An Object-ive View!

Christopher R. Genovese

Department of Statistics & Data Science

04 Sep 2025
Session #4

Plan

Some Git Details

Plan

Some Git Details

Aside: Homework Workflow

Plan

Some Git Details

Aside: Homework Workflow

Objects

Plan

Some Git Details

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/`
  - Version Control `https://36-750.github.io/tools/version-control/`
  - `https://clig.dev/`
  - Writing Commands on home page
  - Git from the Bottom Up by John Wiegley
  - `https://learngitbranching.js.org/`
  - 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.

Goals for Today

Last time, we saw the basic workings of git and looked at standards for command-line apps.

Today's goals:

1. Examine some practical details on git workflows
2. Study objects as a programming paradigm
3. Introduce migit and build an executable

Plan

Git Worktree

`git worktree` is a sub-command that allows us to work on multiple branches *simultaneously*. It reduces problems in switch among branches with fewer problems or complications.

The worktrees are associated with a given repository but located in a separate directory.

Example:

```
cd ~/s750/
ls
#=> see the repo listed: assigments-ME

# make a directory to house your worktrees
mkdir worktrees

# move into the repo
cd assignments-ME

# Create a worktree for two branches swag and migit
git worktree add ../worktrees/swag swag
git worktree add ../worktrees/swag migit
```

Git Worktree

```
# work on one branch
cd ../worktrees/swag

git diff   # make changes and see what they are
# stage and commit them
git add file1 file2 file3
git commit
git status
git log --graph --oneline --color

# back to the repo
cd ../../assignments-ME
git switch swag   # <= doesn't work, swag's already spoken for
git worktree remove ../worktrees/swag
git switch swag   # works, notice changes

# Push changes to remote (see note on next slide)
git push origin swag
git switch main
# work on the other branch
cd ../worktrees/migit
# ...and so on
```

Two Practical Notes

When pushing a new branch to an upstream remote *for the first time*, you need to tell git to establish a tracking relationship between the local branch and the remote branch.

To do this, use

```
git push --set-upstream origin a-new-branch
```

After that, you can do `git push`, `git pull`, and `git fetch` on the branch without adornment.

Remember: this is a one-time setup per branch. You don't *have* to do this, but it makes some things more convenient.

An *upstream* is another branch name associated with a regular, local branch. It is usually a remote "tracking" branch.

```
git branch -a
* main
  a-new-branch
  remotes/origin/main
  remotes/origin/a-new-branch
```

Two Practical Notes

Don't forget to configure git

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

See the System Setup page for details.

Plan

Some Git Details

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 `worktrees` 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`.

Remember: Commit often, write good git messages, and use branches when you need them.

Plan

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** packages *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 – an *is-a* relationship (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
```

Example: 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.

Example: 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
```

Example: Monoids

While we have several choices, including having monoid objects *wrap* values, here we use monoid objects as *interpreters* of values. This eliminates one level of wrapping/unwrapping as we process the data.

Each monoid has an ‘munit‘ property, an ‘mcombine‘ method, and a label, but we exclude ‘label‘ from the protocol. We also include a ‘conforms‘ method in our implementations for checking valid inputs but we do not use it inside mcombine, say, or include it in the protocol.

Let’s consider how to do this in both Python and R.

Example: Monoids

```python
class Monoid:
    @property
    def munit(self):
        ...
    def mdot(self, x, y):
        ...
    def conforms(self, x) -> bool:
        return True
    @property
    def label(self):
        return self.__class__.__name__.rstrip('M')
    def __str__(self):
        return str(self.label)
    def __repr__(self):
        return f'{self.__class__.__name__}()'

def munit(m):
    "The identity/unit element of a given monoid."
    return m.munit

def mcombine(m, x, y):
    "The combine operation for a given monoid and two monoidal values."
    return m.mcombine(x, y)
```

Example: Monoids

For example:

```
class SumM(Monoid):
    "A monoid that sums the numbers it sees."

    def mcombine(self, x, y):
        return x + y

    @property
    def munit(self):
        return 0

    def conforms(self, x) -> bool:
        "Checks for primitive numeric value. We would like this to be more general.
        return isinstance(x, int) or isinstance(x, float) or isinstance(x, complex)

Sum = SumM()
mcombine(Sum, 4, 10)  # => 14
```

Example: Monoids

Now in R S4/S3

```
setClass("Monoid")
setGeneric("munit", function(monoid) { standardGeneric("munit") })
setGeneric("mdot", function(monoid, x, y) { standardGeneric("mdot") })
```

This plays nicely with S3:

```
munit <- function(monoid, ...)                # S3 generic for S3 dispatch
    UseMethod("munit")
setGeneric("munit")                           # S4 generic, default is S3 generic
munit.Monoid <- function(monoid, ...) {}      # S3 method for S3 class
setMethod("munit", "Monoid", munit.Monoid)    # S4 method for S4 class
```

Monoid instances are by inheritance:

```
SumM <- setClass("SumM", contains="Monoid")
Sum <- SumM()
setMethod("munit", signature("SumM"),
          function(monoid) 0)
setMethod("mdot", signature("SumM", "numeric", "numeric"),
          function(monoid, x, y) x + y)
```

Example: Monoids

Now in R with S7

```
Monoid <- new_class("Monoid")
munit <- new_generic("munit", "m")
mdot <- new_generic("mdot", "m")
```

Then, we can do

```
SumM <- new_class("SumM", parent=Monoid)
method(munit, SumM) <- function(m) { return(0) }
method(mdot, SumM) <- function(m, x, y) { return(x + y) }
Sum <- SumM()
mdot(Sum, 4, 10)  #=> 14
```

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

Introducing Migit

We are going to write a small git clone that handles some 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.

Introducing Migit

We are going to write a small git clone that handles some 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. Setup and Driver for the program
2. `GitRepository` object
3. The `init` command (later)

Introducing Migit

We are going to write a small git clone that handles some 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. Setup and Driver for the program
2. `GitRepository` object
3. The `init` command (later)

Next steps:

1. `GitObject` and `GitBlob` classes
2. `cat-file` command
3. `hash-object` command
4. `GitCommit` class

Migit Objects

There are four main types of objects in (mi)git: **blobs**, **trees**, **commits**, and **tags**.

We need to represent each of these objects with an entity in our code.

Objects provide packages for data ***with an interface***. In the case of the (mi)git, many operations can act on any of these objects, interchangeably, so it makes sense to have a common interface for them.

This suggests creating a `GitObject` class from which the other four object types all inherit.

Another fundamental entity in (mi)git is the ***repository***. In the file system, a repository is represented by the presence of a `.git` directory at the project root that contains particular files.

It makes sense to abstract this into an object that we can manipulate without worrying too much about the details of the file system on a particular machine.

To this end, we want to create a `GitRepository` object. For the moment, all we need from this are methods for finding the project root from an arbitrary path and for testing if a path is inside a (mi)git repository.

Migit Task #1: Driver

We will separate our code into an executable that runs the program – a *driver*, and a library that does most of the work.

To do this, we

1. Create an executable file `migit` somewhere on our path

   In Python, `migit` might look like:

   ```
   #!/usr/bin/env python3
   # -*- mode: python -*-

   from migit.main import main

   main()
   ```

   In R, it will be similar

   ```
   #!/usr/bin/env Rscript
   # -*- mode: r-mode -*-

   migit::main()
   ```

Migit Task #1: Driver

We will separate our code into an executable that runs the program – a *driver*, and a library that does most of the work.

To do this, we

2. Create a `migit` *package* which will hold the library code.

   In Python, the minimum requirement is a directory having a `__init__.py`. But more generally, I recommend `uv`:

   ```
   uv init --package migit
   ```

   In R, we can use `devtools`

   ```
   devtools::create("migit")
   ```

Migit Task #1: Driver

We will separate our code into an executable that runs the program – a *driver*, and a library that does most of the work.

To do this, we

3. Create the driver in `main.py` or `main.r`.

   This should support

   - `migit --help` and `migit --version`
   - `migit subcommand --help`
   - `migit subcommand ARGS`

   (See example sketch.)

Migit Task #2: `GitRepository` Object

In the file system, a repository is represented by the presence of a `.git` directory at the project root that contains particular files. You can see what these are by using `git init` in a temporary directory and looking at the `.git` it creates.

For the moment, all we need from the `GitRepository` object are methods for finding the project root from an arbitrary path and for testing if a path is inside a (mi)git repository.

To this end, create class (formal or informal) for a `GitRepository`. What data do you need? What methods?

Migit Task #2: The `init` Command

The `migit init` command should accept the path of a candidate project root as an argument and create a repository there.

However, if a repository already exist at that path, the command should stop. If a files exist but a repository does not, it should set up the `.git` directory.

The End