Write yourself a Git!

1. Introduction

This article is an attempt at explaining the Git version control system from the bottom up, that is, starting at the most fundamental level moving up from there. This does not sound too easy, and has been attempted multiple times with questionable success. But there's an easy way: all it takes to understand Git internals is to reimplement Git from scratch.

No, don't run.

It's not a joke, and it's really not complicated: if you read this article top to bottom and write the code (or just clone the repository — but you should write the code yourself, really), you'll end up with a program, called `wyag`, that will implement all the fundamental features of git: `init`, `add`, `rm`, `status`, `commit`, `log`… in a way that is perfectly compatible with `git` itself. The last commit of this article was actually created with `wyag`, not `git`. And all that in exactly 563 lines of very simple Python code.

But isn't Git too complex for that? That Git is complex is, in my opinion, a misconception. Git is a large program, with a lot of features, that's true. But the core of that program is actually extremely simple, and its apparent complexity stems first from the fact it's often deeply counterintuitive (and Git is a burrito blog posts probably don't help). But maybe what makes Git the most confusing is the extreme

simplicity *and* power of its core model. The combination of core simplicity and powerful applications often makes thing really hard to grasp, because of the mental jump required to derive the variety of applications from the essential simplicity of the fundamental abstraction (monads, anyone?)

Implementing Git will expose its fundamentals in all their naked glory.

What to expect? This article will implement and explain in great details (if something is not clear, please report it!) a very simplified version of Git core commands. I will keep the code simple and to the point, so `wyag` won't come anywhere near the power of the real git command-line — but what's missing will be obvious, and trivial to implement by anyone who wants to give it a try. "Upgrading wyag to a full-featured git library and CLI is left as an exercise to the reader", as they say.

More precisely, we'll implement:

- `add` () git man page
- `cat-file` (wyag source) git man page
- `checkout` (wyag source) git man page
- `commit` () git man page
- `hash-object` (wyag source) git man page
- `init` (wyag source) git man page
- `log` (wyag source) git man page
- `ls-files` (wyag source) git man page
- `ls-tree` () git man page
- `merge` () git man page
- `rebase` () git man page

- `rev-parse` () git man page
- `rm` () git man page
- `show-ref` () git man page
- `tag` (wyag source) git man page

You're not going to need to know much to follow this article: just some basic Git (obviously), some basic Python, some basic shell.

- First, I'm only going to assume some level of familiarity with the most basic **git commands** — nothing like an expert level, but if you've never used `init`, `add`, `rm`, `commit` or `checkout`, you will be lost.
- Language-wise, wyag will be implemented in **Python**. Again, I won't use anything too fancy, and Python looks like pseudo-code anyways, so it will be easy to follow (ironically, the most complicated part will be the command-line arguments parsing logic, and you don't really need to understand that). Yet, if you know programming but have never done any Python, I suggest you find a crash course somewhere in the internet just to get acquainted with the language.
- `wyag` and `git` are terminal programs. I assume you know your way inside a Unix terminal. Again, you don't need to be a l77t h4x0r, but `cd`, `ls`, `rm`, `tree` and their friends should be in your toolbox.

Warning

Note for Windows users

`wyag` should run on any Unix-like system with a Python interpreter, but I have absolutely no idea how it will behave on Windows. The test suite absolutely requires a bash-compatible shell, which I assume the WSL can provide. Also, if you are using WSL, make sure your `wyag` file uses Unix-style line endings (See this StackOverflow solution if you use VS Code). Feedback from Windows users would be appreciated!

Note

Acknowledgments
This article benefited from significant contributions from multiple people, and I'm grateful to them all. Special thanks to:

- Github user tammoippen, who first drafted the `tag_create` function I had simply… forgotten to write (that was #9).
- Github user hjlarry fixed multiple issues in #22.
- GitHub user cutebbb implemented the first version of ls-files in #27, and by doing so finally brought wyag to the wonders of the staging area!

2. Getting started

You're going to need Python 3 (I used 3.6.5: if you encounter issues try using at least this version. Python 2 won't work at all) and your favorite text editor. We won't need third party packages or virtualenvs, or anything besides a regular Python interpreter: everything we need is in Python's standard library.

We'll split the code into two files:

- An executable, called `wyag`;
- A Python library, called `libwyag.py`;

Now, every software project starts with a boatload of boilerplate, so let's get this over with.

We'll begin by creating the binary. Create a new file called `wyag` in your text editor, and copy the following few lines:

```
#!/usr/bin/env python3

import libwyag
libwyag.main()
```

Then make it executable:

```
$ chmod +x wyag
```

You're done!

Now for the library. it must be called `libwyag.py`, and be in the same directory as the `wyag` executable. Begin by opening the empty `libwyag.py` in your text editor.

We're first going to need a bunch of imports (just copy each import, or merge them all in a single line)

- Git is a CLI application, so we'll need something to parse command-line arguments. Python provides a cool module called argparse that can do 99% of the job for us.

```
import argparse
```

- We'll need a few more container types than the base lib provides, most notably an `OrderedDict`. It's in collections.

```
import collections
```

- Git uses a configuration file format that is basically Microsoft's INI format. The configparser module can read and write these files.

```
import configparser
```

- Git uses the SHA-1 function quite extensively. In Python, it's in hashlib.

```
import hashlib
```

- Just one function from math:

```
from math import ceil
```

- os and os.path provide some nice filesystem abstraction routines.

```
import os
```

- we use *just a bit* of regular expressions:

```
import re
```

- We also need sys to access the actual command-line arguments (in `sys.argv`):

```
import sys
```

- Git compresses everything using zlib. Python has that, too:

```
import zlib
```

Imports are done. We'll be working with command-line arguments a lot. Python provides a simple yet reasonably powerful parsing library, `argparse`. It's a nice library, but its interface may not be the most intuitive ever; if need, refer to its documentation.

```
argparser = argparse.ArgumentParser(description="The stupidest content tracker")
```

We'll need to handle subcommands (as in git: `init`, `commit`, etc.) In argparse slang, these are called "subparsers". At this point we only need to declare that our CLI will use some, and that all 调用 will actually *require* one — you don't just call `git`, you call `git COMMAND`.

```
argsubparsers = argparser.add_subparsers(title="Commands", dest="command")
argsubparsers.required = True
```

The `dest="command"` argument states that the name of the chosen subparser will be returned as a string in a field called `command`. So we just need to read this string and call the correct function accordingly. By convention, I'll prefix these functions by `cmd_`. `cmd_*` functions take the parsed arguments as their unique parameter, and are responsible for processing and validating them before executing the actual command.

```
def main(argv=sys.argv[1:]):
    args = argparser.parse_args(argv)

    if   args.command == "add"         : cmd_add(args)
    elif args.command == "cat-file"    : cmd_cat_file(args)
    elif args.command == "checkout"    : cmd_checkout(args)
    elif args.command == "commit"      : cmd_commit(args)
    elif args.command == "hash-object" : cmd_hash_object(args)
    elif args.command == "init"        : cmd_init(args)
    elif args.command == "log"         : cmd_log(args)
    elif args.command == "ls-files"    : cmd_ls_files(args)
    elif args.command == "ls-tree"     : cmd_ls_tree(args)
    elif args.command == "merge"       : cmd_merge(args)
    elif args.command == "rebase"      : cmd_rebase(args)
    elif args.command == "rev-parse"   : cmd_rev_parse(args)
    elif args.command == "rm"          : cmd_rm(args)
    elif args.command == "show-ref"    : cmd_show_ref(args)
    elif args.command == "tag"         : cmd_tag(args)
```

3. Creating repositories: init

Obviously, the first Git command in chronological *and* logical order is `git init`, so we'll begin by creating `wyag init`. To achieve this, we're going to first need some very basic repository abstraction.

3.1. The Repository object

We'll obviously need some abstraction for a repository: almost every time we run a git command, we're trying to do something to a repository, to create it, read from it or modify it.

A repository, in git, is made of two things: a "work tree", where the files meant to be in version control live, and a "git directory", where Git stores its own data. In most cases, the worktree is a regular directory and the git directory is a child directory of the worktree, called `.git`.

Git supports *much more* cases (bare repo, separated gitdir, etc) but we won't need them: we'll stick the basic approach of `worktree/.git`. Our repository object will then just hold two paths: the worktree and the gitdir.

To create a new `Repository` object, we only need to make a few checks:

- We must verify that the directory exists, and contains a subdirectory called `.git`.
- We read its configuration in `.git/config` (it's just an INI file) and control that `core.repositoryformatversion` is 0. More on that field in a moment.

The constructor takes an optional `force` which disables all check. That's because the `repo_create()` function which we'll create later uses a `Repository` object to *create* the repo. So we need a way to create repository even from (still) invalid filesystem locations.

```
class GitRepository (object):
    """A git repository"""

    worktree = None
    gitdir = None
    conf = None

    def __init__(self, path, force=False):
        self.worktree = path
        self.gitdir = os.path.join(path, ".git")

        if not (force or os.path.isdir(self.gitdir)):
            raise Exception("Not a Git repository %s" % path)

        # Read configuration file in .git/config
        self.conf = configparser.ConfigParser()
        cf = repo_file(self, "config")

        if cf and os.path.exists(cf):
            self.conf.read([cf])
        elif not force:
            raise Exception("Configuration file missing")

        if not force:
            vers = int(self.conf.get("core", "repositoryformatversion"))
            if vers != 0:
                raise Exception("Unsupported repositoryformatversion %s" % vers)
```

We're going to be manipulating **lots** of paths in repositories. We may as well create a few utility functions to compute those paths and create missing directory structures if needed. First, just a general path building function:

```
def repo_path(repo, *path):
    """Compute path under repo's gitdir."""
    return os.path.join(repo.gitdir, *path)
```

The two next functions, `repo_file()` and `repo_dir()`, return and optionally create a path to a file or a directory, respectively. The difference between them is that the file version only creates directories up to the last component.

```
def repo_file(repo, *path, mkdir=False):
    """Same as repo_path, but create dirname(*path) if absent.  For
example, repo_file(r, \"refs\", \"remotes\", \"origin\", \"HEAD\") will create
.git/refs/remotes/origin."""

    if repo_dir(repo, *path[:-1], mkdir=mkdir):
        return repo_path(repo, *path)

def repo_dir(repo, *path, mkdir=False):
    """Same as repo_path, but mkdir *path if absent if mkdir."""

    path = repo_path(repo, *path)

    if os.path.exists(path):
        if (os.path.isdir(path)):
            return path
        else:
            raise Exception("Not a directory %s" % path)

    if mkdir:
        os.makedirs(path)
        return path
    else:
        return None
```

To **create** a new repository, we start with a directory (which we create if doesn't already exist, or check for emptiness otherwise) and create the following paths:

- `.git` is the git directory itself, which contains:
  - `.git/objects/` : the object store, which we'll introduce in the next section.

- `.git/refs/` the reference store, which we'll discuss . It contains two subdirectories, `heads` and `tags`.
- `.git/HEAD`, a reference to the current HEAD (more on that later!)
- `.git/config`, the repository's configuration file.
- `.git/description`, the repository's description file.

```
def repo_create(path):
    """Create a new repository at path."""

    repo = GitRepository(path, True)

    # First, we make sure the path either doesn't exist or is an
    # empty dir.

    if os.path.exists(repo.worktree):
        if not os.path.isdir(repo.worktree):
            raise Exception ("%s is not a directory!" % path)
        if os.listdir(repo.worktree):
            raise Exception("%s is not empty!" % path)
    else:
        os.makedirs(repo.worktree)

    assert(repo_dir(repo, "branches", mkdir=True))
    assert(repo_dir(repo, "objects", mkdir=True))
    assert(repo_dir(repo, "refs", "tags", mkdir=True))
    assert(repo_dir(repo, "refs", "heads", mkdir=True))

    # .git/description
    with open(repo_file(repo, "description"), "w") as f:
        f.write("Unnamed repository; edit this file 'description' to name the repository.\n")

    # .git/HEAD
    with open(repo_file(repo, "HEAD"), "w") as f:
        f.write("ref: refs/heads/master\n")

    with open(repo_file(repo, "config"), "w") as f:
        config = repo_default_config()
        config.write(f)
```

```
    return repo
```

The configuration file is very simple, it's a INI-like file with a single section (`[core]`) and three fields:

- `repositoryformatversion = 0`: the version of the gitdir format. 0 means the initial format, 1 the same with extensions. If > 1, git will panic; wyag will only accept 0.
- `filemode = false`: disable tracking of file mode changes in the work tree.

- `bare = false`: indicates that this repository has a worktree. Git supports an optional `worktree` key which indicates the location of the worktree, if not `..`; wyag doesn't.

We create this file using Python's `configparser` lib:

```
def repo_default_config():
    ret = configparser.ConfigParser()

    ret.add_section("core")
    ret.set("core", "repositoryformatversion", "0")
    ret.set("core", "filemode", "false")
    ret.set("core", "bare", "false")

    return ret
```

3.2. The init command

Now that we have code to read and create repositories, let's make this code usable from the command line by creating the `wyag init` command. `wyag init` behaves just like `git init` — with much less customizability, of course. The syntax of `wyag init` is going to be:

```
wyag init [path]
```

We already have the complete repository creation logic. To create the command, we're only going to need two more things:

1. We need to create an argparse subparser to handle our command's argument.

   ```
   argsp = argsubparsers.add_parser("init", help="Initialize a new, empty repository.")
   ```

   In the case of `init`, there's a single, optional, positional argument: the path where to init the repo. It defaults to `.`:

   ```
   argsp.add_argument("path",
                      metavar="directory",
                      nargs="?",
                      default=".",
                      help="Where to create the repository.")
   ```

2. We also need a “bridge” function that will read argument values from the object returned by argparse and call the actual function with correct values.

```
def cmd_init(args):
    repo_create(args.path)
```

And we’re done! If you’ve followed these steps, you should now be able to `wyag init` a git repository anywhere.

3.3. The repo_find() function

While we’re implementing repositories, we’re going to need a function to find the root of the current repository. We’ll use it a lot, since almost all Git functions work on an existing repository (except `init`, of course!). Sometimes that root is the current directory, but it may also be a parent: your repository’s root may be in `~/Documents/MyProject`, but you may currently be working in `~/Documents/MyProject/src/tui/frames/mainview/`. The `repo_find()` function we’ll now create will look for that root, starting at the current directory and recursing back to `/`. To identify a path as a repository, it will check for the presence of a `.git` directory.

```
def repo_find(path=".", required=True):
    path = os.path.realpath(path)

    if os.path.isdir(os.path.join(path, ".git")):
        return GitRepository(path)

    # If we haven't returned, recurse in parent, if w
    parent = os.path.realpath(os.path.join(path, ".."))

    if parent == path:
        # Bottom case
        # os.path.join("/", "..") == "/":
        # If parent==path, then path is root.
        if required:
            raise Exception("No git directory.")
        else:
            return None

    # Recursive case
    return repo_find(parent, required)
```

And we're done with repositories!

4. Reading and writing objects: hash-object and cat-file

4.1. What are objects?

Now that we have repositories, putting things inside them is in order. Also, repositories are boring, and writing a Git implementation shouldn't be just a matter of writing a bunch of `mkdir`. Let's talk about **objects**, and let's implement `git hash-object` and `git cat-file`.

Maybe you don't know these two commands — they're not exactly part of an everyday git toolbox, and they're actually quite low-level ("plumbing", in git parlance). What they do is actually very simple: `hash-object` converts an existing file into a git object, and `cat-file` prints an existing git object to the standard output.

Now, **what actually is a Git object?** At its core, Git is a "content-addressed filesystem". That means that unlike regular filesystems, where the name of a file is arbitrary and unrelated to that file's contents, the names of files as stored by Git are mathematically derived from their contents. This has a very important implication: if a single byte of, say, a text file, changes, its internal name will change, too. To put it simply: you don't *modify* a file, you create a new file in a different location. Objects are just that: files in the git repository, whose path is determined by their contents.

> Warning
>
> **Git is not (really) a key-value store**
> Some documentation, including the excellent Pro Git, call Git a "key-value store". This is not incorrect, but may be misleading. Regular filesystems are actually closer to a key-value store than Git is. Because it computes keys from data, Git should rather be called a *value-value store*.

Git uses objects to store quite a lot of things: first and foremost, the actual files it keeps in version control — source code, for example. Commit are objects, too, as well as tags. With a few notable exceptions (which we'll see later!), almost everything, in Git, is stored as an object.

The path is computed by calculating the SHA-1 hash of its contents. More precisely, Git renders the hash as a lowercase hexadecimal string, and splits it in two parts: the first two characters, and the rest. It uses the first part as a directory name, the rest as the file name (this is because most filesystems hate having too many files in a single directory and would slow down to a crawl. Git's method creates 256 possible intermediate directories, hence dividing the average number of files per directory by 256)

> **Note**
>
> **What is a hash function?**
> Simply put, a hash function is a kind of unidirectional mathematical function: it is easy to compute the hash of a value, but there's no way to compute which value produced a hash. A very simple example of a hash function is the `strlen` function. It's really easy to compute the length of a string, and the length of a given string will never change (unless the string itself changes, of course!) but it's impossible to retrieve the original string, given only its length. *Cryptographic* hash functions are just a much more complex version of the same, with the added property that computing an input meant to produce a given hash is hard enough to be practically impossible. (With `strlen`, producing an input `i` with `strlen(i) == 12`, you just have to type twelve random characters. With algorithms such as SHA-1. it would take much, much longer — long enough to be practically impossible[1].

Before we start implementing the object storage system, we must understand their exact storage format. An object starts with a header that specifies its type: `blob`, `commit`, `tag` or `tree`. This header is followed by an ASCII space (0x20), then the size of the object in bytes as an ASCII number,

then null (0x00) (the null byte), then the contents of the object. The first 48 bytes of a commit object in Wyag's repo look like this:

```
00000000  63 6f 6d 6d 69 74 20 31  30 38 36 00 74 72 65 65  |commit 1086.tree|
00000010  20 32 39 66 66 31 36 63  39 63 31 34 65 32 36 35  | 29ff16c9c14e265|
00000020  32 62 32 32 66 38 62 37  38 62 62 30 38 61 35 61  |2b22f8b78bb08a5a|
```

In the first line, we see the type header, a space (`0x20`), the size in ASCII (1086) and the null separator `0x00`. The last four bytes on the first line are the beginning of that object's contents, the word "tree" — we'll discuss that further when we'll talk about commits.

The objects (headers and contents) are stored compressed with `zlib`.

4.2. A generic object object

Objects can be of multiple types, but they all share the same storage/retrieval mechanism and the same general header format. Before we dive into the details of various types of objects, we need to abstract over these common features. The easiest way is to create a generic `GitObject` with two unimplemented methods: `serialize()` and `deserialize()`. Later, we'll subclass this generic class, actually implementing these functions for each object format.

```
class GitObject (object):

    repo = None

    def __init__(self, repo, data=None):
        self.repo=repo

        if data != None:
            self.deserialize(data)

    def serialize(self):
        """This function MUST be implemented by subclasses.

It must read the object's contents from self.data, a byte string, and do
whatever it takes to convert it into a meaningful representation.  What exactly that means depend on eac
        raise Exception("Unimplemented!")

    def deserialize(self, data):
        raise Exception("Unimplemented!")
```

4.3. Reading objects

To read an object, we need to know its hash. We then compute its path from this hash (with the formula explained above: first two characters, then a directory delimiter `/`, then the remaining part) and look it up inside of the "objects" directory in the gitdir. That is, the path to `e673d1b7eaa0aa01b5bc2442d570a765bdaae751` is `.git/objects/e6/73d1b7eaa0aa01b5bc2442d570a765bdaae751`.

We then read that file as a binary file, and decompress it using `zlib`.

From the decompressed data, we extract the two header components: the object type and its size. From the type, we determine the actual class to use. We convert the size to a Python integer, and check if it matches.

When all is done, we just call the correct constructor for that object's format.

```
def object_read(repo, sha):
    """Read object object_id from Git repository repo.  Return a
    GitObject whose exact type depends on the object."""

    path = repo_file(repo, "objects", sha[0:2], sha[2:])

    with open (path, "rb") as f:
        raw = zlib.decompress(f.read())

        # Read object type
        x = raw.find(b' ')
        fmt = raw[0:x]

        # Read and validate object size
        y = raw.find(b'\x00', x)
        size = int(raw[x:y].decode("ascii"))
        if size != len(raw)-y-1:
            raise Exception("Malformed object {0}: bad length".format(sha))

        # Pick constructor
        if   fmt==b'commit' : c=GitCommit
        elif fmt==b'tree'   : c=GitTree
        elif fmt==b'tag'    : c=GitTag
        elif fmt==b'blob'   : c=GitBlob
        else:
            raise Exception("Unknown type {0} for object {1}".format(fmt.decode("ascii"), sha))

        # Call constructor and return object
        return c(repo, raw[y+1:])
```

We haven't introduced the `object_find` function yet. It's actually a placeholder, and looks like this:

```
def object_find(repo, name, fmt=None, follow=True):
    return name
```

The reason for this strange small function is that Git has a *lot* of ways to refer to objects: full hash, short hash, tags… `object_find()` will be our name resolution function. We'll only implement it later, so this is a temporary placeholder. This means that until we implement the real thing, the only way we can refer to an object will be by its full hash.

4.4. Writing objects

Writing an object is reading it in reverse: we compute the hash, insert the header, zlib-compress everything and write the result in place. This really shouldn't require much explanation, just notice that the hash is computed **after** the header is added.

```
def object_write(obj, actually_write=True):
    # Serialize object data
    data = obj.serialize()
    # Add header
    result = obj.fmt + b' ' + str(len(data)).encode() + b'\x00' + data
    # Compute hash
    sha = hashlib.sha1(result).hexdigest()

    if actually_write:
        # Compute path
        path=repo_file(obj.repo, "objects", sha[0:2], sha[2:], mkdir=actually_write)

        with open(path, 'wb') as f:
            # Compress and write
            f.write(zlib.compress(result))

    return sha
```

4.5. Working with blobs

Of the four Git object types, blobs are the simplest, because they have no actual format. Blobs are user content: every file you put in git is stored as a blob. That make them easy to manipulate, because they have no actual syntax or constraints beyond the basic object storage mechanism: they're just unspecified data. Creating a `GitBlob` class is thus trivial, the `serialize` and `deserialize` functions just have to store and return their input unmodified.

```
class GitBlob(GitObject):
    fmt=b'blob'

    def serialize(self):
        return self.blobdata

    def deserialize(self, data):
        self.blobdata = data
```

4.6. The cat-file command

With all that, we can now create `wyag cat-file`. The basic syntax of `git cat-file` is just two positional arguments: a type and an object identifier:

```
git cat-file TYPE OBJECT
```

The subparser is very simple:

```
argsp = argsubparsers.add_parser("cat-file",
                                 help="Provide content of repository objects")

argsp.add_argument("type",
                   metavar="type",
                   choices=["blob", "commit", "tag", "tree"],
                   help="Specify the type")

argsp.add_argument("object",
                   metavar="object",
                   help="The object to display")
```

And the functions themselves shouldn't need any explanation, they just call into already written logic:

```
def cmd_cat_file(args):
    repo = repo_find()
    cat_file(repo, args.object, fmt=args.type.encode())

def cat_file(repo, obj, fmt=None):
    obj = object_read(repo, object_find(repo, obj, fmt=fmt))
    sys.stdout.buffer.write(obj.serialize())
```

4.7. The hash-object command

`hash-object` is basically the opposite of `cat-file`: it reads a file, computes its hash as an object, either storing it in the repository (if the -w flag is passed) or just printing its hash.

The basic syntax of `git hash-object` looks like this:

```
git hash-object [-w] [-t TYPE] FILE
```

Which converts to:

```
argsp = argsubparsers.add_parser(
    "hash-object",
    help="Compute object ID and optionally creates a blob from a file")

argsp.add_argument("-t",
                   metavar="type",
                   dest="type",
                   choices=["blob", "commit", "tag", "tree"],
                   default="blob",
                   help="Specify the type")

argsp.add_argument("-w",
                   dest="write",
                   action="store_true",
                   help="Actually write the object into the database")

argsp.add_argument("path",
                   help="Read object from <file>")
```

The actual implementation is very simple. As usual, we create a small bridge function:

```
def cmd_hash_object(args):
    if args.write:
        repo = GitRepository(".")
    else:
        repo = None

    with open(args.path, "rb") as fd:
        sha = object_hash(fd, args.type.encode(), repo)
        print(sha)
```

and the actual implementation. The repo argument is optional:

```
def object_hash(fd, fmt, repo=None):
    data = fd.read()

    # Choose constructor depending on
    # object type found in header.
    if   fmt==b'commit' : obj=GitCommit(repo, data)
    elif fmt==b'tree'   : obj=GitTree(repo, data)
    elif fmt==b'tag'    : obj=GitTag(repo, data)
    elif fmt==b'blob'   : obj=GitBlob(repo, data)
    else:
        raise Exception("Unknown type %s!" % fmt)

    return object_write(obj, repo)
```

4.8. What about packfiles?

What we've just implemented is called "loose objects". Git has a second object storage mechanism called packfiles. Packfiles are much more efficient, but also much more complex, than loose objects. Simply put, a packfile is a compilation of loose objects (like a `tar`) but some are stored as deltas (as a transformation of another object). Packfiles are way too complex to be supported by wyag.

The packfile is stored in `.git/objects/pack/`. It has a `.pack` extension, and is accompanied by an index file of the same name with the `.idx` extension. Should you want to convert a packfile to loose objects format (to play with `wyag` on an existing repo, for example), here's the solution.

First, *move* the packfile outside the gitdir (just copying it won't work).

```
mv .git/objects/pack/pack-d9ef004d4ca729287f12aaaacf36fee39baa7c9d.pack .
```

You can ignore the `.idx`. Then, from the worktree, just `cat` it and pipe the result to `git unpack-objects`:

```
cat pack-d9ef004d4ca729287f12aaaacf36fee39baa7c9d.pack | git unpack-objects
```

5. Reading commit history: log

5.1. Parsing commits

Now that we can read and write objects, we should consider commits. A commit object (uncompressed, without headers) looks like this:

```
tree 29ff16c9c14e2652b22f8b78bb08a5a07930c147
parent 206941306e8a8af65b66eaaaea388a7ae24d49a0
author Thibault Polge <thibault@thb.lt> 1527025023 +0200
committer Thibault Polge <thibault@thb.lt> 1527025044 +0200
gpgsig -----BEGIN PGP SIGNATURE-----

 iQIzBAABCAAdFiEExwXquOM8bWb4Q2zVGxM2FxoLkGQFAlsEjZQACgkQGxM2FxoL
 kGQdcBAAqPP+ln4nGDd2gETXjvOpOxLzIMEw4A9gU6CzWzm+oB8mEIKyaH0UFIPh
 rNUZ1j7/ZGFNeBDtT55LPdPIQw4KKlcf6kC8MPWP3qSu3xHqx12C5zyai2duFZUU
 wqOt9iCFCscFQYqKs3xsHI+ncQb+PGjVZA8+jPw7nrPIkeSXQV2aZb1E68wa2YIL
 3eYgTUKz34cB6tAq9YwHnZpyPx8UJCZGkshpJmgtZ3mCbtQaO17LoihnqPn4UOMr
 V75R/7FjSuPLS8NaZF4wfi52btXMSxO/u7GuoJkzJscP3p4qtwe6Rl9dc1XC8P7k
 NIbGZ5Yg5cEPcfmhgXFOhQZkD0yxcJqBUcoFpnp2vu5XJl2E5I/quIyVxUXi6O6c
 /obspcvace4wy8uO0bdVhc4nJ+Rla4InVSJaUaBeiHTW8kReSFYyMmDCzLjGIu1q
 doU61OM3Zv1ptsLu3gUE6GU27iWYj2RWN3e3HE4Sbd89IFwLXNdSuM0ifDLZk7AQ
 WBhRhipCCgZhkj9g2NEk7jRVslti1NdN5zoQLaJNqSwO1MtxTmJ15Ksk3QP6kfLB
 Q52UWybBzpaP9HEd4XnR+HuQ4k2K0ns2KgNImsNvIyFwbpMUyUWLMPimaV1DWUXo
 5SBjDB/V/W2JBFR+XKHFJeFwYhj7DD/ocsGr4ZMx/lgc8rjIBkI=
 =lgTX
 -----END PGP SIGNATURE-----

Create first draft
```

The format is a simplified version of mail messages, as specified in RFC 2822. It begins with a series of key-value pairs, with space as the key/value separator, and ends with the commit message, that may span over multiple lines. Values may continue over multiple lines, subsequent lines start with a space which the parser must drop.

Let's have a look at those fields:

- `tree` is a reference to a tree object, a type of object that we'll see soon. A tree maps blobs IDs to filesystem locations, and describes a state of the work tree. Put simply, it is the actual content of the commit: files, and where they go.
- `parent` is a reference to the parent of this commit. It may be repeated: merge commits, for example, have multiple parents. It may also be absent: the very first commit in a repository obviously doesn't have a parent.
- `author` and `committer` are separate, because the author of a commit is not necessarily the person who can commit it (This may not be obvious for GitHub users, but a lot of projects do Git through e-mail)
- `gpgsig` is the PGP signature of this object.

We'll start by writing a simple parser for the format. The code is obvious. The name of the function we're about to create, `kvlm_parse()`, may be confusing: it isn't called `commit_parse()` because tags have the very same format, so we'll use it for both objects types. I use KVLM to mean "Key-Value List with Message".

```
def kvlm_parse(raw, start=0, dct=None):
    if not dct:
        dct = collections.OrderedDict()
        # You CANNOT declare the argument as dct=OrderedDict() or all
        # call to the functions will endlessly grow the same dict.

    # We search for the next space and the next newline.
    spc = raw.find(b' ', start)
    nl = raw.find(b'\n', start)

    # If space appears before newline, we have a keyword.

    # Base case
    # =========
    # If newline appears first (or there's no space at all, in which
    # case find returns -1), we assume a blank line.  A blank line
    # means the remainder of the data is the message.
    if (spc < 0) or (nl < spc):
        assert(nl == start)
        dct[b''] = raw[start+1:]
        return dct

    # Recursive case
    # ==============
    # we read a key-value pair and recurse for the next.
    key = raw[start:spc]

    # Find the end of the value.  Continuation lines begin with a
    # space, so we loop until we find a "\n" not followed by a space.
    end = start
    while True:
        end = raw.find(b'\n', end+1)
        if raw[end+1] != ord(' '): break
```

```
    # Grab the value
    # Also, drop the leading space on continuation lines
    value = raw[spc+1:end].replace(b'\n ', b'\n')

    # Don't overwrite existing data contents
    if key in dct:
        if type(dct[key]) == list:
            dct[key].append(value)
        else:
            dct[key] = [ dct[key], value ]
    else:
        dct[key]=value

    return kvlm_parse(raw, start=end+1, dct=dct)
```

We use an `OrderedDict` here because `cat-file`, as we've implemented it, will print a commit object by parsing it and re-serializing it, so we need fields to be in the exact same order they were defined. Also, in Git, the order keys appear in commit and tag object seem to matter.

We're going to need to write similar objects, so let's add a `kvlm_serialize()` function to our toolkit.

```
def kvlm_serialize(kvlm):
    ret = b''

    # Output fields
    for k in kvlm.keys():
        # Skip the message itself
        if k == b'': continue
        val = kvlm[k]
        # Normalize to a list
        if type(val) != list:
            val = [ val ]

        for v in val:
            ret += k + b' ' + (v.replace(b'\n', b'\n ')) + b'\n'

    # Append message
    ret += b'\n' + kvlm[b'']

    return ret
```

5.2. The Commit object

Now we have the parser, we can create the `GitCommit` class:

```
class GitCommit(GitObject):
    fmt=b'commit'

    def deserialize(self, data):
        self.kvlm = kvlm_parse(data)

    def serialize(self):
        return kvlm_serialize(self.kvlm)
```

5.3. The log command

We'll implement a much, much simpler version of `log` than what Git provides. Most importantly, we won't deal with representing the log *at all*. Instead, we'll dump Graphviz data and let the user use `dot` to render the actual log. (If you don't know how to use Graphviz, just paste the raw output into this site. If the link is dead, lookup "graphviz online" in your favorite search engine)

```
argsp = argsubparsers.add_parser("log", help="Display history of a given commit.")
argsp.add_argument("commit",
                   default="HEAD",
                   nargs="?",
                   help="Commit to start at.")
```

```
def cmd_log(args):
    repo = repo_find()

    print("digraph wyaglog{")
    log_graphviz(repo, object_find(repo, args.commit), set())
    print("}")

def log_graphviz(repo, sha, seen):

    if sha in seen:
        return
    seen.add(sha)

    commit = object_read(repo, sha)
    assert (commit.fmt==b'commit')

    if not b'parent' in commit.kvlm.keys():
        # Base case: the initial commit.
        return

    parents = commit.kvlm[b'parent']

    if type(parents) != list:
        parents = [ parents ]

    for p in parents:
        p = p.decode("ascii")
        print ("c_{0} -> c_{1};".format(sha, p))
        log_graphviz(repo, p, seen)
```

You can now use our log command like this:

```
wyag log e03158242ecab460f31b0d6ae1642880577ccbe8 > log.dot
dot -O -Tpdf log.dot
```

5.4. Anatomy of a commit

You may have noticed a few things right now.

First and foremost, we've been playing with commits, browsing and walking through commit objects, building a graph of commit history, without ever touching a single file in the worktree or a blob. We've done a lot with commits *without considering their contents*. This is important: work tree contents are just a part of a commit. But a commit is made of everything: its contents, its authors, and also its parents. If you remember that the ID (the SHA-1 hash) of a commit is computed from the whole commit object, you'll understand what it means that commits are immutable: if you change the author, the parent commit or a single file, you've actually created a new, different object. Each and every commit is bound to its place and its relationship to the whole repository up to the very first commit. To put it otherwise, a given commit ID not only identifies some file contents, but it also binds the commit to its whole history and to the whole repository.

It's also worth noting that from the point of view of a commit, time runs backwards: we're used to considering the history of a project from its humble beginnings as an evening distraction, starting with a few lines of code, some initial commits, and progressing to its present state (millions of lines of code, dozens of contributors, whatever). But each commit is completely unaware of its future, it's only linked to the past. Commits have "memory", but no premonition.

Note

In Terry Pratchett's Discworld, trolls believe they progress in time from the future to the past. The reasoning behind that belief is that when you walk, what you can see is what's *ahead* of you. Of time, all you can perceive is the past, because you remember; hence it's where you're headed. Git was written by a Discworld troll.

So what makes a commit? To sum it up:

- A tree object, which we'll discuss now, that is, the contents of a worktree, files and directories;
- Zero, one or more parents;
- An author identity (name and email);
- A committer identity (name and email);
- An optional PGP signature
- A message;

All this hashed together in a SHA-1 identifier.

 Note

Wait, does that make Git a blockchain?
Because of cryptocurrencies, blockchains are all the hype these days. And yes, *in a way*, Git is a blockchain: it's a sequence of blocks (commits) tied together by cryptographic means in a way that guarantee that each single element is associated to the whole history of the structure. Don't take the comparison too seriously, though: we don't need a GitCoin. Really, we don't.

6. Reading commit data: checkout

It's all well that commits hold a lot more than files and directories in a given state, but that doesn't make them really useful. It's probably time to start implementing tree objects as well, so we'll be able to checkout commits into the work tree.

6.1. What's in a tree?

Informally, a tree describes the content of the work tree, that it, it associates blobs to paths. It's an array of three-element tuples made of a file mode, a path (relative to the worktree) and a SHA-1. A typical tree contents may look like this:

Mode	SHA-1	Path
`100644`	`894a44cc066a027465cd26d634948d56d13af9af`	`.gitignore`
`100644`	`94a9ed024d3859793618152ea559a168bbcbb5e2`	`LICENSE`
`100644`	`bab489c4f4600a38ce6dbfd652b90383a4aa3e45`	`README.md`
`100644`	`6d208e47659a2a10f5f8640e0155d9276a2130a9`	`src`
`040000`	`e7445b03aea61ec801b20d6ab62f076208b7d097`	`tests`
`040000`	`d5ec863f17f3a2e92aa8f6b66ac18f7b09fd1b38`	`main.c`

Mode is just the file's mode, path is its location. The SHA-1 refers to either a blob or another tree object. If a blob, the path is a file, if a tree, it's directory. To instantiate this tree in the filesystem, we would begin by loading the object associated to the first path (`.gitignore`) and check its type. Since it's a blob, we'll just create a file called `.gitignore` with this blob's contents; and same for `LICENSE` and `README.md`. But the object associated with `src` is not a blob, but another tree: we'll create the directory `src` and repeat the same operation in that directory with the new tree.

> **Warning**
>
> **A path is a single filesystem entry**
> The path identifies exactly one object. Not two, not three. If you have five levels of nested directories, you're going to need five tree objects recursively referring to one another. You cannot take the shortcut of putting a full path in a single tree entry, like `dir1/dir2/dir3/dir4/dir5`.

6.2. Parsing trees

Unlike tags and commits, tree objects are binary objects, but their format is actually quite simple. A tree is the concatenation of records of the format:

```
[mode] space [path] 0x00 [sha-1]
```

- `[mode]` is up to six bytes and is an ASCII representation of a file mode. For example, 100644 is encoded with byte values 49 (ASCII "1"), 48 (ASCII "0"), 48, 54, 52, 52.
- It's followed by 0x20, an ASCII space;
- Followed by the null-terminated (0x00) path;

- Followed by the object's SHA-1 in binary encoding, on 20 bytes. Why binary? God only knows.

The parser is going to be quite simple. First, create a tiny object wrapper for a single record (a leaf, a single path):

```
class GitTreeLeaf (object):
    def __init__(self, mode, path, sha):
        self.mode = mode
        self.path = path
        self.sha = sha
```

Because a tree object is just the repetition of the same fundamental data structure, we write the parser in two functions. First, a parser to extract a single record, which returns parsed data and the position it reached in input data:

```
def tree_parse_one(raw, start=0):
    # Find the space terminator of the mode
    x = raw.find(b' ', start)
    assert(x-start == 5 or x-start==6)

    # Read the mode
    mode = raw[start:x]

    # Find the NULL terminator of the path
    y = raw.find(b'\x00', x)
    # and read the path
    path = raw[x+1:y]

    # Read the SHA and convert to a hex string
    sha = format(int.from_bytes(raw[y+1:y+21], "big"), "040x")
    return y+21, GitTreeLeaf(mode, path, sha)
```

And the “real” parser which just calls the previous one in a loop, until input data is exhausted.

```
def tree_parse(raw):
    pos = 0
    max = len(raw)
    ret = list()
    while pos < max:
        pos, data = tree_parse_one(raw, pos)
        ret.append(data)

    return ret
```

Last but not least, we’ll need a serializer:

```
def tree_serialize(obj):
    ret = b''
    for i in obj.items:
        ret += i.mode
        ret += b' '
        ret += i.path
        ret += b'\x00'
        sha = int(i.sha, 16)
        ret += sha.to_bytes(20, byteorder="big")
    return ret
```

And now we just have to combine all that into a class:

```
class GitTree(GitObject):
    fmt=b'tree'

    def deserialize(self, data):
        self.items = tree_parse(data)

    def serialize(self):
        return tree_serialize(self)
```

While we're at it, let's add the `ls-tree` command to wyag. It's so easy there's no reason not to.

```
argsp = argsubparsers.add_parser("ls-tree", help="Pretty-print a tree object.")
argsp.add_argument("object",
                   help="The object to show.")

def cmd_ls_tree(args):
    repo = repo_find()
    obj = object_read(repo, object_find(repo, args.object, fmt=b'tree'))

    for item in obj.items:
        print("{0} {1} {2}\t{3}".format(
            "0" * (6 - len(item.mode)) + item.mode.decode("ascii"),
            # Git's ls-tree displays the type
            # of the object pointed to.  We can do that too :)
            object_read(repo, item.sha).fmt.decode("ascii"),
            item.sha,
            item.path.decode("ascii")))
```

6.3. The checkout command

We're going to oversimplify the actual git command to make our implementation clear and understandable. We're also going to add a few safeguards. Here's how our version of checkout will work:

- It will take two arguments: a commit, and a directory. Git checkout only needs a commit.
- It will then instantiate the tree in the directory, **if and only if the directory is empty**. Git is full of safeguards to avoid deleting data, which would be too complicated and unsafe to try to reproduce in wyag. Since the point of wyag is to demonstrate git, not to produce a working implementation, this limitation is acceptable.

Let's get started. As usual, we need a subparser:

```
argsp = argsubparsers.add_parser("checkout", help="Checkout a commit inside of a directory.")

argsp.add_argument("commit",
                   help="The commit or tree to checkout.")

argsp.add_argument("path",
                   help="The EMPTY directory to checkout on.")
```

A wrapper function:

```
def cmd_checkout(args):
    repo = repo_find()

    obj = object_read(repo, object_find(repo, args.commit))

    # If the object is a commit, we grab its tree
    if obj.fmt == b'commit':
        obj = object_read(repo, obj.kvlm[b'tree'].decode("ascii"))

    # Verify that path is an empty directory
    if os.path.exists(args.path):
        if not os.path.isdir(args.path):
            raise Exception("Not a directory {0}!".format(args.path))
        if os.listdir(args.path):
            raise Exception("Not empty {0}!".format(args.path))
    else:
        os.makedirs(args.path)

    tree_checkout(repo, obj, os.path.realpath(args.path).encode())
```

And a function to do the actual work:

```
def tree_checkout(repo, tree, path):
    for item in tree.items:
        obj = object_read(repo, item.sha)
        dest = os.path.join(path, item.path)

        if obj.fmt == b'tree':
            os.mkdir(dest)
            tree_checkout(repo, obj, dest)
        elif obj.fmt == b'blob':
            with open(dest, 'wb') as f:
                f.write(obj.blobdata)
```

7. Refs, tags and branches

7.1. What's a ref?

Git references, or refs, are probably the most simple type of things git holds. They live in subdirectories of `.git/refs`, and are text files containing a hexadecimal representation of an object's hash, encoded in ASCII. They're actually as simple as this:

```
6071c08bcb4757d8c89a30d9755d2466cef8c1de
```

Refs can also refer to another reference, and thus only indirectly to an object, in which case they look like this:

```
ref: refs/remotes/origin/master
```

> Note
>
> **Direct and indirect references**
> From now on, I will call a reference of the form `ref: path/to/other/ref` an **indirect** reference, and a ref with a SHA-1 object ID a **direct reference**.

This whole section will describe the uses of refs. For now, all that matter is this:

- they're text files, in the `.git/refs` hierarchy;
- they hold the sha-1 identifier of an object, or a reference to another reference.

To work with refs, we're first going to need a simple recursive solver that will take a ref name, follow eventual recursive references (refs whose content begin with `ref:`, as exemplified above) and return a SHA-1:

```
def ref_resolve(repo, ref):
    with open(repo_file(repo, ref), 'r') as fp:
        data = fp.read()[:-1]
        # Drop final \n ^^^^^
    if data.startswith("ref: "):
        return ref_resolve(repo, data[5:])
    else:
        return data
```

Let's create two small functions, and implement the `show-refs` command. First, a stupid recursive function to collect refs and return them as a dict:

```
def ref_list(repo, path=None):
    if not path:
        path = repo_dir(repo, "refs")
    ret = collections.OrderedDict()
    # Git shows refs sorted.  To do the same, we use
    # an OrderedDict and sort the output of listdir
    for f in sorted(os.listdir(path)):
        can = os.path.join(path, f)
        if os.path.isdir(can):
            ret[f] = ref_list(repo, can)
        else:
            ret[f] = ref_resolve(repo, can)

    return ret
```

```
argsp = argsubparsers.add_parser("show-ref", help="List references.")

def cmd_show_ref(args):
    repo = repo_find()
    refs = ref_list(repo)
    show_ref(repo, refs, prefix="refs")

def show_ref(repo, refs, with_hash=True, prefix=""):
    for k, v in refs.items():
        if type(v) == str:
            print ("{0}{1}{2}".format(
                v + " " if with_hash else "",
                prefix + "/" if prefix else "",
                k))
        else:
            show_ref(repo, v, with_hash=with_hash, prefix="{0}{1}{2}".format(prefix, "/" if prefix else
```

7.2. What's a tag?

The most simple use of refs is tags. A tag is just a user-defined name for an object, often a commit. A very common use of tags is identifying software releases: You've just merged the last commit of, say, version 12.78.52 of your program, so your most recent commit (let's call it `6071c08`) *is* your version 12.78.52. To make this association explicit, all you have to do is:

```
git tag v12.78.52 6071c08
# the object hash ^here^^ is optional and defaults to HEAD.
```

This creates a new tag, called `v12.78.52`, pointing at `6071c08`. Tagging is like aliasing: a tag introduces a new way to refer to an existing object. After the tag is created, the name `v12.78.52` refers to `6071c08`. For example, these two commands are now perfectly equivalent:

```
git checkout v12.78.52
git checkout 6071c08
```

Note

Versions are a common use of tags, but like almost everything in Git, tags have no predefined semantics: they mean whatever you want them to mean, and can point to whichever object you want, you can even tag *blobs*!

7.3. Parsing tag objects

You've probably guessed already that tags are actually refs. They live in the `.git/refs/tags/` hierarchy. The only point worth noting is that they come in two flavors: lightweight tags and tags objects.

"Lightweight" tags
: are just regular refs to a commit, a tree or a blob.

Tag objects
: are regular refs pointing to an object of type `tag`. Unlike lightweight tags, tag objects have an author, a date, an optional PGP signature and an optional annotation. Their format is the same as a commit object.

We don't even need to implement tag objects, we can reuse `GitCommit` and just change the `fmt` field:

```
class GitTag(GitCommit):
    fmt = b'tag'
```

And now we support tags.

7.4. The tag command

Let's add the tag command. In Git, it does two things: it creates a new tag or list existing tags (by default). So you can invoke it with:

```
git tag                 # List all tags
git tag NAME [OBJECT]   # create a new *lightweight* tag NAME, pointing
                        # at HEAD (default) or OBJECT
git tag -a NAME [OBJECT] # create a new tag *object* NAME, pointing at
                        # HEAD (default) or OBJECT
```

This translates to argparse as follows. Notice we ignore the mutual exclusion between `--list` and `[-a] name [object]`, which seems too complicated for argparse.

```
argsp = argsubparsers.add_parser(
    "tag",
    help="List and create tags")

argsp.add_argument("-a",
                   action="store_true",
                   dest="create_tag_object",
                   help="Whether to create a tag object")

argsp.add_argument("name",
                   nargs="?",
                   help="The new tag's name")

argsp.add_argument("object",
                   default="HEAD",
                   nargs="?",
                   help="The object the new tag will point to")
```

The `cmd_tag` function will dispatch behavior (list or create) depending on whether or not `name` is provided.

```
def cmd_tag(args):
    repo = repo_find()

    if args.name:
        tag_create(repo,
                   args.name,
                   args.object,
                   type="object" if args.create_tag_object else "ref")
    else:
        refs = ref_list(repo)
        show_ref(repo, refs["tags"], with_hash=False)
```

And we just need one more function to actually create the tag:

```
def tag_create(repo, name, reference, type):
    # get the GitObject from the object reference
    sha = object_find(repo, reference)

    if type=="object":
        # create tag object (commit)
        tag = GitTag(repo)
        tag.kvlm = collections.OrderedDict()
        tag.kvlm[b'object'] = sha.encode()
        tag.kvlm[b'type'] = b'commit'
        tag.kvlm[b'tag'] = name.encode()
        # Feel free to let the user give their name!
        tag.kvlm[b'tagger'] = b'Wyag <wyag@example.com>'
        # …and a tag message!
        tag.kvlm[b''] = b"A tag generated by wyag, which won't let you customize the message!"
        tag_sha = object_write(tag, repo)
        # create reference
        ref_create(repo, "tags/" + name, tag_sha)
    else:
        # create lightweight tag (ref)
        ref_create(repo, "tags/" + name, sha)

def ref_create(repo, ref_name, sha):
    with open(repo_file(repo, "refs/" + ref_name), 'w') as fp:
        fp.write(sha + "\n")
```

7.5. What's a branch?

It's time to address the elephant in the room: like most Git users, wyag still doesn't have any idea what a branch is. It currently treats a repository as a bunch of disorganized objects, some of them commits, and has no representation whatsoever of the fact that commits are grouped in branches,

and that at every point in time there's a commit that's `HEAD`, *ie*, the **head** commit of the **current** branch.

Now, what's a branch? The answer is actually surprisingly simple, but it may also end up being simply surprising: **a branch is a reference to a commit**. You could even say that a branch is a kind of a name for a commit. In this regard, a branch is exactly the same thing as a tag. Tags are refs that live in `.git/refs/tags`, branches are refs that live in `.git/refs/heads`.

There are, of course, differences between a branch and a tag:

1. Branches are references to a *commit*, tags can refer to any object;
2. But most importantly, the branch ref is updated at each commit. This means that whenever you commit, Git actually does this:
    1. a new commit object is created, with the current branch's ID as its parent;
    2. the commit object is hashed and stored;
    3. the branch ref is updated to refer to the new commit's hash.

That's all.

But what about the **current** branch? It's actually even easier. It's a ref file outside of the `refs` hierarchy, in `.git/HEAD`, which is an **indirect** ref (that is, it is of the form `ref: path/to/other/ref`, and not a simple hash).

Note

Detached HEAD

When you just checkout a random commit, git will warn you it's in "detached HEAD state". This means you're not on any branch anymore. In this case, `.git/HEAD` is a **direct** reference: it contains a SHA-1.

7.6. Referring to objects: the `object_find` function

7.6.1. Resolving names

Remember when we've created the stupid `object_find` function that would take four arguments, return the second unmodified and ignore the other three? It's time to replace it by something more useful. We're going to implement a small, but usable, subset of the actual Git name resolution algorithm. The new `object_find()` will work in two steps: first, given a name, it will return a complete sha-1 hash. For example, with `HEAD`, it will return the hash of the head commit of the current branch, etc. More precisely, this name resolution function will work like this:

- If `name` is HEAD, it will just resolve `.git/HEAD`;
- If `name` is a full hash, this hash is returned unmodified.
- If `name` looks like a short hash, it will collect objects whose full hash begin with this short hash.
- At last, it will resolve tags and branches matching name.

Notice how the last two steps *collect* values: the first two are absolute references, so we can safely return a result. But short hashes or branch names can be ambiguous, we want to enumerate all possible meanings of the name and raise an error if we've found more than 1.

Short hashes

For convenience, Git allows to refer to hashes by a prefix of their name. For example, `5bd254aa973646fa16f66d702a5826ea14a3eb45` can be referred to as `5bd254`. This is called a “short hash”.

```
def object_resolve(repo, name):
    """Resolve name to an object hash in repo.

This function is aware of:

 - the HEAD literal
 - short and long hashes
 - tags
 - branches
 - remote branches"""
    candidates = list()
    hashRE = re.compile(r"^[0-9A-Fa-f]{4,40}$")

    # Empty string?  Abort.
    if not name.strip():
        return None

    # Head is nonambiguous
    if name == "HEAD":
        return [ ref_resolve(repo, "HEAD") ]

    if hashRE.match(name):
        if len(name) == 40:
            # This is a complete hash
            return [ name.lower() ]

        # This is a small hash 4 seems to be the minimal length
        # for git to consider something a short hash.
        # This limit is documented in man git-rev-parse
        name = name.lower()
        prefix = name[0:2]
        path = repo_dir(repo, "objects", prefix, mkdir=False)
        if path:
            rem = name[2:]
```

```
            for f in os.listdir(path):
                if f.startswith(rem):
                    candidates.append(prefix + f)

    return candidates
```

The second step is to follow the object we found to an object of the required type, if a type argument was provided. Since we only need to handle trivial cases, this is a very simple iterative process:

- If we have a tag and `fmt` is anything else, we follow the tag.
- If we have a commit and `fmt` is tree, we return this commit's tree object
- In all other situations, we abort.

(The process is iterative because it may take an undefined number of steps, since tags themselves can be tagged)

```
def object_find(repo, name, fmt=None, follow=True):
    sha = object_resolve(repo, name)

    if not sha:
        raise Exception("No such reference {0}.".format(name))

    if len(sha) > 1:
        raise Exception("Ambiguous reference {0}: Candidates are:\n - {1}.".format(name,  "\n - ".join(s

    sha = sha[0]

    if not fmt:
        return sha

    while True:
        obj = object_read(repo, sha)

        if obj.fmt == fmt:
            return sha

        if not follow:
            return None

        # Follow tags
        if obj.fmt == b'tag':
            sha = obj.kvlm[b'object'].decode("ascii")
        elif obj.fmt == b'commit' and fmt == b'tree':
            sha = obj.kvlm[b'tree'].decode("ascii")
        else:
            return None
```

7.6.2. The rev-parse command

The `git rev-parse` commands does a lot, but one of its use cases is solving revision (commits) references. For the purpose of testing the "follow" feature of `object_find`, we'll add an optional `wyag-type` arguments to its interface.

```
argsp = argsubparsers.add_parser(
    "rev-parse",
    help="Parse revision (or other objects )identifiers")

argsp.add_argument("--wyag-type",
                   metavar="type",
                   dest="type",
                   choices=["blob", "commit", "tag", "tree"],
                   default=None,
                   help="Specify the expected type")

argsp.add_argument("name",
                   help="The name to parse")
```

```
def cmd_rev_parse(args):
    if args.type:
        fmt = args.type.encode()
    else:
        fmt = None

    repo = repo_find()

    print (object_find(repo, args.name, fmt, follow=True))
```

8. The staging area and the index file

This final step will bring us to actually create commits.

You probably know that to commit in Git, you first "stage" some changes, using `git add` and `git rm`, *then* you commit them. It would seem logical to use a commit-like object to represent the staging area, but Git goes a completely different way, and uses a completely different mechanism, in the form of the index file. The **index file** feels a bit like a tree object: it associates paths to blobs, and so on.

Note

The staging area
The contents of the index file are called the **staging area**. The staging area is not the same as the worktree, because git lets you stage (with `git add -i`) "chunks", that is, only a *part* of changes made to a file. The staging area may thus be different from *both* HEAD and the worktree.
If it's unclear, imagine you've modified two unrelated functions from a single source file. You may want to turn your two changes into two different commits. From a diff, you'd stage the first change only, commit it, then stage the rest of the changes before committing them in turn.

Note

Why not just another tree?

> I believe there are two reasons why the index file isn't just another tree object. The first reason is anecdotal: performance. The index is uncompressed, and stores data in raw form. The most important reason, thus, is that a tree is always a complete, unambiguous representation, whereas the index needs to also represent conflicts (like merge conflicts, when automatic merge failed). Strictly speaking, indexes are a superset of trees.

8.1. Parsing the index

The index file is by far the most complicated piece of data a Git repository can hold. Its complete documentation can be found in Git source tree at `Documentation/gitformat-index.txt`; you can read it on the Github mirror. The index is made of three parts:

- A classic header with a signature and a few basic info, most importantly the number of entries it holds;
- A series of entries, sorted, each representing a change;
- A series of optional extensions, which we'll ignore.

The first thing we need to represent is a single entry. It actually holds quite a lot of stuff:

```
class GitIndexEntry (object):
    def __init__(self, ctime=None, mtime=None, dev=None, ino=None,
                 mode_type=None, mode_perms=None, uid=None, gid=None,
                 fsize=None, object_hash=None, flag_assume_valid=None,
                 flag_extended=None, flag_stage=None,
                 flag_name_length=None, name=None):
      """The last time a file's metadata changed.  This is a tuple (seconds, nanoseconds)"""
      self.ctime = ctime
      """The last time a file's data changed.  This is a tuple (seconds, nanoseconds)"""
      self.mtime = mtime
      """The ID of device containing this file"""
      self.dev = dev
      """The file's inode number"""
      self.ino = ino
      """The object type, either b1000 (regular), b1010 (symlink), b1110 (gitlink). """
      self.mode_type = mode_type
      """The object permissions, an integer."""
      self.mode_perms = mode_perms
      """User ID of owner"""
      self.uid = uid
      """Group ID of ownner (according to stat 2.  Isn'th)"""
      self.gid = gid
      """Size of this object, in bytes"""
      self.fsize = fsize
      """The object's hash as a hex string"""
      self.object_hash = object_hash
      self.flag_assume_valid = flag_assume_valid
      self.flag_extended = flag_extended
      self.flag_stage = flag_stage
      """Length of the name if < 0xFFF (yes, three Fs), -1 otherwise"""
      self.flag_name_length = flag_name_length
      self.name = name
```

```
class GitIndex (object):
    signature = None
    version = None
    entries = []
    # ext = None
    # sha = None

    def __init__(self, file):
        raw = None
        with open(file, 'rb') as f:
            raw = f.read()

        header = raw[:12]
        self.signature = header[:4]
        self.version = hex(int.from_bytes(header[4:8], "big"))
        nindex = int.from_bytes(header[8:12], "big")

        self.entries = list()

        content = raw[12:]
        idx = 0
        for i in range(0, nindex):
            ctime = content[idx: idx+8]
            mtime = content[idx+8: idx+16]
            dev = content[idx+16: idx+20]
            ino = content[idx+20: idx+24]
            mode = content[idx+24: idx+28] # TODO
            uid = content[idx+28: idx+32]
            gid = content[idx+32: idx+36]
            fsize = content[idx+36: idx+40]
            object_hash = content[idx+40: idx+60]
            flag = content[idx+60: idx+62] # TODO
            null_idx = content.find(b'\x00', idx+62) # TODO
            name = content[idx+62: null_idx]
```

```
            idx = null_idx + 1
            idx = 8 * ceil(idx / 8)

            self.entries.append(
                GitIndexEntry(ctime=ctime, mtime=mtime, dev=dev, ino=ino, mode_type=mode, uid=uid, gid=g
```

8.2. The ls-files command

```
argsp = argsubparsers.add_parser("ls-files", help = "List all the stage files")

def cmd_ls_files(args):
  repo = repo_find()
  for e in GitIndex(os.path.join(repo.gitdir, 'index')).entries:
    print("{0}".format(e.name.decode("utf8")))
```

9. Final word

9.1. Comments, feedback and issues

This page has no comment system. To report issues or just give feedback, you may use:

Github
: If you have a Github account, you can create an issue, either to report a problem or a bug, or just for) general feedback :

E-mail
: If you don't have/want a Github account, you can send me an e-mail at thibault@thb.lt.

Pull requests
: are accepted (and welcome!) by e-mail or on Github. Before you start making any changes, please read the next section.

IRC
: I'm thblt on Libera, feel free to message me!

9.2. Contributing

If you want to contribute changes or improvements to this article or the wyag program, library and/or test suite, please read the following before you start working.

- Do *not* make changes to `wyag` or `libwyag.py`. These files are extracted ("tangled") from `write-yourself-a-git.org` and are included in the repository as a convenience. Instead, search the code blocks corresponding to the changes you want to make and update them.
- The source file for *both* the article and the program is `write-yourself-a-git.org`. Its format may seem unusual, but it's pretty simple and not that far from markdown.

9.3. Release information

Key	Value
Creation date	Thu Apr 20 13:21:41 2023
On commit	`0.1-53-ge306292` (clean)
By	(`thblt` on `dru`)
Emacs version	30.0.50

Key	Value
Org-mode version	9.6.1

9.4. License

[1]

> You may know that collisions have been discovered in SHA-1. Git actually doesn't use SHA-1 anymore: it uses a hardened variant which is not SHA, but which applies the same hash to every known input but the two PDF files known to collide.