**PyGit**

PyGit is an attempt to recreate the core functionalities of Git from scratch using Python. After reading through GitInternals, it’s clear that git’s operations can be broken down into some modules: Take a file and calculates its SHA-1 value, store it as a blob, index a number of blobs to a tree, write the tree, add an author, time, and message to a tree, and write a commit. Keeping this in mind, let’s jump into it.

**Tools required:**

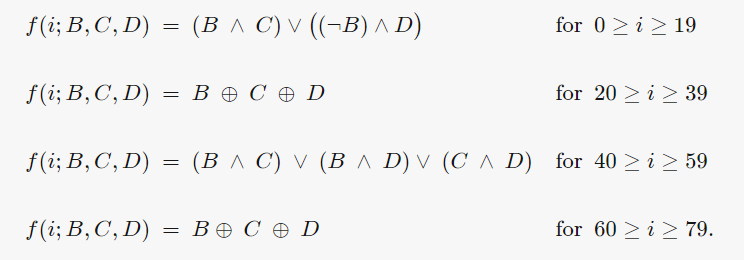
**SHA1 Algorithm:** Secure Hash Algorithm -1 (SHA-1) is a pretty popular cryptographic algorithm. It takes in an input and outputs a 160 bit long hash-value, usually represented as a 40-character long hex value. Also it is designed to be a one-way algorithm.

SHA characteristics: SHA-1 has following 3 important characters:

1. Pre-image resistance: Very hard and time consuming to find the original value m, given the has value h.
2. Second pre-image resistance: Given a message M1, it is very hard to find a new message M2 that hashes to the same value.
3. Collision resistance: Two messages having same hash value, such messages are extremely difficult to find.

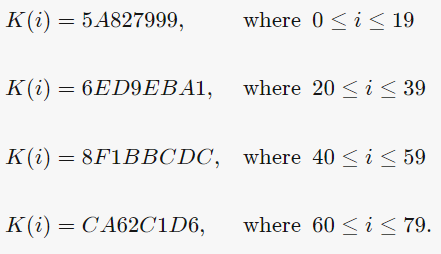
Process:

* Works by inputting a string of length < 264 bits, and outputting a 160-bit hash value.
* Simple method explained here, two methods.
* Take ‘abc’ for example.
* Represent it in binary.
* Generate 5 random strings of hex characters. These are H0, H1…H4.N
* Now, to the original binary message (24 bits), append a 1, and then append 0’s until it reaches 448 bits.
* The length of the message (24) is represented in 64 bits and added to the binary string, bringing its length to 512.
* Let this string be M1.
* The M1 is divided into chunks of 512 bits each (only 1 chunk here).
* Each chunk is divided into 16 32-bit words. W0, W1…W15.
* For each chunk, begin 80 iterations.
* For iterations 16 to 79, where 16 <= I <= 79, perform W(i) = S1(W(i-3) ^ W(i-8) ^ W(i-14) ^ W(i-16)), where ^ represents XOR operation.
* Sn is the left circular shift of a binary string by n-bits.
* Now define the following variables: A = H0, B = H1, C = H2, D = H3, and E = H4.
* Now, for 80 iterations, 0 <= i <= 79, do the following.
* TEMP = S5 \* (A) + f (i; B, C, D) + E + W(i) + K(i)
* Now reassign the variables:
  + E = D
  + D = C
  + C = S30(B)
  + B = A
  + A = TEMP
* Store the value of chunk’s hash to the overall value of all chunks as shown, and proceed to next chunk.
  + H0 = H0 + A
  + H1 = H1 + B
  + H2 = H2 + C
  + H3 = H3 + D
  + H4 = H4 + E
* The f function is a function on B,C,D and is dependent on value of i.



f produces a 32-bit output by taking in 3 32-bit inputs.

* K(i) used is a hex string and is also dependent upon i.



K also returns a 32-bit output.

* The final hash value H would be, H = S128(H0) OR S96(H1) OR S64(H2) OR S32(H3) OR (H4).

For our purpose, we’d use python’s hashlib library. The sha1() of hashlib would compute the SHA1 hash-value for a string, and you can access the digest using the hexdigest(), it converts the 160-bit string to 40 character hex value.

**Bash-compatible shell:** Bash is a command process that runs commands that causes actions. We need a python compatible bash shell, so either you can run it on a UNIX-like environment, or use WSL if on windows.

**Library to parse command-line arguments:** To make the application work on command line interface, we need a library to parse command-line arguments. We will use ’argparse’.

**Argparse:** This library provides functionality to parse command line arguments. It has a couple of different components. It has argument, options, and parameters.

* First create a ArguementParser object. It holds all the necessary information to parse the command line into python data types.
* Adding arguments tells the parser how to take string and turn them into objects.
* parse\_args() method will parse the command line, convert each argument to appropriate data type, and then invoke the appropriate method.
* Subparsers are added to module the commands into different subcommands (git commit, git init etc).

**Configuration file**

Git uses a configuration file format in Microsoft INI format. It is a configuration file for computer software. It consists of key-value pairs for different properties of the software. Different keys are partitioned into different sections denoted by [].

We’ll implement this using Python library ConfigParser.

**Compression**

Git compresses files using zlib, so we’ll use the python library zlib.

**Starting off**

* Create two files, a code file named libpygit.py, and an executable file called pygit.
* Inside the executable file, import the libpygit, and then call the main function.
* Make the file executable by going to your shell and using chmod +x pygit. (Use sudo if permission issues arise).
* Inside the libpygit, import the following libraries:
  + argparse
  + collections
  + configparser
  + hashlib
  + os
  + re
  + sys
  + zlib

**Creating argumentparser**

* Create a argparser using: argparser = argparse.ArguementParser(description = “argument parser”)
* Use subparsers to handle subcommands. argsubparsers = argparser.add\_subparsers(title = “subcommands”, dest = “command”)
* Make subcommands mandatory by argsubparsers.required = True

**Creating main function**

* Define a main function by def main. It takes in all the sys arguments as paarmeters except the first one.
* Run the argparser.parse\_args on argv[1:], and store the result in a variable args.
* Create a switch case/ if else based on the args.command(the subcommand), and reroute each one to a different function.

**Starting with the repository object**

Git works on the concept of a repository, in which a folder is initialized to be a repository. A repository is made up of two things: A work tree (the original folder), and a directory for storing stuff (the .git folder inside the original folder).

* Declare a class Repository.
* It consists of three objects, a worktree, a gitdir, and a config variable. All are None at the beginning.
* Declare an \_\_init\_\_ function, which takes in 2 additional parameters, a path, and a bool force (default = False).
* Assign self.worktree = path.
* Assign self.gitdir = os.path.join(path, ‘.pygit’)
* Check if the given path contains a directory with the name ‘.pygit’, if not raise an exception.
* Check for “config” file insid .pygit folder. If it exists, read it and store it in self.conf. Make sure to initialize self.conf = configparser.ConfigParser(). Else raise an exception.
* Check for version number inside the config file. If it’s not 0, raise an exception.

**Creating the repo\_create function (takes in path as a parameter)**

* Create a repository object by passing path and force = True.
* Check if the worktree of the repo exists. If not, create it. Check if it’s a directory. If not, raise exceptions.
* Create some folders inside the .pygit folder.
  + Branches
  + Objects
  + Refs/tags
  + Refs/heads
* Create a description file inside the .pygit folder.
* Create a HEAD file inside .pygit folder. Write it with “ref: refs/heads/master\n”.
* Write the default configurations inside the config file.
  + Add a section ‘core’.
  + Inside core, set “repositoryformatversion” to be “0”.
  + Set “filemode” to be “false”
  + Set “bare” to be “false”.

**Init Command**

* Create a subparser for the init command.
* Add an argument path to the subparser for init command.
* Create a function cmd\_init which would be called from main and takes args as the parameter.

**Git hash-object and cat-file**

Here, we start off with implementing git objects. There are 4 objects: blob, tree, commit, and tag. The object starts with a header that specifies its type. Followed by a space. Then the size of the object in bytes as an ASCII number. Then a null byte. Then the content of the object.

* First we create a generic object class, named GitObject.
* Each GitObject has a property repo, which is the repository object it is associated to.
* In the \_\_init\_\_ function, pass the repo object as a parameter, and set the repo = repo.
* Also pass a data parameter which is by default None.
* Build two functions, serialize and deserialize, which we would implement later.

Now, we have to build an object\_read function. It takes in a Repository and an SHA value as arguments, and builds an object based on it.

* First assert that the file exists. To do this, check for a folder based on first two characters of the SHA value, and the filename is the rest 38 characters.
* Open and read the file, decompress the content using zlib, and store it in a raw variable.
* Find the space inside the raw string.
* The object type is the string from 0 to space.
* Find the null character inside the raw string. The string from space to null is the size.
* Check if the size is equal to the length of the rest of the string.
* Return an object of the type found in the start.

Now, for object\_write function to write objects. It takes an obj and a bool actually\_write as parameters.

* Extract data by data = obj.serialize()
* Compute result by result = obj.fmt + b’ ‘ + str(len(data)).encode() + b’x\00’ + data
* Hash it using SHA-1 and store its hexxdigest to a variable sha.
* If actually\_write is True, make a directory using first two characters of hash value, inside it, make a file named last 38 characters of hash value. Inside the file, store the result by compressing it using zlib.
* Return sha.

**Implementing Blob Class**

We start with implementing blob, the simplest object.

* A object fmt, which is equal to ‘blob’ here.
* A serialize function, which returns self.blobdata
* A deserialize function, which sets self.blobdata = data.

**Implementing Cat-file**

* Add a subparser for cat-file.
* Add argument “type” and “object”.
* Make a bridge function cmd\_cat\_file to be called from main.
* Inside it, find the repo using repo\_find.
* Call a cat\_file function using repo, object, and type as parameters.

Now implementing cat\_file:

* Use object\_read to get an object.
* Write the object into the buffer using object.serialize.

**Implementing hash-object**

* Add a subparser for hash-object.
* Add arguments -t and -w for type and write. Also a path argument for the file path.

Implement a cmd\_hash\_obejct to be called from main:

* If the write argument is given, find the current repo.
* Else leave the repo object as None.
* Create an object\_hash function, which takes in a file, object type, and a repo as parameters.
* Unpack data from the file. And depending on the object type, create an object of that type.
* Write the object using object\_write function and return its sha value.
* Call this function inside cmd\_hash\_object.

**Commit History**

A commit object is a snapshot of a tree, along with additional data. Let’s take a look at the structure of a commit object.

* Tree: A reference to a tree object.
* Parent: A reference to the parent of this commit. This is another commit.
* Author
* Committer
* gpgsig: PGP signature of the object

We start by building a parser for the object type:

* The function takes in raw string, start index, and a dictionary as input.
* If dict is not defined, define it.
* Find the index of the first space
* Find the index of the first \n
* If there is no space or the \n appears first, set the b’ ‘ to the string from start to end, and return the dict.
* Else, store a key as the string from start to space.
* Loop until you find a line where \n is not followed by the space.
* Store the string from space to end in value variable, and remove the spaces after \n.
* If the key exists in the dictionary, append the value to the list attached to key.
* Else set an entry to the dict.
* Recursively call a function using start = end + 1