**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:**

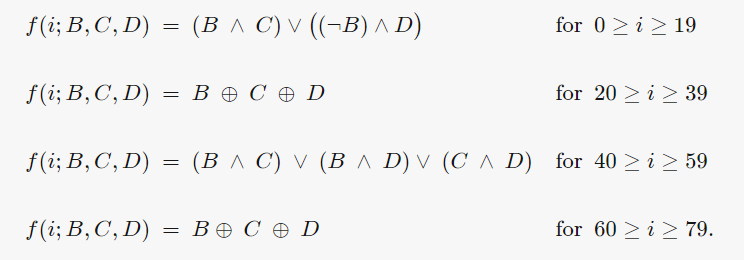
**SHA1 Algorithm:** Secure Hash Algorithm -1 (SHA-1) is a pretty popular cryptographic algorithm. It takes in an input and outputs a 160 bit long hash-value, usually represented as a 40-character long hex value. Also it is designed to be a one-way algorithm.

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.

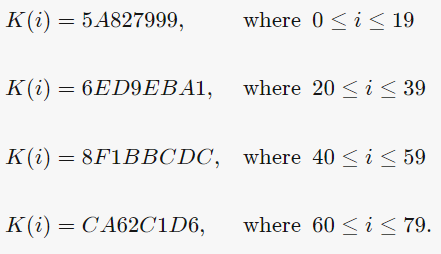
Process:

* 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).
* All the functionalities would be implemented inside src folder, and all the functions would be called to libpygit.py as a wrapper.

**File Utilities**

* Built inside src/fil\_utilities.py
* read\_file function takes in a path parameter, opens the data present in the file, and returns the data in binary form.
* write\_file function takes in a path and data parameter, opens the file present at the path in binary form, and writes the data.

**Hash Object**

* hash\_object takes in data, obj\_type, and a bool write as inputs.
* The header is a binary string. The first word is the file type, followed by a space, and then includes the length of the data.
* The full data is the header appended with a null character and the data string.
* Find the sha1 value of the full data using hashlib.sha1. Convert it to hexdigest.
* If write bool is true, open a file in the path. Path is inside the .pygit/objects/sha1[:2]/sha1[2:].
* Inside the file, dump the full data after compressing it with zlib.

**Read and Find Object**

* Built inside the object\_utilities.py.
* find\_obj takes in a sha1 prefix and finds an object using it.
* Check if the length of sha1 prefix > 2.
* Go inside the .pygit/objects/sha1[:2] directory. If it does not exist, return an exception.
* Maintain rest = sha1[2:].
* Cycle through the files inside the path. If any of them starts with the rest, appends it to an objects list.
* If the length of the objects list is 0, return an error saying ‘no objects found’.
* If the length of the objects list is > 1, return an error saying multiple files found.
* Else, return the path to the object file found.
* read\_obj also takes in an sha1-prefix and returns the object type as well as the data present inside.
* First call the find\_obj using the sha1-prefix, store its return value in a path
* Extract full data using read\_file(path) and decompressing it.
* Find the null character index in the full data.
* Extract header as full\_data[:null\_index].
* Extract obj\_type and obj\_size by splitting the header by the space.
* Extract data as full data [null\_index + 1:].
* Check if the obj\_size == len(data).
* Return obj\_type and the data.

**Index**

Now let’s start with the index part of the pygit. Indexing is one of the most important components of the pygit functionality.

**Structure of the index object.**