Git Outta Here!

Christopher R. Genovese

Department of Statistics & Data Science

02 Sep 2025
Session #3

Plan

Binary Search Debrief

Plan

Binary Search Debrief

The Command Line

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

Announcements

- Office Hours Thursday 2pm and by appointment. I encourage you to ask me questions.
- **Reading**:
  - The Shell `https://36-750.github.io/tools/shell/`
  - `https://clig.dev/`
  - Writing Commands on home page
  - Git from the Bottom Up by John Wiegley
  - `https://learngitbranching.js.org/`
  - Finish Thinking Languages Part 1 if not already
  - Try out the `git-challenge` exercise in the `problem-bank` repo
- **Homework**:
  - **swag** assignment due Tue 9 Sep. Available on Canvas and github problem bank.

Questions?

Goals

Last week, we set up the idea of types as a tool for thinking, and we saw a strong mapping between logic and types.

We started a story using binary search as a driver. The goal is several deep lessons about proofs, search algorithms, and design.

We will return to continue that story soon, after some time to digest what we've seen so far.

But first, today, we will pull on a very different thread.

Our goals for today

- Debrief on Binary Search coding
- Practical guidance on two key tools: the command line and git.
- Discussion of objects as a programming paradigm
- Introduction to migit

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

Binary Search Debrief

Target the third formulation:

- We have *strictly increasing* $f : \mathbb{N} \longrightarrow \mathbb{N}$.
- We have a target value $t \in \mathbb{N}$.
- We want to find an $x \in \mathbb{N}$ such that $t = f(x)$.
- Alternatively, we want to find the smallest $x \in \mathbb{N}$ such that $t \leq f(x)$ and whether this is an equality.

Note: for any $x \in \mathbb{N}$, $x < f(x + 1)$.

So, $t \leq f(t) < f(t + 1)$ and we can restrict our attention to $x \in [0\,..\,t]$

Our basic approach is to test the midpoint and successively split the interval accordingly.

Binary Search Debrief

Here are simple (but not quite right) versions, in TL2 and TL1. Assessment?

```
function search(f, t):
    lo, hi = 0, t

    while lo <= hi:
        m = (lo + hi) div 2
        f_m = f(m)

        if t < f_m:
            hi = m - 1
        else if t == f_m:
            return m
        else:
            lo = m + 1
    return None
```

```
search : (Nat -> Nat) -> Nat -> Maybe Nat
search f t = bisect 0 t
  where
    bisect lo hi
      | lo > hi       = None
      | t < f_m       = bisect lo (m - 1)
      | t == f_m      = Some m
      | otherwise     = bisect (m + 1) hi
      where m = (lo + hi) `div` 2
            f_m = f m
```

Binary Search Debrief

Here are simple (but not quite right) versions, in TL2 and TL1. Assessment?

```
function search(f, t):
    function bisect(lo, hi):
        if lo > hi:
            return None

        m = (lo + hi) div 2
        f_m = f(m)

        if t < f_m:
            return bisect(lo, m - 1)
        else if t == f_m:
            return m
        else:
            return bisect(m + 1, hi)

    return bisect(0, t)
```

```
search : (Nat -> Nat) -> Nat -> Maybe Nat
search f t = bisect 0 t
  where
    bisect lo hi
      | lo > hi       = None
      | t < f_m       = bisect lo (m - 1)
      | t == f_m      = Some m
      | otherwise     = bisect (m + 1) hi
      where m = (lo + hi) `div` 2
            f_m = f m
```

Binary Search Debrief (cont'd)

What happens when $f(n) = 2^n$?

As this case shows, it is useful to *bound* the target before the search. That is, we first find integers a, b such that $f(a) < t \leq f(b)$.

If needed, i.e., $t \leq f(0)$, we can set $(a, f(a))$ to be $(-1, -1)$ as a ***sentinel***. Otherwise, it suffices to find $k \in [1\,..)$ such that $f(2^{k-1}) < t \leq f(2^k)$. This must exist and requires at most logarithmic steps to find.

```
function bound(f, t):
    if t <= f(0):
        return (-1, 0)

    p = 1
    while t > f(p):
        p *= 2

    return (p div 2, p)
```

```
bound : (Nat -> Nat) -> Nat -> (Int, Nat)
bound f t = if t <= f 0
              then (-1, 0)
              else (p `div` 2, p)
  where
    p = while (\x -> t > f x) (* 2) 1
```

Binary Search Debrief (cont'd)

Now we adjust `search` to focus on the interval $(a\,..\,b]$ and find the smallest x such that $t \leq f(x)$. We know already that $t \leq f(b)$, so such an x exists. (Notice the change of type.)

```
function smallest(f, t, lo_hi):
    a, b = lo_hi

    while a + 1 < b:
        m = (a + b) div 2
        if t <= f(m):
            b = m
        else:
            a = m
    return b

function search(f, t):
    x = smallest(f, t, bound(f, t))
    return (x, f(x) == t)
```

```
smallest : (Nat -> Nat) -> Nat -> (Int, Nat) -> Nat
smallest f t (a, b)
  | a + 1 >= b = b
  | t <= f m   = smallest f t (a, m)
  | otherwise  = smallest f t (m, b)
  where m = (a + b) `div` 2

search : (Nat -> Nat) -> Nat -> Either Nat Nat
search f t = let x = smallest f t (bound f t)
             in
               if f x == t
                 then Right x
                 else Left x
```

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

Why Use the Command Line?

Some disadvantages:

- It has a nontrivial learning curve.
- There are many commands and configuration options, many not particularly mnemonic.
- Many commands focus on processing text files.

But this approach has many compensating advantages:

- Operations may be easily combined to form new and useful operations.
- Very complex commands can be built from several simple pieces.
- Commands can be recorded for later recollection, reuse, modification, or automation.
- Commands can be parameterized for more general uses.
- Once mastered, working on the command line is **fast**.

Shells, Terminals, Files, Commands

- What is a shell?
- Terminals, ttys, and terminal emulators
- Files and directories

Principles

1. Design for Composability
2. Design for Humans
3. Make Consistent Interfaces
4. Effective feedback
5. Provide great help
6. Make your Interface Discoverable
7. Handle errors clearly
8. Decoration is wonderful (when appropriate)

Running Programs on the Command Line

The command line:

- Prompt (from the shell)
- Command name
- Flags and arguments
- Subcommands
- Pipes and redirection

Examples to look at: jq, git, curl, brew, ls, ...

Input and Output

- Standard input, standard output, standard error
- Redirection
- Pipes and composability
- Text (line-based) and JSON
- The necessary pain of handling I/O errors

Program Interface Conventions

https://clig.dev/ gives a nice overview, especially see Arguments and flags section.

Always include:

- `--help` and `-h`
- `--version` (and `-v` or `-V`

Provide usage string with no arguments (if some required). `--help` can accept arguments for specialized documentation.

Where appropriate:

- `--quiet`
- `--verbose`
- `--dry-run`
- `--plain`
- `--no-input`
- `--no-color`
- `--json`

Other commonly used arguments:

- `--force`
- `--debug`
- `--output`

Program Interface Conventions

https://clig.dev/ gives a nice overview, especially see Arguments and flags section.

Other considerations:

- For file input, read from stdin if none
- Default output to stdout, messages to stderr.
- Keep output human readable. Fancy output and ttys.
- For file I/O, support '-' for stdin/stdout.
- Prefer flags to arguments except for uniform operations.
- For flags with optional arguments, have a special value for not supplied.
- Prompt for user input where appropriate but do not ever *require* a prompt.
- Confirm before executing a dangerous action (if possible).
- Check user input asap.

Handling Command-Line Arguments

- Multi-language: docopt
- Python: https://docs.python.org/3/library/argparse.html and click
- R: argparse and optigrab

Some very nice examples in other languages show some of the possiblities and are interesting to study.

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

Some Benefits of Version Control

- Keep a complete record of changes (and when and by whom the changes were made). You can always revert back to a previous state and even continue forward development from a past state.
- Store a message with every change, so the rationale for changes is always recorded.
- Maintain multiple parallel branches of development, merging or discarding them as desired.
- View differences across time and identify the source of changes/problems across your history.
- Mark important snapshots of your code: "the version submitted to JASA" or "the version used for my conference presentation".
- Easily distribute your code or back it up, e.g., with a Git hosting service like GitHub or Bitbucket.

Git

Git is a popular, distributed version control system. It's underlying design is conceptually lean and elegant; it's practical complicated, like an overgrown garden (or a dumpster fire). High-Level Ideas:

1. Every object in git has a *unique identifier* (hash) that derived from its content.
2. A **commit** is an *immutable snapshot* of your project at specific time. Each commit is equipped with a pointer to its predecessor(s) (the state of HEAD at the time the commit is created).

   The resulting chains of commits creates a "history" for the project.
3. A **branch** is a mutable *named pointer* to commit (and thus its linked history).

   Branches are not special, except when a commit is made "on a branch," the pointer is advanced to the new commit.
4. A **tag** is an *immutable* named pointer to a commit with its own description.

To use git effectively, it is surprisingly helpful to understand the underlying concepts.

Key Concept: The Three Trees

A git repository is a collection of commits and tags, along with a set of branches that point to the tips of the history along one or more several lines of development.

In the background, git manipulates four different types of objects: blobs, trees, commits, and tags. All the objects are stored under `.git/objects` in your project.

These objects are arranged into high-level "trees" that describe the state of your project at a given moment. There are three high-level trees that structure the git workflow.

1. HEAD – the latest commit on the current branch
2. Staging Area (Index) – a tree forming a *proposed next commit*, a set of changes from HEAD that are currently registered to be added to the next commit.
3. Working tree – the current (possibly modified) file tree for your project.

When you edit your code or add or remove files, you are changing the working tree. You then stage selected changes into the Index, building up a new commit. Eventually, you commit those staged changes, which creates a new commit with HEAD as its predecessor and advances HEAD to point to it.

Key Concept: The Three Trees (cont'd)

Here's a stylized view of HEAD in a fictitious repository.

```
[MyProject]      |  100000 MyProject    tree [100001, 100002, 100003, 100006,
                                                100007, 100010, 100014]
  README         |  100001 README       blob ad6b32ed752dac56d8dc7559308195ec
  core.py        |  100002 core.py      blob e25e6b0e7c7bf92d4e861734436c2fd2
  [data]         |  100003 data         tree [100004, 100005]
    maps.gis     |  100004 map.gis      blob 239ccf3e9b7c400c09946af1581dd711
    config.json  |  100005 config.json  blob d383ccd77254e01ea22a55313e94ad14
  gis-util.py    |  100006 gis-util.py  blob abb6712b4ca6d1ba7a2ee6f4a1642c8e
  [tests]        |  100007 tests        tree [100008, 100009]
    test-gis.py  |  100008 test-gis.py  blob e488f57b8c424702898b776d195f2f1c
    test-gfx.py  |  100009 test-gfx.py  blob 731ea463f7b3c9632b32ce57d9805375
  [doc]          |  100010 doc          tree [100011]
    [generated]  |  100011 generated    tree [100012, 100013]
      0AF323DB1  |  100012 0AF323DB1    blob 726232cd6702e61932fbf3ea44a03c6f
      BFEEC9978  |  100013 BFEEC9978    blob d792cf03a1af84fd546054caee49613c
    doc.md       |  100014 doc.md       blob cee77a23ad50c9e419bf77e835f36aed
...              |  100015 ...          ...  ...
```

On the left is high-level file-system view of the tree; on the right a more packed encoding like what git uses. (Ids in trees are shorthand for hashes.)

Key Concept: The Three Trees (cont'd)

This gives an associative map of hashes to contents. We do not need file names for this to work; names become *metadata*.

This tree also has a hash, so we end up with a snapshot, a single commit:

```
100000
  HEAD, main
```

HEAD and main are named pointers to this commit – branches.

Key Concept: The Three Trees (cont'd)

Now we make a change to one file README. What happens?

```
100000 MyProject    tree    [100001, 100002, 100003, 100006,
                             100007, 100010, 100014]
100001 README       blob    ad6b32ed752dac56d8dc7559308195ec
100002 core.py      blob    e25e6b0e7c7bf92d4e861734436c2fd2
100003 data         tree    [100004, 100005]
100004 map.gis      blob    239ccf3e9b7c400c09946af1581dd711
100005 config.json  blob    d383ccd77254e01ea22a55313e94ad14
100006 gis-util.py  blob    abb6712b4ca6d1ba7a2ee6f4a1642c8e
100007 tests        tree    [100008, 100009]
100008 test-gis.py  blob    e488f57b8c424702898b776d195f2f1c
100009 test-gfx.py  blob    731ea463f7b3c9632b32ce57d9805375
100010 doc          tree    [100011]
100011 generated    tree    [100012, 100013]
100012 0AF323DB1    blob    726232cd6702e61932fbf3ea44a03c6f
100013 BFEEC9978    blob    d792cf03a1af84fd546054caee49613c
100014 doc.md       blob    cee77a23ad50c9e419bf77e835f36aed
100015 ...          ...     ...
101033 README'      blob    a1d09363bc48b91ecf2b2bae8a0a533b
101034 MyProject'   tree    [100133, 100002, 100003, 100006,
                             100007, 100010, 100014]
```

Key Concept: The Three Trees (cont'd)

Now we make a change to one file README. What happens?

Now keys 101034 and 100000 point to two different versions in our history.
And do we have two snapshots in a sequence: two commits in a history.

```
100000          <---  101034
                          HEAD, main
```

Branches HEAD and main have automatically advanced to the tip of the history.

Key Concept: The Three Trees (cont'd)

We make another change, regenerating the files in doc.

```
00242 0AF323DB1'   blob    1dcab182eb803c7f3e4459c69f20c5f0
00243 BFEEC9978'   blob    abb0acf24574303090c3455c71bbf94c
00244 generated    tree    [100242, 100243]
00250 doc'         tree    [100244]
02388 MyProject''  tree    [100133, 100002, 100003, 100006,
                            100007, 100250, 100014]
```

This yields a chain of three commits that gives a history

```
100000 <--- 101034 <--- 102388
                         HEAD, main
```

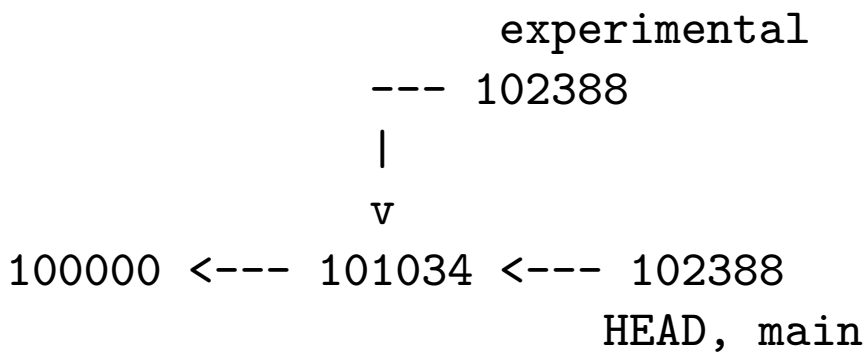
Again branches HEAD and main have advanced to stay at the tip of the history. (We can explicitly move them as well.)

Key Concept: The Three Trees (cont'd)

Thinking about it, we realize that we want to try something out starting at commit 101034. So we "move to that snapshot" and make a change: adding a new file `packages` inside the `MyProject` directory.

```
02521 packages     blob    e26fe3c3348c976e937b81423b561ebe
02388 MyProject''' tree    [100133, 100002, 100003, 100006, 100007,
                            100010, 100014, 102521]
```

We call this branch `experimental`, and our history now looks like



Key Concept: The Three Trees (cont'd)

Staging: Working Tree → Index

The **working tree** is the version of the files that we see in our editor. We can make what changes we like when we are working without changing the other trees.

When we have made changes that we think we want to save, we *first* add them to the **Index**. This is a tree that starts as a copy of the latest commit but is amended by changes **staged** from the working tree.

Key Concept: The Three Trees (cont'd)

Committing: Index → HEAD

When we have staged all the changes we like, we are ready to **commit**. We enter metadata like a *commit message* and the Index tree becomes the newest commit. It is added to the history with a pointer to the previous HEAD, and HEAD is moved to point to it.

Key Concept: The Three Trees (cont'd)

Checkout: HEAD $\rightarrow$ Working Tree

When we "move" in the history, we move the HEAD pointer to another commit and **checkout** that commit. The commit pointed to in the history is copied and becomes the files we see in the new working tree.

Key Concept: The Three Trees (cont'd)

That's git's main job. All the rest is details!

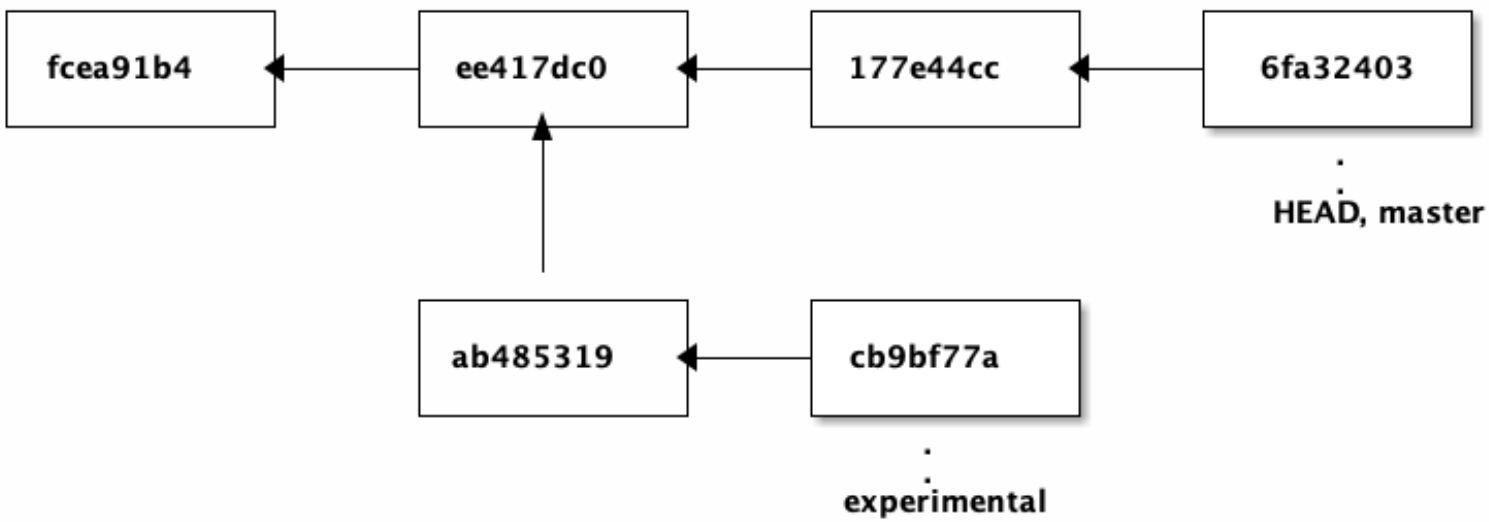
Branching and Merging

Creating a branch (like experimental above), creates a new named pointer that points to some specified commit. That commit is by definition considered the "tip" of a history.

Branching and Merging

Creating a branch (like experimental above), creates a new named pointer that points to some specified commit. That commit is by definition considered the "tip" of a history.

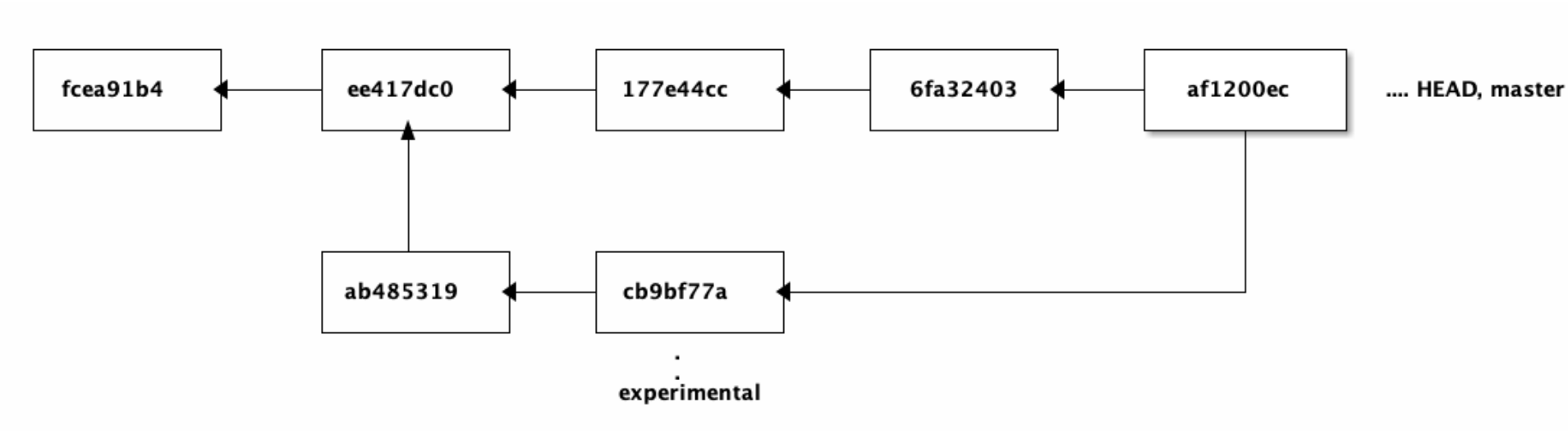
Switching to a branch moves HEAD to that branch, making it current. Commits at that point will advance both HEAD and the currrent branch.



Branching and Merging

Creating a branch (like experimental above), creates a new named pointer that points to some specified commit. That commit is by definition considered the "tip" of a history.

If we want to reconcile two branches of development, we **merge** them



We may need to address *conflicts* between the two trees!

Git Commands

Using git happens at several levels: porcelain, plumbing, and hosting services. Most common uses are at a high level, but the plumbing can be useful.

Common commands:

`git status`	Get current state of repo and three trees
`git log`	Look at repository history
`git diff`	See what has changed between commits
`git init`	Initialize a new repository
`git add`	Stage one or more files
`git branch`	Create and manage branches
`git switch`	Switch branches
`git checkout`	Checkout a commit
`git commit`	Commit the current index
`git merge`	Combine changes from multiple commits
`git remote`	Associate repo with a remote repo
`git push`	Send new commits to a remote copy of this repo
`git fetch/pull`	Receive new commits from a remote repo
`git worktree`	Work on multiple branches at the same time

Quick demo

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

Homework Workflow

1. Use `new-homework` script to create and populate a branch for each assignment. Start in `main` branch when doing so.
2. Suggestion: use `git worktree` to work on an assignment branch. This prevents branch mixups and makes it easy to work on several assignments simultaneously.
3. Put an `exercises` directory in `~/s750` and outside your assignments directory. Use that as the target of your worktrees.
4. When you are done with an assignment/commit, push to your github repo.
5. When ready to (re)submit, create a Pull Request on Github, comparing the exercise branch to `clean-start`.

Questions?

Good Git Practices

- Commit often

  Committing after individual, complete changes makes it much easier to track your progress – and move backward when needed.

  Even provisional commits can be reasonable if so labeled.

- Write good commit messages

  It should explain what you changed and *why* you changed it. The first line should be a short summary, easy to interpret from the log.

  Label breaking changes, keep a log of bugs/issues and indicate the fix or change by id.

- Hooks can take regular actions at each commit, e.g., run tests.

- Use branches for trying things out, especially when collaborating.

- `git blame` is a great tool for finding changes.

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

What is Happening Here?

```
fit <- lm(dist ~ speed, data=cars)
predict(fit)
summary(fit)
```

What about?

```
path = Path(start_path)
if not path.exists():
    return None
path = path.resolve()
```

Programming with *Nouns*

An **object** is a collection of *data* together with *methods* that operate on that data.

A **class** is a Type that describes a specific set of objects.

Object-oriented programming (OOP) uses objects/classes as the central abstraction. "Objects as nouns."

Key Principles:

- Encapsulation
- Polymorphism
- Extensibility
- Inheritance (use carefully)
- Separation of Concerns
- Dependency Inversion (dependence on the abstraction not the details)
- Safety

Aside: The Many Faces of OOP in R

R has many different systems for object-oriented programming. The main ones are:

- **S3**

  The oldest and simplest system, built on lists (usually). An object is just a variable that's been labeled as having a certain class. Generic functions can be written to operate on different classes. Commonly used in base R.

  See help on `Methods_for_S3`.
- **S4**

  A more sophisticated system with inheritance, multiple dispatch, and more formality. Heavily used in some circles (e.g., Matrix, Bioconductor), and it has some big advantages. But it is less commonly used overall.

  See help on `Methods_Details`, `Classes_Details`, `setClass`.
- **R6**

  A lighter weight and higher performance version of RC (Reference Classes). Has reference semantics, public/private methods, properties (called /active bindings/), and other features familiar in other languages. (See r6.r-lib.org for details.)
- **S7**

  A new OOP style that tries to combine the best of S3 + S4 (thus S7) with a more modern, ergonomic feel. (See here for details.) This is not yet in base R but is available from CRAN.

Example: Trees

Consider the type of binary trees that store their data in their internal nodes and a few operations on them:

```
data BinaryTree a = EmptyTree | Branch (BinaryTree a) a (BinaryTree a)

insert : Ord a => BinaryTree a -> a -> BinaryTree a
delete : BinaryTree a -> a -> BinaryTree a
inorder : BinaryTree a -> List a
find : BinaryTree a -> a -> Bool
```

A class that describes this type gives a *template* for the data and a definition of the operations as ***methods***.

```
class BinaryTree[A]:
    def __init__(self):
        self._left: BinaryTree[A] | None = None
        self._data: A | None = None    # invariant: None implies empty tree
        self._right: BinaryTree[A] | None = None

    ...
```

Example: Trees

```
class BinaryTree[A]:
    def __init__(self):
        self._left: BinaryTree[A] | None = None
        self._data: A | None = None    # invariant: None implies empty tree
        self._right: BinaryTree[A] | None = None

    def insert(self, value):       # Assume A is ordered
        if self._data is None:     # Empty tree, just change data
            self._data = value

        if value == self._data:
            return

        if value < self._data:
            if self._left is None:
                self._left = BinaryTree().insert(value)
            else:
                self._left.insert(value)
        else:
            if self._right is None:
                self._right = BinaryTree().insert(value)
            else:
                self._right.insert(value)

    ...
```

Example: Trees

```
    def delete(self, value):
        ...

    def inorder(self):
        ...

    def find(self, target):
        ...

    ...
```

Example: Trees

```
b = BinaryTree()
for d in [2, 4, 7, 1, 16, 3]:
    b.insert(d)

b.search(4) # True
b.search(5) # False
b.insert(5)
b.search(5) # True

b2 = BinaryTree()
for d in [-3, 2, 17]:
    b2.insert(d)
b2.search(5) # False
```

Here `b` and `b2` are *instances* of the `BinaryTree` class. Each has separate data and does not affect the other.

The *attributes* of the objects are also available (e.g., `b._data`), but with some exceptions, we discourage accessing them directly.

We could provide some convenient sugar, e.g., read-only access:

```
@property
def data(self):
    return self._data
```

Hyperreal Numbers

Suppose I would like to model the *hyperreal numbers*. I won't go into great detail, but the hyperreals offer a rigorous definition of infinitesimals, the "dx"s you often handwaved away in introductory calculus. A hyperreal number has a real part (or standard part) and an infinitesimal part.

Again, we can make a template, this time using S4.

Hyperreal Numbers

```
setClass("hyperreal", slots = c(x = "numeric", dx = "numeric"))

hyper <- function(x, dx) {
  new("hyperreal", x = x, dx = dx)
}

setMethod("+", signature(e1 = "hyperreal", e2 = "hyperreal"),
          function(e1, e2) {
              hyper(e1@x + e2@x, e1@dx + e2@dx)
          })

setMethod("+", signature(e1 = "hyperreal", e2 = "numeric"),
          function(e1, e2) {
              hyper(e1@x + e2, e1@dx)
          })
...
setMethod("sin", signature(x="hyperreal"),
          function(x) {
              hyper(sin(x@x), cos(x@x) * x@dx)
          })
setMethod("cos", signature(x="hyperreal"),
          function(x) {
              hyper(cos(x@x), - sin(x@x) * x@dx)
          })
...
```

Hyperreal Numbers (cont'd)

Once these methods are defined, any function that works on numerics also work on hyperreals.

```
foo <- function(x) {
    sin(x)^2 + 3*x^2 + log(x) - 4
}
```

`foo` is ***polymorphic*** or ***generic***: it operates on any type which implements the required operations. This yields, e.g.,

```
> foo(4)
[1] 45.95904

> foo(hyper(4, 1))
An object of class "hyperreal"
Slot "x":
[1] 45.95904

Slot "dx":
[1] 25.23936
```

Interactive Examples

What are the data being encapsulated? What are the methods/operations? Think of this as types

1. Images
2. Documents
3. Books and Libraries
4. Paths
5. Monoids

Plan

Binary Search Debrief

The Command Line

Version Control and Git

Aside: Homework Workflow

Objects

Migit

Migit

We are going to write a small git clone that handles many of the most common tasks. This will help us understand how git works and give us a practical case study for thinking about object-oriented programming, structuring, writing, and testing good code.

Migit

We are going to write a small git clone that handles many of the most common tasks. This will help us understand how git works and give us a practical case study for thinking about object-oriented programming, structuring, writing, and testing good code.

First steps:

1. Separate executable and library
2. Driver routine for the program
3. Design discussion, structure summary

Migit

We are going to write a small git clone that handles many of the most common tasks. This will help us understand how git works and give us a practical case study for thinking about object-oriented programming, structuring, writing, and testing good code.

First steps:

1. Separate executable and library
2. Driver routine for the program
3. Design discussion, structure summary

Next steps:

1. The `init` command
2. Create an abstraction for the Repository (at least in somewhat simplified form)

The End