GIT'S GUTS

delving with a little help from Python ☺

INSPIRATION

- inspired by a PluralSight course "**How Git Works**" by Paolo Perrotta

- his course decomposes git into **functional layers**:

high level

low level

layer	concepts
DISTRIBUTED	remotes, fetch, push
VERSION CONTROL	branch, merge, diff
CONTENT TRACKER	blobs, trees, commits
OBJECT DATABASE	hashes, store, get

API used by developers

implementation details

DISCLAIMER

- I am not a git expert! ☺

- this will not teach you how to use git
  - …but may help demystify some of its workings

- this is not scary or difficult!

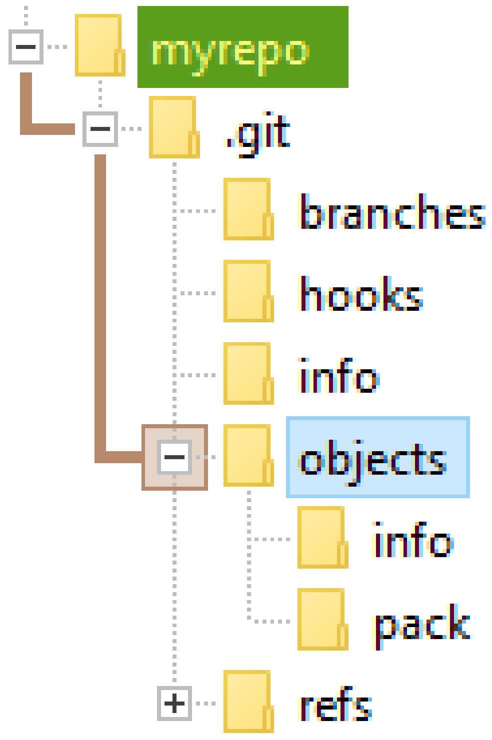
GIT OBJECT DATABASE

CREATING A GIT OBJECT DATABASE

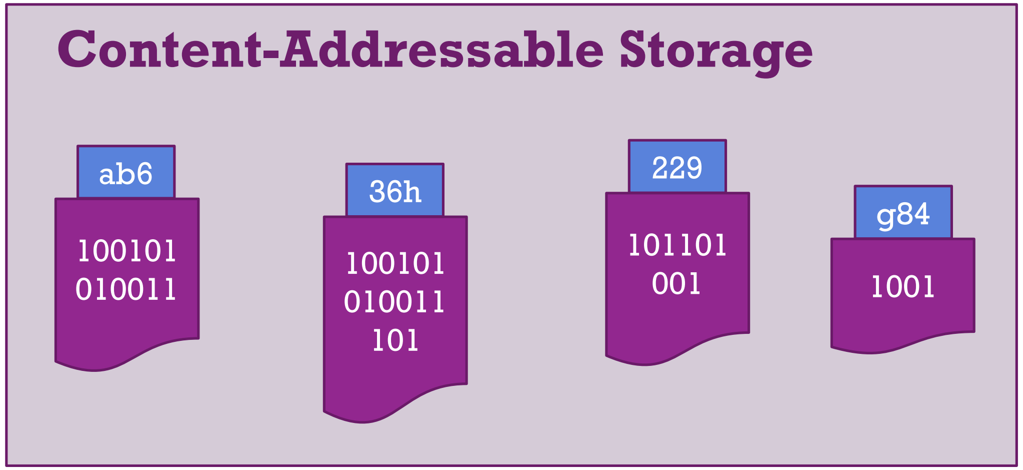
- easy! – initialize a new git repo:
  - **mkdir myrepo**
  - **cd myrepo**
  - **git init**

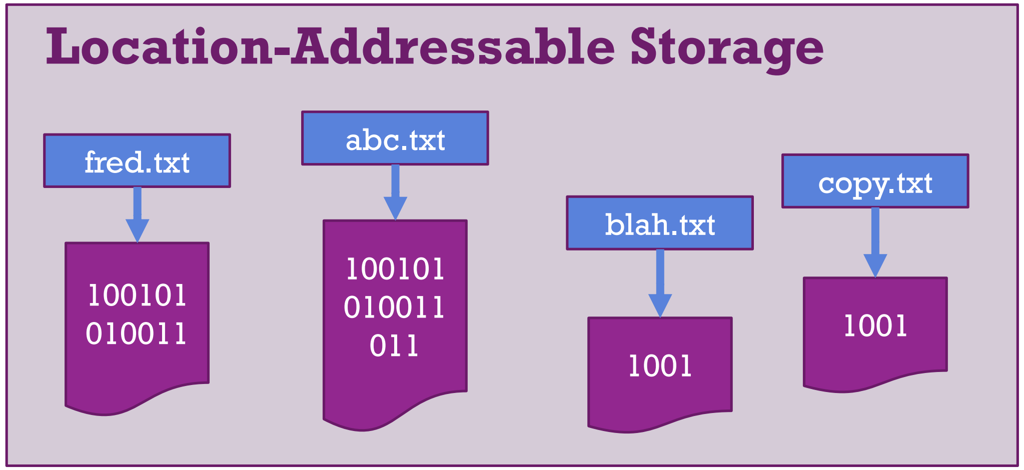
- this will create a subfolder:
  - **.git/objects**
  - (ignore the other files/folders)

- this is used as **content-addressable storage**



ASIDE: CONTENT-ADDRESSABLE STORAGE?!



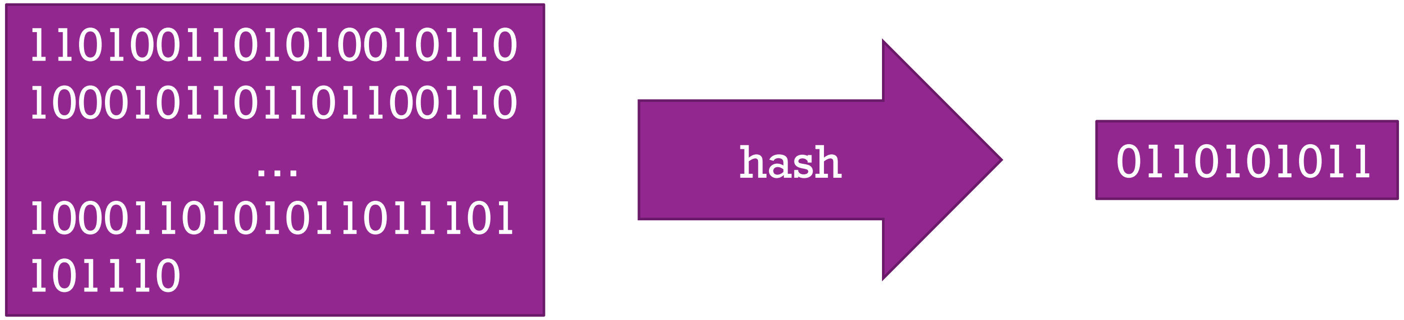


- **content address** is based on the object content alone, usually a **hash**
- can't have two different objects with the same content
- efficient for storing objects where content rarely changes

- objects are **addressed** by a **location**, eg. path/filename
- can have separate objects with duplicate content
- efficient for storing objects where content changes frequently

ASIDE: WHAT IS A HASH?

- a **hash function** maps arbitrary-sized data (eg. file contents) to fixed-size data:



- **hash functions** have many applications throughout computing
  - data integrity, cryptography, data structures, file systems, caching, cryptocurrency …

- there are many different **hash functions** with different characteristics
  - need to pick the appropriate one for your application!

GIT USES SHA-1

- SHA-1 is a **cryptographic hash function**
- it produces a 20-byte output hash
  - (commonly displayed as 40 hex chars)
- accidental collisions are **extremely unlikely**
  - but SHA-1 is no longer considered cryptographically secure…
  - … git is migrating to SHA-256

- there is a SHA-1 implementation in the python standard library:

```
>>> import hashlib
>>> hash = hashlib.sha1()
>>> hash.update(b"The quick brown fox")
>>> hash.hexdigest()
'c519c1a06cdbeb2bc499e22137fb48683858b345'
```

- also the linux **sha1sum** command:

```
$ echo -n "The quick brown fox" | sha1sum
c519c1a06cdbeb2bc499e22137fb48683858b345  -
```

GIT OBJECT FORMAT

- there is a git command to calculate the SHA1 hash of some content:
  - **git hash-object**

```
$ echo -en "Hello world!" | git hash-object --stdin
6769dd60bdf536a83c9353272157893043e9f7d0
```

- but git prefixes a **header** to the content before it calculates the **hash**:
  - **object type** – eg. “blob”
  - **space character**
  - **content length** – eg. “12”
  - **zero byte**

```
>>> import hashlib
>>> hash = hashlib.sha1()
>>> hash.update(b"blob 12\0Hello world!")
>>> hash.hexdigest()
'6769dd60bdf536a83c9353272157893043e9f7d0'
```

STORING AN OBJECT

- we can use the same command to **store** an object in the database
  - just add the **-w** option

- this creates a new folder and file in the **.git/objects** folder:
  - subfolder name is first 2 chars of hash
  - filename is remaining 38 chars of hash

- subfolder is created to avoid storing all objects in a single folder

```
$ echo -en "Hello world!" | git hash-object --stdin -w
6769dd60bdf536a83c9353272157893043e9f7d0
```

```
.git
├── HEAD
├── branches
├── config
├── description
├── hooks
├── info
├── objects
│   ├── 67
│   │   └── 69dd60bdf536a83c9353272157893043e9f7d0
│   ├── info
│   └── pack
└── refs
```

FETCHING AN OBJECT

- we can use ‘git cat-file’ to retrieve an object
  - locate using it’s **content address**
  - we only need to give a partial hash

```
$ git cat-file -p 6769dd60
Hello world!
```

- we can also retrieve the object **type** and **length**:

```
$ git cat-file -t 6769dd60
blob

$ git cat-file -s 6769dd60
12
```

FETCHING AN OBJECT IN PYTHON

- let's just try reading in the file…

```
>>> filename = ".git/objects/67/69dd60bdf536a83c9353272157893043e9f7d0"
>>>
>>> contents = open(filename, "rb").read()
>>>
>>> contents
b'x\x01K\xca\xc9OR04b\xf0H\xcd\xc9\xc9W(\xcf/\xcaIQ\x04\x00B\xa8\x06\x80'
```

- …hmm – looks like gibberish!

- not gibberish – it's just compressed:

```
>>> import zlib
>>>
>>> zlib.decompress(contents)
b'blob 12\x00Hello world!'
```

PARSING AN OBJECT IN PYTHON

- given a decompressed object we can easily separate the **header** from the **content**

```
>>> data = b"blob 12\0Hello world!"
>>>
>>> (header, content) = data.split(b"\0", 1)
>>> (header, content)
(b'blob 12', b'Hello world!')
```

- and similarly, extract the object **type** and **length** from the header:

```
>>> (object_type, length) = header.split(b" ", 1)
>>> (object_type, length)
(b'blob', b'12')
```

ENUMERATING OBJECTS IN PYTHON

- but can we see which objects are in the database?

- they are just subfolders of files, so use the standard **glob** library

- glob takes a path pattern, where:
  - "?" means any character
  - "*" means any sequence of characters
  - it returns an array of matching paths

```
>>> import glob
>>>
>>> glob.glob(".git/objects/??/*")
['.git/objects/67/69dd60bdf536a83c9353272157893043e9f7d0']
```

PUTTING IT ALL TOGETHER

- write a little script to:
  - enumerate all the objects
  - load, parse and display them

- function **fetch_object(path)**
  - reads data from disk
  - uncompresses data

- function **parse_object(data):**
  - splits out header metadata
  - returns as dictionary

git-object-dump.py

```python

import glob, zlib, hashlib

def fetch_object(path):
    """Load and decompress an object"""
    compressed = open(path, "rb").read()
    uncompressed = zlib.decompress(compressed)
    return uncompressed

def parse_object(data):
    """Extract type, length and content from an object"""
    (header,content) = data.split(b"\0", 1)
    (object_type, length) = header.split(b" ")
    return {
        "type": object_type,
        "length": int(length),
        "content": content,
    }

for path in glob.glob(".git/objects/??/*"):
    uncompressed = fetch_object(path)
    parsed = parse_object(uncompressed)
    print("------------------------------------------")
    print(path)
    print(parsed)

```

WELL, THAT WAS EASY!

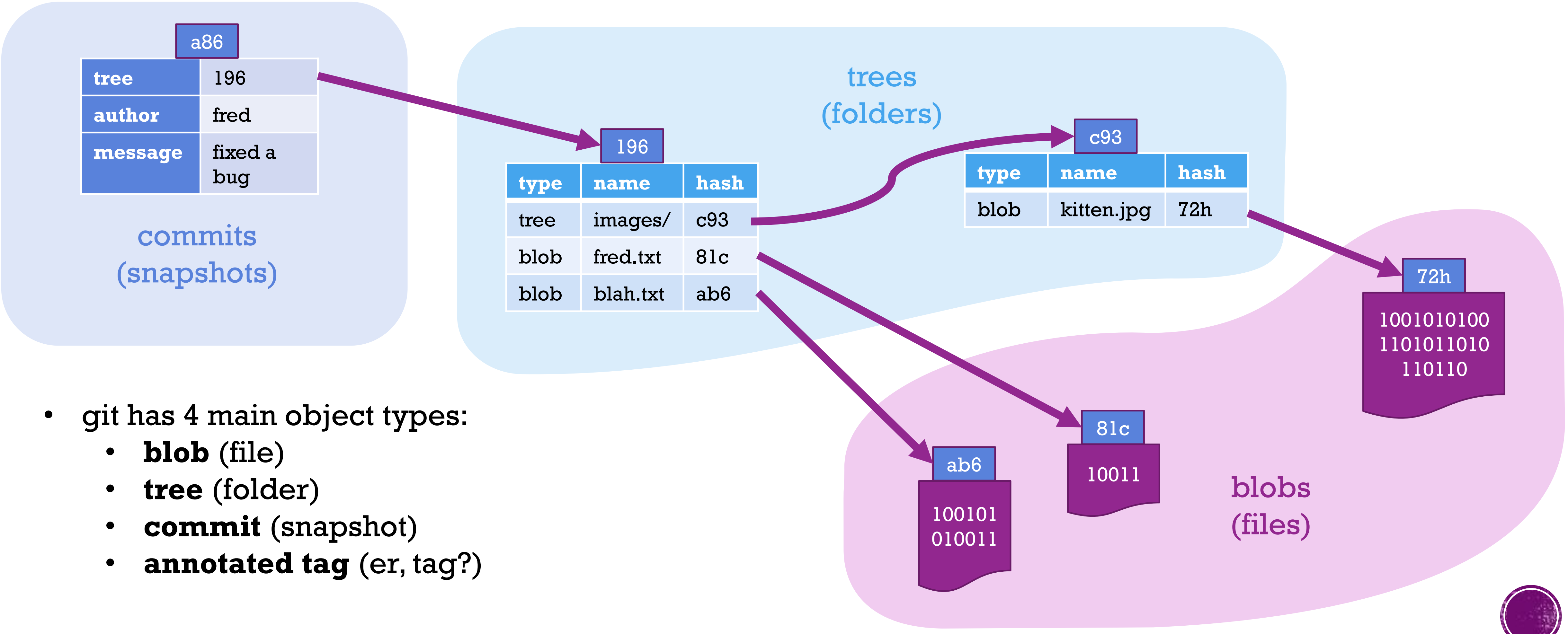
This is testament to:

- the **fluency** and **productivity** of python
  - this would have taken several times longer to write in Java / C# / etc

- the **power** of the python standard library
  - python's "batteries included" approach gives everything you need

- the **simplicity** of git's object database
  - there's not much to it… but we have elided some details:
    - content filtering, 'packed' objects, garbage collection, …

WHERE NEXT? BEYOND BLOBS...



- git has 4 main object types:
  - **blob** (file)
  - **tree** (folder)
  - **commit** (snapshot)
  - **annotated tag** (er, tag?)