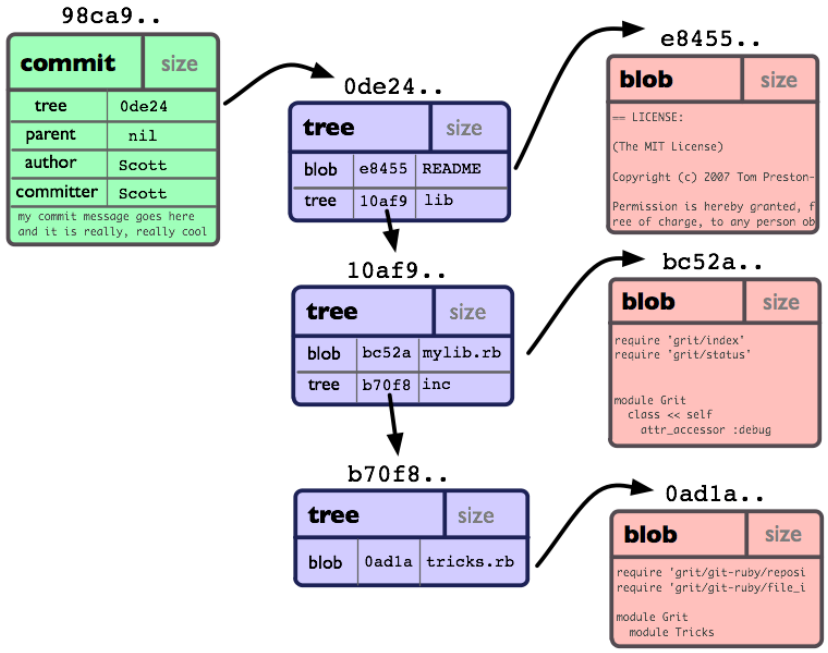
Language Comparisons

Feature	C	Python	JavaScript
Type System	Static typing	Dynamic typing	Dynamic typing
Syntax	C-like syntax	Indentation-based	C-like syntax
Execution	Compiled	Interpreted	Interpreted (or Just-In-Time compiled)
Memory Management	Manual memory management	Automatic memory management	Automatic memory management (Garbage collection)
Platform	Cross-platform	Cross-platform	Browser-based, server-side (Node.js)
Concurrency	Limited support for concurrency	Threading, multiprocessing	Event-driven, asynchronous I/O
Usage	Systems programming, embedded systems	General-purpose programming, scripting	Web development, client-side scripting
Libraries	Standard library + external libraries	Rich standard library + extensive third-party libraries	Extensive ecosystem of libraries, frameworks (e.g., React, Express)
Typical Applications	Operating systems, drivers, embedded systems	Web development, automation, data analysis, scientific computing	Web development, server-side scripting, browser extensions

Git Objects

Object Types

blobs	Contents of a file, sequence of bytes
trees	Directory structure of a project at a particular snapshot References to blobs and other trees
commit	Object metadata, author, committer, timestamp, commit message Reference to top level tree → creates a tree



Blob Objects

- Blobs are unique
- Blobs are stored in the object directory of .git as a subdirectory starting with the first two chars of the hash
- Git stores the compressed data in a blob, along with metadata in the header
    - identifer `blob`
    - size of content
    - /0 delimiter
    - content
- Git hash object
    - asking for the SHA1 of contents ... git hash-object --stdin
    - generating the SHA1 of the contents, with metadata ... openssl sha1
    - when running a cache method the same result is always achieved

- Contents are just a sequence of bytes → git does not care

```
$ echo “green” | git hash-object --stdin -w
: <HASH>
```

- Creates a hash from standard input, or whatever the content is

`$ git cat-file -t <HASH>` Prints out type that <HASH> refers to
`: blob`

`$ git cat-file -p <HASH>` Prints out contents of <HASH>
`: “green”`

`$ git cat-file -s <HASH>` Prints out size of contents that <HASH> refers to
`: 12`

NOTE: Simple repo has 1 tree, 1 blob, and 1 commit. Object content (blob, tree, commit) is compressed and can not be read with a simple cat command, instead git cat-file.

Blob Contents

`blob 20øArma virumque can`

Type, length, null byte, contents, 1 byte for each character

BUT

`$ less .git/objects/24/b234a8s89d8j3s08djf8`

→ gets compressed (See compression)

Tree Objects

- info on filenames and directory structures are stored in a tree
- trees contain pointers (using SHA1) to blobs and other trees (directed graph)
- trees also contains metadata
    - type of pointer (blob or tree
    - filename or directory name
    - mode (executable file, symbolic link, etc.)
- identical content is only stored once -> saving tons of space on hd

Like a dictionary in a linux file system

- mapping from file names to files
- mapping “file” name to objects

git cat-file -p master^{tree}

- You can have two different names mapped to the same ID
- Like having hard links

ZONED OUT AT THIS POINT → he drew a diagram at this point
Collision free hash function is KEY

Cachinfo

`$ git update-index –add –cacheinfo 100644`

100644 because `-rw-r--r--` → chmod 644 file

```
011100100
 6  4  4
```

```
$ git cat-file -p 2h39li08n9ub9ub9un
```

Commit Objects

- Results of a git commit command
- Contents
  Idenitifier that its a commit
  Pointer to a tree
  Authorship timestamp, commit timestamp
  Author name, commiters name
  Commit message

Create a commit
```
$ git commit-tree TREE_OBJECT COMMIT_MESSAGE
```

Git model – do not change objects

- If it looks like its changing contents, its actually editing new copies of objects

3 Trees

Working tree

- `git init`
- Current files you are working on

Index(staging) tree

- `git write-tree` → converts staging area into a tree
- anytime you're updating index, you're putting into the index tree
- `git update-index` → putting files into the index tree
- *`git add`* uses *`git write-tree`* and *`git update-index`*

Commit tree

- last commit
- Identifies which tree you want to commit
- `git commit-tree TREE_OBJECT COMMIT_MESSAGE`
- `git commit -m`

→ everything combined into git commit -m

```
$ git update-index --add --cacheinfo 100644
HASH_TO_BLOB_OBJECT first.txt
```
//links the file to the object
→ puts it into a staging area

```
$ git write-tree
: HASH_TO_TREE_OBJECT
```
→ converts the staging area into a tree, which can then be committed

```
$ git cat-file -t HASH_TO_TREE_OBJECT
: tree
$ git cat-file -p HASH_TO_TREE_OBJECT
: 100644 blob HASH_TO_BLOB_OBJECT first.txt

$ echo “Commit message” | git commit-tree HASH_TO_TREE_OBJECT
: HASH_TO_COMMIT_OBJECT
```

- Passes in an index-tree commit, and adds it to the commit tree

```
$ git log HASH_TO_COMMIT_OBJECT
: commit HASH_TO_COMMIT_OBJECT
  Author: Zach Fischer <zfischer42@gmail.com>
  Date:   Fri Feb 23 14:30:06 2024 -0800
     “Commit message”
```

References

.git/refs

`$ git update-ref refs/heads/main <commit-hash>`

- Updates the reference `refs/heads/main` to point to `<commit-hash>`
- Effectively moves the branch pointer to the specified commit, allowing you to rewrite history or reset the branch to a different commit

`$ git symbolic-ref HEAD`

- Outputs the full ref name (e.g., refs/heads/main) that HEAD is pointing to

`$ git symbolic-ref HEAD refs/heads/new-branch`

- Creates or updates the symbolic reference of HEAD to point to the specified branch (refs/heads/new-branch).
- Effectively changes the currently checked out branch to new-branch.

`$ git checkout -b branch`

- Anytime you want to make a branch, you point the head to the branch
- Achieved by using references
- HEAD always points to the current branch
- update-ref → symbolic ref
- Looks at the last commit on the commit tree → when you check out its based off of the branch you’re currently on

Tags

- Lightweight pointer to a commit, does not move with branch

```
$ git update-ref refs/tags/v1.0 <commit-hash>
```

Annotated Tags:

```
$ git tag -a v1.1 1b774a1d8764e8e95b0c069cbe87a76856038f3b -m “Tag v1”
$ cat .git/refs/tags/v1.1
: a80f48a63305a3545022353b701c3149158037aa
```

→ gives new taghash

```
$ git cat-file -p a80f48a63305a3545022353b701c3149158037aa
: type commit
  tag v1.1
  tagger Zach Fischer <zfischer42@gmail.com> 1708728379 -0800

  Tag v1
```

Remotes

```
$ git remote add origin {url}     // specific repo url
$ git push origin master          // push to master branch in url

$ cat .git/refs/remotes/origin/master
```

Zlib - Decompression

zlib is a software library used for data compression
It provides functions for compression and decompression of data using the zlib format

Hash functions are already compressed, you need to decompress them

python3

```
>>> import zlib
>>> commit_file=open("1b/774a1d8764e8e95b0c069cbe87a76856038f3b",
"rb")
```

→ read binary file

```
>>> print(zlib.decompress(commit_file.read()))
```

b'commit 191\x00tree a0c06ee2fbdcd389fea0c2cf65d39d419ee0476c\nauthor Zach Fischer <zfischer42@gmail.com> 1708727406 -0800\ncommitter Zach Fischer <zfischer42@gmail.com> 1708727406 -0800\n\n\xe2\x80\x9cCommit message\xe2\x80\x9d\n'

Assignment 5 (Git repository organization)

Topological Sort

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove any node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if length of L < number of nodes in graph then
    return error   (graph has at least one cycle)
else
    return L       (a topologically sorted order)
```

```
"""
Use khans algorithm  to create a topological sort
"""

def topological_sort(graph, root_commits):
   result = []
   queue = deque()

   # Add all the root nodes to the queue as starting points for the ordering
   for root in root_commits:
       queue.append(root)

   while queue:
       node = queue.popleft()
       result.append(node)

       # Sort the children of the current node
       children = sorted(graph[node].children)

       for child in children:
           graph[child].parents.remove(node)

           # If no parents, add child to the queue
           if len(graph[child].parents) == 0:
               queue.append(child)

   # Return the result in reversed order
   return result[::-1]
```

```
"""
Finds the path to the .git directory
Can be called from any subdirectory
Exits if no .git directory is found
"""
def discover_git_directory():
   current_path = os.getcwd()

   while True:
       git_path = os.path.join(current_path, '.git')
       if os.path.isdir(git_path):
           return git_path  # Return the path if .git directory is found

       parent_path = os.path.dirname(current_path)
       if parent_path == current_path:
           # NO .GIT FOUND
           print("Not inside a Git repository", file=sys.stderr)
           sys.exit(1)  # Exit with status 1
       else:
           # Move up to the parent directory and continue the search
           current_path = parent_path
   pass
```

```
"""
Creates a dictionary of branches, where the key is the branch
name and the value is the list of commits.
"""

def get_branches():
   git_dir = discover_git_directory()
   refs_dir = os.path.join(git_dir, 'refs/heads')
   branches = dict()
   #print(refs_dir)

   for (root, dirs, files) in os.walk(refs_dir):
       for filename in files:

           branch = os.path.join(root, filename)
           branch_name = os.path.relpath(os.path.join(root, filename), start=refs_dir)
           #print(branch_name)
           commit = (open(branch).read()).strip()
           branches[branch_name] = commit

   #print(branches)
   return branches
```

```
"""
Build the commit graph by traversing the root nodes and creating or populating
nodes when necessary
"""
def build_commit_graph():
   branch  = get_branches()
   git_dir = discover_git_directory()
   graph = dict()
   stack = []
   root_commits = set()

   # Traverse commits by following each branch
   for b, commit in branch.items():
       # Add branch to
       stack.append(commit)

       while stack:
           current_commit = stack.pop()
           commit_node = CommitNode(current_commit)

           # Get git object
           with open(os.path.join(git_dir, "objects", commit_node.commit_hash[:2],
                     commit_node.commit_hash[2:]), "rb") as f:
               content = zlib.decompress(f.read()).decode().split('\n')
               #parents = [line.split(' ')[1] for line in content if
                           line.startswith('parent')]
               #message = content[5]
           has_parents = False
           for string in content:
               if string.startswith("parent"):
                   parent = string[7:]
                   # Add parent hash to parent list
                   commit_node.parents.add(parent)
                   stack.append(parent)
                   has_parents = True

           if has_parents == False:
               # No parents means root commit
               root_commits.add(current_commit)

           # If the commit has not been stored already, add it to the dictionary
           if current_commit in graph:
               pass
           else:
               graph[current_commit] = commit_node
   # Add the children to all the nodes in graph
   for commit, node in graph.items():
       parents = node.parents
       for parent in parents:
           graph[parent].children.add(commit)
   #print(graph)
   return graph, root_commits
```

```
        switch(c) {
            case 'i':
                // printf("test");
                if (strcmp("rdrand", optarg) == 0) {
                    opt->input = RDRAND;
                } else if (strcmp("lrand48_r", optarg) == 0) {
                    /* Uses mrand implementation but I'm keeping
                       the "lrand48_r" input as specified in the spec */
                    opt->input = MRAND48_R;
                } else if ('/' == optarg[0]) {
                    opt->input = SLASH_F;
                    opt->r_src = optarg;
                } else {
                    fprintf(stderr, "ERR Valid args needed for -i: rdrand or lrand\n");
                    exit(1);
                }
                opt->valid = true;
                break;
            case 'o':
                if (strcmp("stdio", optarg) == 0) {
                    opt->output = STDIO;
                } else {
                    opt->output = NUM;
                    opt->block_size = (atoi(optarg));

                    if (opt->block_size == 0){
                        fprintf(stderr, "Err: Valid block size needed for opt -o N\n");
                        exit(1);
                    }

                    if (optind >= argc) {
                        fprintf(stderr, "ERROR: Option -o requires an operand N\n");
                        exit(1);
                    }
                }
                opt->valid = true;
                break;
            default:
                fprintf (stderr, "ERROR: invalid arguments");
                exit(1);
                break;
        }
    }

    /* Assign number of bytes to randomize */
    opt->nbytes = atol(argv[optind]);
    if (opt->nbytes >= 0) {
        opt->valid = true;
        if (opt->output == STDIO) {
            opt->block_size = opt->nbytes;
        }
    }
}
```

output.c (putchar)

```
// Write specified bytes to std output
bool
writeBytes (unsigned long long x, int nbytes)
{
   do {
     if (putchar (x) < 0)
        return false;
     x >>= CHAR_BIT;
     nbytes--;
   }
   while (0 < nbytes);

 return true;
}
```

Byte Output (malloc + write)

```
if(opt.output == STDIO){
   // Standard Output - user writeBytes to print each random byte
   do {
       unsigned long long x = rand64();
       int outbytes = nbytes < wordsize ? nbytes : wordsize;
       if (!writeBytes (x, outbytes))
       {
         output_errno = errno;
         break;
       }
         nbytes -= outbytes;
   } while (0 < nbytes);
 }
 else {
   // N block output - write each character to a buffer
   char * obuffer = (char *) malloc(opt.block_size);
   // Loop through all bytes in nbytes
   do {
     // Set outbytes to appropriate number of bytes (up to block_size)
     int outbytes = nbytes < opt.block_size ? nbytes : opt.block_size;
     for (int i = 0; i < outbytes; i++) {
       obuffer[i] = rand64();
       // Most of the rand64() data is unused but oh well
     }

     // Write the bytes from the buffer to standard output
     int status = write(1, obuffer, outbytes);
     if (status < 0){
       fprintf(stderr, "Write failed");
     }

     nbytes -= outbytes;
   } while (nbytes > 0);
   free(obuffer);
 }
```

malloc()

```
// Allocate memory for a buffer of size 10 bytes
   int size = 10;
   char *buffer = (char *)malloc(size * sizeof(char));

// Check if memory allocation was successful
   if (buffer == NULL) {
       fprintf(stderr, "Memory allocation failed\n");
       return 1; // Exit with error
   }
```

write()

```
// Write some data to the buffer
   const char *data = "Hello";
   int dataLength = strlen(data);
   if (dataLength < size) {
       strcpy(buffer, data); // Copy data to the buffer
   } else {
       fprintf(stderr, "Data is too large to fit into the buffer\n");
       free(buffer); // Free allocated memory
       return 1; // Exit with error
   }

// Print the content of the buffer
   printf("Content of the buffer: %s\n", buffer);

// Free allocated memory
   free(buffer);
```

mrand64()

```
/* IMPLEMENT MRAND Randomness Generation */
long int higher, lower;

/* Initialize the hardware mrand48 implementation. */
void hardware_mrand48_init (void) { }

/* Return a random value, using hardware operations.  */
unsigned long long
hardware_mrand48 (void)
{
   higher = mrand48();
   lower = mrand48();
   //bitwise shift of 32
   return ((((unsigned long long) higher) << 32) | ((unsigned long long) lower));
}

/* Finalize the hardware mrand48_r implementation.  */
void hardware_mrand48_fini (void) { }
```