(*msg*, *choices*)

git_well._utils.**find_git_root**(*dpath*)

1.2.3 git_well.git_autoconf_gpgsign module

class git_well.git_autoconf_gpgsign.**GitAutoconfGpgsignCLI**(**args*, ***kwargs*)

Bases: DataConfig

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> # xdoctest: +SKIP
>>> from git_well.git_autoconf_gpgsign import *  # NOQA
>>> cmdline = 0
>>> kwargs = dict()
>>> cls = GitAutoconfGpgsignCLI
>>> cls.main(cmdline=cmdline, **kwargs)
```

default = {'remote': <Value(None)>, 'repo_dpath': <Value('.')>}

git_well.git_autoconf_gpgsign.**lookup_gpg_keyinfos**(*identifier*, *verbose=0*, *capabilities=None*, *allow_subkey=True*, *allow_mainkey=True*, *full=True*, *filter_expired=True*, *mintrust=None*)

python ~/local/scripts/xgpg.py lookup_keyid "Emmy" python ~/local/scripts/xgpg.py lookup_keyid "Crall" –allow_mainkey=False –capabilities=sign python ~/local/scripts/xgpg.py lookup_keyid "Crall" –allow_mainkey=False –capabilities=encrypt python ~/local/scripts/xgpg.py lookup_keyid "Crall" –allow_mainkey=False –capabilities=auth python ~/local/scripts/xgpg.py lookup_keyid "Jonathan Crall"

git_well.git_autoconf_gpgsign.**gpg_entries**(*identifier=None*, *verbose=0*)

References

Format of the colon listings https://github.com/gpg/gnupg/blob/master/doc/DETAILS

git_well.git_autoconf_gpgsign.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> # xdoctest: +SKIP
>>> from git_well.git_autoconf_gpgsign import *  # NOQA
>>> cmdline = 0
>>> kwargs = dict()
>>> cls = GitAutoconfGpgsignCLI
>>> cls.main(cmdline=cmdline, **kwargs)
```

1.2.4 git_well.git_branch_cleanup module

class git_well.git_branch_cleanup.**CleanDevBranchConfig**(**args*, ***kwargs*)

Bases: DataConfig

Cleanup branches that have been merged into main.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_branch_cleanup import CleanDevBranchConfig
>>> from git_well.repo import Repo
>>> cls = CleanDevBranchConfig
>>> repo = Repo.demo()
>>> # TODO: add commits so they aren't all considerd the same branch
>>> repo.cmd('git checkout -b dev/1.0.0')
>>> repo.cmd('git checkout -b dev/2.1.0')
>>> repo.cmd('git checkout main')
>>> assert repo.active_branch.name == 'main'
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> kwargs['yes'] = True
>>> cls.main(cmdline=cmdline, **kwargs)
```

default = {'keep_last': <Value(1)>, 'remove_merged': <Value(False)>, 'repo_dpath': <Value('.')>, 'yes': <Value(False)>}

git_well.git_branch_cleanup.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_branch_cleanup import CleanDevBranchConfig
>>> from git_well.repo import Repo
>>> cls = CleanDevBranchConfig
>>> repo = Repo.demo()
>>> # TODO: add commits so they aren't all considerd the same branch
>>> repo.cmd('git checkout -b dev/1.0.0')
>>> repo.cmd('git checkout -b dev/2.1.0')
>>> repo.cmd('git checkout main')
>>> assert repo.active_branch.name == 'main'
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> kwargs['yes'] = True
>>> cls.main(cmdline=cmdline, **kwargs)
```

1.2.5 git_well.git_branch_upgrade module

A git tool for handling the dev/<version> branch patterns

See the GitDevbranchConfig for functionality

Notes

to remove branches from remotes

f"git push origin –delete {branch_name}"

to see results:

git fetch –prune

Requires:
scriptconfig git-python packaging ubelt

class git_well.git_branch_upgrade.**UpdateDevBranch**(**args*, ***kwargs*)

Bases: DataConfig

Upgrade to the latest "dev" branch. I.e. search for the branch dev/<version> with the greatest semantic version.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_branch_upgrade import UpdateDevBranch
>>> from git_well.repo import Repo
>>> cls = UpdateDevBranch
>>> repo = Repo.demo()
>>> repo.cmd('git checkout -b dev/1.0.0')
>>> repo.cmd('git checkout -b dev/2.1.0')
>>> repo.cmd('git checkout main')
>>> assert repo.active_branch.name == 'main'
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> cls.main(cmdline=cmdline, **kwargs)
>>> assert repo.active_branch.name == 'dev/2.1.0'
```

default = {'repo_dpath': <Value('.')>}

git_well.git_branch_upgrade.**dev_branches**(*repo*)

git_well.git_branch_upgrade.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_branch_upgrade import UpdateDevBranch
>>> from git_well.repo import Repo
>>> cls = UpdateDevBranch
>>> repo = Repo.demo()
>>> repo.cmd('git checkout -b dev/1.0.0')
>>> repo.cmd('git checkout -b dev/2.1.0')
>>> repo.cmd('git checkout main')
>>> assert repo.active_branch.name == 'main'
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> cls.main(cmdline=cmdline, **kwargs)
>>> assert repo.active_branch.name == 'dev/2.1.0'
```

1.2.6 git_well.git_discover_remote module

class git_well.git_discover_remote.**GitDiscoverRemoteCLI**(**args*, ***kwargs*)

Bases: DataConfig

Attempt to discover a ssh remote based on an ssh host.

Like git-sync, the remote machine must have the same directory structure relative to the home drive.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_discover_remote import GitDiscoverRemoteCLI
>>> from git_well.repo import Repo
>>> cls = GitDiscoverRemoteCLI
>>> repo = Repo.demo()
>>> # TODO: make a plausible scenario
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> import pytest
>>> with pytest.raises(Exception):
>>>     cls.main(cmdline=cmdline, **kwargs)
```

default = {'forward_ssh_agent': <Value(False)>, 'home': <Value(None)>, 'host': <Value(None)>, 'remote': <Value(None)>, 'remote_cwd': <Value(None)>, 'repo_dpath': <Value('.')>, 'test_remote': <Value(True)>}

git_well.git_discover_remote.**fsspec_shh_connect**(*host*)

git_well.git_discover_remote.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_discover_remote import GitDiscoverRemoteCLI
>>> from git_well.repo import Repo
>>> cls = GitDiscoverRemoteCLI
>>> repo = Repo.demo()
>>> # TODO: make a plausible scenario
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> import pytest
>>> with pytest.raises(Exception):
>>>     cls.main(cmdline=cmdline, **kwargs)
```

1.2.7 git_well.git_rebase_add_continue module

class git_well.git_rebase_add_continue.**GitRebaseAddContinue**(**args*, ***kwargs*)

Bases: DataConfig

A single step to make rebasing easier.

Usually a rebase has the user explicitly add and then continue. This script checks all of the paths for conflicts and then if none exist adds all files and continues.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_rebase_add_continue import GitRebaseAddContinue
>>> from git_well.repo import Repo
>>> cls = GitRebaseAddContinue
>>> repo = Repo.demo()
>>> # TODO: make a plausible scenario
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> import pytest
>>> with pytest.raises(RuntimeError):
>>>     cls.main(cmdline=cmdline, **kwargs)
```

default = {'repo_dpath': <Value('.')>, 'skip_editor': <Value(True)>}

git_well.git_rebase_add_continue.**parsed_rebase_git_status**(*repo_dpath*)

a git status output has several possible sections it can output, check for those, and set the state based on them. Information within each state will be indented

git_well.git_rebase_add_continue.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_rebase_add_continue import GitRebaseAddContinue
>>> from git_well.repo import Repo
>>> cls = GitRebaseAddContinue
>>> repo = Repo.demo()
>>> # TODO: make a plausible scenario
>>> cmdline = 0
>>> kwargs = dict()
>>> kwargs['repo_dpath'] = repo
>>> import pytest
>>> with pytest.raises(RuntimeError):
>>>     cls.main(cmdline=cmdline, **kwargs)
```

1.2.8 git_well.git_remote_protocol module

class git_well.git_remote_protocol.**GitRemoteProtocol**(**args*, ***kwargs*)

Bases: DataConfig

Change the protocol for all remotes that match a specific user / group.

The new protocol can be git or https.

An alias for this command is `git permit` because it "permits" a specific group to use ssh permissions.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

default = {'group': <Value('special:auto')>, 'protocol': <Value('ssh')>, 'repo_dpath': <Value('.')>}

main(***kwargs*)

Example

```
>>> from git_well.git_remote_protocol import GitRemoteProtocol
>>> from git_well.repo import Repo
>>> repo = Repo.demo()
>>> repo.cmd('git remote add origin https://github.com/Foobar/foobar.git')
>>> cmdline = 0
>>> GitRemoteProtocol.main(cmdline=cmdline, repo_dpath=repo, protocol='git')
>>> assert len(repo.remotes) == 1
```

(continues on next page)

(continued from previous page)

```
>>> assert list(repo.remotes[0].urls)[0] == 'git@github.com:Foobar/foobar.git'
>>> GitRemoteProtocol.main(cmdline=cmdline, repo_dpath=repo, protocol='https')
>>> assert list(repo.remotes[0].urls)[0] == 'https://github.com/Foobar/foobar.
↪git'
>>> GitRemoteProtocol.main(cmdline=cmdline, repo_dpath=repo, protocol='git')
>>> assert list(repo.remotes[0].urls)[0] == 'git@github.com:Foobar/foobar.git'
```

class git_well.git_remote_protocol.**GitURL**(*data*)

Bases: str

Represents a url to a git repo and can parse info about / modify the protocol

References

https://git-scm.com/docs/git-clone#_git_urls

CommandLine

```
xdoctest -m git_well.git_remote_protocol GitURL
```

Example

```
>>> from git_well.git_remote_protocol import *  # NOQA
>>> urls = [
>>>     GitURL('https://foo.bar/user/repo.git'),
>>>     GitURL('ssh://foo.bar/user/repo.git'),
>>>     GitURL('ssh://git@foo.bar/user/repo.git'),
>>>     GitURL('git@foo.bar:group/repo.git'),
>>>     GitURL('host:path/to/my/repo/.git'),
>>> ]
>>> for url in urls:
>>>     print('---')
>>>     print(f'url = {url}')
>>>     print(ub.urepr(url.info))
>>>     print('As git   : ' + url.to_git())
>>>     print('As ssh   : ' + url.to_ssh())
>>>     print('As https : ' + url.to_https())
```

_parse()

property info

to_git()

to_ssh()

to_https()

git_well.git_remote_protocol.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_remote_protocol import GitRemoteProtocol
>>> from git_well.repo import Repo
>>> repo = Repo.demo()
>>> repo.cmd('git remote add origin https://github.com/Foobar/foobar.git')
>>> cmdline = 0
>>> GitRemoteProtocol.main(cmdline=cmdline, repo_dpath=repo, protocol='git')
>>> assert len(repo.remotes) == 1
>>> assert list(repo.remotes[0].urls)[0] == 'git@github.com:Foobar/foobar.git'
>>> GitRemoteProtocol.main(cmdline=cmdline, repo_dpath=repo, protocol='https')
>>> assert list(repo.remotes[0].urls)[0] == 'https://github.com/Foobar/foobar.git'
>>> GitRemoteProtocol.main(cmdline=cmdline, repo_dpath=repo, protocol='git')
>>> assert list(repo.remotes[0].urls)[0] == 'git@github.com:Foobar/foobar.git'
```

1.2.9 git_well.git_squash_streaks module

This git-squash-streaks command

Requirements:

pip install ubelt pip install GitPython pip install scriptconfig

class git_well.git_squash_streaks.**SquashStreakCLI**(**args*, ***kwargs*)

Bases: DataConfig

Squashes consecutive commits that meet a specified criteiron.

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

default = {'authors': <Value(None)>, 'auto_rollback': <Value(False)>, 'custom_streak': <Value(None)>, 'dry': <Value(True)>, 'force': <Value(None)>, 'inplace': <Value(False)>, 'oldest_commit': <Value(None)>, 'pattern': <Value(None)>, 'preserve_tags': <Value(True)>, 'tags': <Value(False)>, 'timedelta': <Value('sameday')>, 'verbose': <Value(True)>}

main(***kwargs*)

git-squash-streaks

Usage:

See argparse

normalize()

git_well.git_squash_streaks.**print_exc**(*exc_info=None*)

Example

```
>>> try:
>>>     raise Exception('foobar')
>>> except Exception as ex:
>>>     import sys
>>>     exc_info = sys.exc_info()
>>>     print_exc(exc_info)
```

class git_well.git_squash_streaks.**Streak**(*child*, *_streak=None*)

Bases: NiceRepr

append(*commit*)

property before_start

property after_stop

property start

property stop

git_well.git_squash_streaks.**find_pseudo_chain**(*head*, *oldest_commit=None*, *preserve_tags=True*)

Finds start and end points that can be safely squashed between

CommandLine

```
xdoctest -m git_well.git_squash_streaks find_pseudo_chain
```

Example

```
>>> # xdoctest: +REQUIRES(LINUX)
>>> from git_well.git_squash_streaks import *  # NOQA
>>> import git
>>> from git_well import demo
>>> repo_dpath = demo.make_dummy_git_repo()
>>> repo = git.Repo(repo_dpath)
>>> head = repo.commit('HEAD')
>>> pseudo_chain = find_pseudo_chain(head)
>>> print('pseudo_chain = {}'.format(ub.urepr(pseudo_chain, nl=1)))
```

git_well.git_squash_streaks.**git_nx_graph**(*head*, *oldest_commit=None*, *preserve_tags=False*)

Example

```
>>> # xdoctest: +REQUIRES(LINUX)
>>> from git_well.git_squash_streaks import *  # NOQA
>>> from git_well import demo
>>> import git
>>> repo_dpath = demo.make_dummy_git_repo()
>>> repo = git.Repo(repo_dpath)
>>> head = repo.commit('HEAD')
>>> oldest_commit = 'master'
>>> oldest_commit = None
>>> graph = git_nx_graph(head, oldest_commit)
```

git_well.git_squash_streaks.**find_chain**(*head*, *authors=None*, *preserve_tags=True*, *oldest_commit=None*)

Find a chain of commits starting at the HEAD. If *authors* is specified the commits must be from one of these authors.

The term chain is used in the graph-theory sense. It is a list of commits where all non-endpoint commits have exactly one parent and one child.

Todo:

- **[ ] allow a chain to include branches if all messages on all branches**
  conform to the chain pattern (e.g. wip)

def search(node, current_path):

if current_path:
pass

child_paths = [] for parent in node.parents:

path = search(parent, current_path) child_paths.append(path)

if len(child_paths) == 0:
pass

if len(child_paths) == 1:
normal one parent case pass

else:
pass # Branching case # ACCEPT THE BRANCHING PATHS IF: # * PARENT OF ALL PATHS HAVE A COMMON ENDPOINT # * HANDLE CASE WHERE PATHS OVERLAPS

Parameters

- **head** (*Commit*) – starting point
- **authors** (*set*) – valid authors
- **preserve_tags** (*bool*) – if True the chain is not allowed to extend past any tags. If a set, then we will not procede past any tag with a name in the set. Defaults to True

Example

```
>>> # xdoctest: +REQUIRES(LINUX)
>>> # assuming you are in a git repo
>>> import git
>>> from git_well.git_squash_streaks import *  # NOQA
>>> from git_well.git_squash_streaks import _squash_between
>>> from git_well import demo
>>> repo_dpath = demo.make_dummy_git_repo()
>>> repo = git.Repo(repo_dpath)
>>> chain = find_chain(repo.head.commit)
```

git_well.git_squash_streaks.**find_streaks**(*chain*, *authors=None*, *timedelta='sameday'*, *pattern=None*)

Given a chain, finds subchains (called streaks) that have the same author and are within a timedelta threshold of each other.

Parameters

- **chain** (*List[Commit]*) – from *find_chain*
- **authors** (*set*) – valid authors
- **timedelta** (*float | str*) – minimum time between commits in seconds or a categorical value such as 'sameday' or 'alltime'
- **pattern** (*str*) – instead of squashing messages with the same name, squash only if they match this pattern Defaults to None, None means the consecutive messages should match.

git_well.git_squash_streaks.**checkout_temporary_branch**(*repo*, *suffix='-temp-script-branch'*)

Changes to a temporary branch so we don't messup anything on the main one.

If the temporary branch exists, it is deleted, so make sure you choose your suffix so that it never conflicts with any real branches.

git_well.git_squash_streaks.**commits_between**(*repo*, *start*, *stop*)

Parameters

- **start** (*Commit*) – toplogically first (i.e. chronologically older) commit
- **stop** (*Commit*) – toplogically last (i.e. chronologically newer) commit

Returns
between commits

Return type
List[Commit]

References

https://stackoverflow.com/questions/18679870/commits-between-2-hashes https://stackoverflow.com/questions/462974/diff-double-and-triple-dot

Warning: this gets messy any node on the path between <start> and <stop> has more than one parent that is not on a path between <start> and <stop>

Notes

As a prefix: the carrot (^A) removes commits reachable from A. As a suffix: the carrot (A^) references the 1st parent of A Furthermore:

(A^n) references the n-th parent of A (A~n) references the n-th ancestor of A The tilde and carrot can be chained. A^^ = A~2 = the grandparent of A

Reachable means everything in the past.

PAST.....PRESENT <p1> – <start> – <A> – <B> – <stop>

/

<p2> __/

Example

```
>>> # xdoctest: +REQUIRES(LINUX)
>>> from git_well.git_squash_streaks import *  # NOQA
>>> from git_well import demo
>>> import git
>>> repo_dpath = demo.make_dummy_git_repo()
>>> repo = git.Repo(repo_dpath)
>>> stop = repo.head.commit
>>> start = stop.parents[0].parents[0].parents[0].parents[0]
>>> commits = commits_between(repo, start, stop)
>>> assert commits[0] == stop
>>> assert commits[-1] == start
>>> assert len(commits) == 5
```

exception git_well.git_squash_streaks.**RollbackError**

Bases: Exception

git_well.git_squash_streaks.**_squash_between**(*repo*, *start*, *stop*, *dry=False*, *verbose=True*)

inplace squash between, use external function that sets up temp branches to use this directly from the commandline.

git_well.git_squash_streaks.**do_tags**(*verbose=True*, *inplace=False*, *dry=True*, *auto_rollback=False*)

git_well.git_squash_streaks.**squash_streaks**(*authors*, *timedelta='sameday'*, *pattern=None*, *inplace=False*, *auto_rollback=True*, *dry=False*, *verbose=True*, *custom_streak=None*, *preserve_tags=True*, *oldest_commit=None*)

Squashes consecutive commits that meet a specified criteiron.

Parameters

- **authors** (*set*) – "level-set" of authors who's commits can be squashed together.
- **timedelta** (*str or int*) – strategy mode or max number of seconds to determine how far apart two commits can be before they are squashed. (Default: 'sameday'). Valid values: ['sameday', 'alltime', 'none', <n_seconds:float>]
- **pattern** (*str*) – instead of squashing messages with the same name, squash only if they match this pattern (Default: None). Default of None means that squash two commits if they have the same message.

- **inplace** (*bool*) – if True changes will be applied directly to the current branch otherwise a temporary branch will be created. Then you must manually reset the current branch to this branch and delete the temp branch. (Default: False)
- **auto_rollback** (*bool*) – if True the repo will be reset to a clean state if any errors occur. (Default: True)
- **dry** (*bool*) – if True this only executes a dry run, that prints the chains that would be squashed (Default: False)
- **verbose** (*bool, default=True*) – verbosity flag
- **custom_streak** (*tuple*) – hack, specify two commits to explicitly squash only this streak is used. We do not automatically check for others.
- **preserve_tags** (*bool, default=True*) – if True the chain is not allowed to extend past any tags. If a set, then we will not procede past any tag with a name in the set.
- **oldest_commit** (*str | None*) – if specified we will only squash commits toplogically after this commit in the graph.

`git_well.git_squash_streaks.`**`git_squash_streaks`**(*cmdline=1*, ***kwargs*)

git-squash-streaks

Usage:

See argparse

`git_well.git_squash_streaks.`**`main`**(*cmdline=1*, ***kwargs*)

git-squash-streaks

Usage:

See argparse

1.2.10 git_well.git_stats module

SeeAlso:

https://github.com/erikbern/git-of-theseus https://stackoverflow.com/questions/42715785/how-do-i-show-statistics-for-authors-contributions-in-git

`class git_well.git_stats.`**`GitStatsCLI`**(**args*, ***kwargs*)

Bases: `DataConfig`

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

`classmethod` **`main`**(*cmdline=1*, ***kwargs*)

Example

```
>>> # xdoctest: +SKIP
>>> from git_well.git_stats import *  # NOQA
>>> cmdline = 0
>>> kwargs = dict()
>>> cls = GitStatsCLI
>>> cls.main(cmdline=cmdline, **kwargs)
```

default = {'repo_dpath': <Value('.')>}

git_well.git_stats.**commit_stats**(*repo*)

git_well.git_stats.**author_stats**(*repo*)

git_well.git_stats.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> # xdoctest: +SKIP
>>> from git_well.git_stats import *  # NOQA
>>> cmdline = 0
>>> kwargs = dict()
>>> cls = GitStatsCLI
>>> cls.main(cmdline=cmdline, **kwargs)
```

1.2.11 git_well.git_sync module

class git_well.git_sync.**GitSyncCLI**(**args*, ***kwargs*)

Bases: DataConfig

Sync a git repo with a remote server via ssh

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

default = {'dry': <Value(False)>, 'force': <Value(False)>, 'forward_ssh_agent': <Value(False)>, 'host': <Value(None)>, 'message': <Value('wip [skip ci]')>, 'remote': <Value(None)>}

main(***kwargs*)

git_well.git_sync.**main**(*cmdline=True*, ***kwargs*)

git_well.git_sync.**getcwd**()

Workaround to get the working directory without dereferencing symlinks. This may not work on all systems.

References

https://stackoverflow.com/questions/1542803/getcwd-dereference-symlinks

git_well.git_sync.**git_default_push_remote_name**()

git_well.git_sync.**_devcheck**()

TODO: need to resolve the receive.denyCurrentBranch problem less manually

remote: error: refusing to update checked out branch: refs/heads/updates remote: error: By default, updating the current branch in a non-bare repository remote: is denied, because it will make the index and work tree inconsistent remote: with what you pushed, and will require 'git reset –hard' to match remote: the work tree to HEAD.

On the remote:

git config –local receive.denyCurrentBranch warn

git_well.git_sync.**git_sync**(*host*, *remote=None*, *message='wip [skip ci]'*, *forward_ssh_agent=False*, *dry=False*, *force=False*, *home=None*)

Commit any changes in the current working directory, ssh into a remote machine, and then pull those changes.

Parameters

- **host** (*str*) – The name of the host to sync to: e.g. user@remote.com
- **remote** (*str*) – The git remote used to push and pull from
- **message** (*str, default='wip [skip ci]'*) – Default git commit message.
- **forward_ssh_agent** (*bool*) – Enable forwarding of the ssh authentication agent connection
- **force** (*bool, default=False*) – if True does a forced push and additionally forces the remote to do a hard reset to the remote state.
- **dry** (*bool, default=False*) – Executes dry run mode.
- **home** (*str | PathLike | None*) – if specified, overwrite where git-sync thinks the home location is

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> host = 'user@remote.com'
>>> remote = 'origin'
>>> message = 'this is the commit message'
>>> home = getcwd()  # pretend the home is here for the test
>>> git_sync(host, remote, message, dry=True, home=home)
git commit -am "this is the commit message"
git push origin
ssh user@remote.com "cd ... && git pull origin ..."
```

1.2.12 git_well.git_track_upstream module

Requirements:
pip install GitPython

class git_well.git_track_upstream.**TrackUpstreamCLI**(**args*, ***kwargs*)

Bases: DataConfig

Set the branch upstream with sensible defaults if possible.

This script can auto-choose sensible default if there is only one remote that also has the working branch. When there is an ambiguity the user will be asked to choose from a list of available remotes with this branch.

Once the remote is found the script executes:

..code:: bash

git branch –set-upstream-to=<remote>/<branch> <branch>

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod main(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_track_upstream import TrackUpstreamCLI
>>> from git_well.repo import Repo
>>> repo = Repo.demo()
>>> repo.cmd('git remote add origin https://github.com/Erotemic/git_well.git')
>>> # TODO: make this test work without the network
>>> repo.cmd('git fetch origin')
>>> repo.cmd('git reset --hard origin/main')
>>> cmdline = 0
>>> cls = TrackUpstreamCLI
>>> kwargs = cls()
>>> kwargs['repo_dpath'] = repo
>>> cls.main(cmdline=cmdline, **kwargs)
```

default = {'force': <Value(False)>, 'repo_dpath': <Value('.')>}

git_well.git_track_upstream.**unique_remotes_with_branch**(*repo*, *branch*)

git_well.git_track_upstream.**main**(*cmdline=1*, ***kwargs*)

Example

```
>>> from git_well.git_track_upstream import TrackUpstreamCLI
>>> from git_well.repo import Repo
>>> repo = Repo.demo()
>>> repo.cmd('git remote add origin https://github.com/Erotemic/git_well.git')
>>> # TODO: make this test work without the network
>>> repo.cmd('git fetch origin')
>>> repo.cmd('git reset --hard origin/main')
>>> cmdline = 0
>>> cls = TrackUpstreamCLI
>>> kwargs = cls()
>>> kwargs['repo_dpath'] = repo
>>> cls.main(cmdline=cmdline, **kwargs)
```

1.2.13 git_well.main module

class git_well.main.**GitWellModalCLI**(*description=''*, *sub_clis=None*, *version=None*)

Bases: ModalCLI

squash_streaks

alias of *SquashStreakCLI*

branch_upgrade

alias of *UpdateDevBranch*

sync

alias of *GitSyncCLI*

branch_cleanup

alias of *CleanDevBranchConfig*

track_upstream

alias of *TrackUpstreamCLI*

rebase_add_continue

alias of *GitRebaseAddContinue*

remote_protocol

alias of *GitRemoteProtocol*

discover_remote

alias of *GitDiscoverRemoteCLI*

autoconf_gpgsign

alias of *GitAutoconfGpgsignCLI*

git_well.main.**main**()

1.2.14 git_well.repo module

class git_well.repo.**Repo**(*path: str | ~os.PathLike[str] | None = None*, *odbt: ~typing.Type[~gitdb.db.loose.LooseObjectDB] = <class 'git.db.GitCmdObjectDB'>*, *search_parent_directories: bool = False*, *expand_vars: bool = True*)

Bases: Repo

Create a new Repo instance.

Parameters

- **path** – The path to either the root git directory or the bare git repo:

```
repo = Repo("/Users/mtrier/Development/git-python")
repo = Repo("/Users/mtrier/Development/git-python.git")
repo = Repo("~/Development/git-python.git")
repo = Repo("$REPOSITORIES/Development/git-python.git")
repo = Repo("C:\Users\mtrier\Development\git-python\.git")
```

  - In *Cygwin*, path may be a `cygdrive/...` prefixed path.
  - If it evaluates to false, `GIT_DIR` is used, and if this also evals to false, the current-directory is used.

- **odbt** – Object DataBase type - a type which is constructed by providing the directory containing the database objects, i.e. .git/objects. It will be used to access all object data
- **search_parent_directories** – If True, all parent directories will be searched for a valid repo as well.

  Please note that this was the default behaviour in older versions of GitPython, which is considered a bug though.

Raises

- **InvalidGitRepositoryError** –
- **NoSuchPathError** –

Returns

git.Repo

cmd(*command*, ***kwargs*)

Execute a command in the root of the repo.

property dpath

Alias of working_dir wraped in a ubelt Path

property is_submodule

True if the submodule for another repo.

property config_fpath

classmethod coerce(*data*)

Try to construct a Repo object from input dat

Parameters

data (*str | PathLike | Repo*) – If a Repo object, data is returned as-is. If a path inside a git repo, return a *Repo* object that references the repo root.

Returns
Repo

classmethod **demo**()

Create a demo repo for tests

Returns
Repo

Example

```
>>> from git_well.repo import Repo
>>> self = Repo.demo()
```

find_merged_branches(*main_branch='main'*)

1.3 Module contents

Basic

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U