COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

# GR5206: lecture 4

*Computational Statistics*
*And Introduction to Data Science*

Thibault Vatter
Department of Statistics, Columbia University

9/27/2019

# Outline

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

# The two programming paradigms



- **Imperative:**
  - The programmer instructs the machine how to change its state.
  - Two kinds:
    - **Procedural:** groups instructions into procedures.
    - **Object-oriented:** groups instructions together with the part of the state they operate on.
- **Declarative:**
  - The programmer declares properties of the desired result, but not how to compute it.
  - Three kinds:
    - **Functional:** the output results of a series of function applications.
    - **Logic:** the output is the answer to a question about a system of facts and rules.
    - **Mathematical:** the output is the solution of an optimization problem.

# What about R?



- A bit of everything:
  - Powerful but complex.
- Declarative:
  - Mathematical: optimization with `optim` and specialized packages.
  - Functional: the hearth of R.
- Imperative:
  - Procedural: functions loaded with `source()`.
  - Object-oriented: the `S3` class system (and others).

# OO programming languages

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Based on "objects", which can contain
  - **Data** (fields):
    - often known as attributes or properties,
  - **Code** (procedures):
    - often known as methods.
- Two important concepts:
  - **Polymorphism:**
    - A function's interface is separate from its implementation.
    - Possible to use the same function for different types of input.
  - **Encapsulation:**
    - Users don't need to worry about details of an object because they are hidden behind a standard interface.

```
summary(ggplot2::diamonds$carat)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    0.20    0.40    0.70    0.80    1.04    5.01

summary(ggplot2::diamonds$cut)
#>      Fair      Good Very Good   Premium     Ideal
#>      1610      4906     12082     13791     21551
```

# OO systems and classes

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- OOS systems:
  - **Class:**
    - The type of an object.
    - What the object **is**.
  - **Methods:**
    - Procedures/implementations for a specific class.
    - What the object can **do**.
- Classes:
  - Define **fields:**
    - Data possessed by every instance of that class.
  - Organized in a hierarchy:
    - **Inheritance:** if a method does not exist for one class, its parent's method is used.
    - E.g., ordered factors inherit from regular factors.
    - **Method dispatch:** how to find the correct method given a class.

# Two main paradigms of OOP



- **Encapsulated** OOP:
  - Methods belong to objects or classes.
  - Method calls look like `object.method(arg1, arg2)`.
  - Objects encapsulate both
    - data (with fields),
    - and behavior (with methods).
  - Paradigm found in most popular languages.
- **Functional** OOP:
  - Methods belong to **generic** functions
  - Method calls look like ordinary function calls: `generic(object, arg2, arg3)`.
  - Functional because:
    - It looks like any function call from the outside.
    - And internally the components are also functions.

(Using the terminology of *Extending R* (Chambers 2016))

# OOP in R



- **S3**
  - R's first OOP system
  - See *Statistical Models in S* (Chambers and Hastie 1992).
  - Informal implementation of functional OOP.
  - Relies on common conventions rather than ironclad guarantees.
  - Easy to get started with!
- **S4**
  - Formal rewrite of S3
  - See *Programming with Data* (Chambers 1998).
  - Harder than S3, but provides guarantees and greater encapsulation.
- **RC**
  - Encapsulated OOP.
  - Special type of S4 objects that are also **mutable**.
- **R6**
  - A package ((Chang 2017)).
  - Encapsulated OOP like RC, but better.
  - See section 14 of Advanced-R.

# sloop



- "Sail the seas of OOP"
- Helpers to fill in missing pieces in base R.

```r
library(sloop)
```

- otype() to figure out the OOP system used by an object:

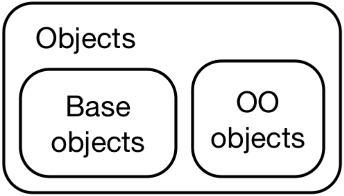
```r
otype(1:10)
#> [1] "base"

otype(mtcars)
#> [1] "S3"
```

# Outline

# Base types

*Everything that exists in R is an object.*
*— John Chambers*

- . . . but not everything is object-oriented.
  - Base objects come from S:
    - Developed before the need for an OOP system.
    - Tools and nomenclature evolved organically.
    - No single guiding principle.
- In R, we use the terms **base objects** and **OO objects** to distinguish objects and object-oriented objects.



# Base versus OO objects



- Use `is.object()` or `sloop::otype()` to tell the difference.

```r
# A base object
is.object(1:10)
#> [1] FALSE
otype(1:10)
#> [1] "base"
```

```r
# An OO object
is.object(mtcars)
#> [1] TRUE
otype(mtcars)
#> [1] "S3"
```

- OO objects have a “class” attribute.

```r
x <- matrix(1:4, nrow = 2)
attr(x, "class")
#> NULL
attr(mtcars, "class")
#> [1] "data.frame"
```

- Use `sloop::s3_class()` instead of `class()`.

```r
class(x)
#> [1] "matrix"
s3_class(x)
#> [1] "matrix"  "integer" "numeric"
```

# Base types

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- Every object has a **base type**.

```
typeof(1:10)
#> [1] "integer"
typeof(mtcars)
#> [1] "list"
```

- Vectors.

```
typeof(NULL)
#> [1] "NULL"
typeof(1L)
#> [1] "integer"
typeof(1i)
#> [1] "complex"
```

- Functions.

```
typeof(mean)
#> [1] "closure"
typeof(`[`)
#> [1] "special"
typeof(sum)
#> [1] "builtin"
```

# Base types cont'd

- Environments.

```
typeof(globalenv())
#> [1] "environment"
```

- The S4 type.

```
typeof(stats4::mle(function(x = 1) (x - 2) ^ 2))
#> [1] "S4"
```

- Language components.

```
typeof(quote(a))
#> [1] "symbol"
typeof(quote(a + 1))
#> [1] "language"
typeof(formals(mean))
#> [1] "pairlist"
```

- . . . and more

# Outline

# S3



- R's first and simplest OO system.
  - The only system used in the base and stats packages.
  - The most commonly used system in packages.
  - Informal and ad hoc, but elegantly minimalist.
    - Take away any part and it becomes useless.
  - Without a compelling reason to do otherwise, **use it**.
- Very flexible.
  - Possible to do ill-advised things (contrasted to Java/C++).
  - But gives a lot of freedom.
- Outline:
  - The main components: classes, generics, and methods.
  - Creating a new class: constructors and helpers.
  - S3 generics, S3 methods, and method dispatch.
  - Inheritance and how to make a class "subclassable".

# Basics



- A base type with at least a `class` attribute.

```
f <- factor(c("a", "b", "c"))

typeof(f)
#> [1] "integer"
attributes(f)
#> $levels
#> [1] "a" "b" "c"
#>
#> $class
#> [1] "factor"
```

- Get the underlying base type by `unclass()`ing it.

```
unclass(f)
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

# Generic functions



- S3 objects behave differently from the base type whenever passed to **generics**.
- Use `sloop::ftype()` to tell if a function is a generic.

```
ftype(print)
#> [1] "S3"      "generic"
ftype(str)
#> [1] "S3"      "generic"
ftype(unclass)
#> [1] "primitive"
```

- Generic functions define interfaces:
  - Implementation depend on the argument's class.
  - Many base R functions are generics.

```
print(f)
#> [1] a b c
#> Levels: a b c
print(unclass(f)) # stripping class reverts to integer behaviour
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

# Generic functions cont'd

- The generic's job is to
  - define the interface (i.e. the arguments),
  - and find the right implementation.
- Two definitions:
  - **Method**, the implementation for a specific class.
  - **Method dispatch**, how the generic finds that method.
    - Use `sloop::s3_dispatch()` to see the process of method dispatch:

```
s3_dispatch(print(f))
#> => print.factor
#>  * print.default
```

# Generic functions cont'd

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- S3 methods are functions with special naming scheme (`generic.class()`).
  - E.g., `factor` method for the `print()` generic is `print.factor()`.
  - Identify a method by the presence of `.` in the name.
    - ... but some important functions were written before S3.
    - If unsure, use `sloop::ftype()`.

```
ftype(t.test)
#> [1] "S3"      "generic"
ftype(t.data.frame)
#> [1] "S3"     "method"
```

- Use `sloop::s3_get_method()` to see the source code.

```
weighted.mean.Date
#> Error in eval(expr, envir, enclos): object 'weighted.mean.Date' not found
s3_get_method(weighted.mean.Date)
#> function (x, w, ...)
#> structure(weighted.mean(unclass(x), w, ...), class = "Date")
#> <bytecode: 0x55f8e7e6fd40>
#> <environment: namespace:stats>
```

# Classes



- S3 has no formal definition of a class!
  - Different from most other OOP languages.
  - Set the **class attribute** to make an instance of a class.
  - E.g., during/after creation with `structure()`/`class<-()`.

```r
# Create and assign class in one step
x <- structure(list(), class = "my_class")

# Create, then set class
x <- list()
class(x) <- "my_class"
```

- Determine the class:

```r
class(x)
#> [1] "my_class"
s3_class(x)
#> [1] "my_class"
```

- See if it's a class' instance:

```r
inherits(x, "my_class")
#> [1] TRUE
inherits(x, "your_class")
#> [1] FALSE
```

- Class name can be any string.
  - Good: use only letters and _.
  - Bad: use .

# Classes cont'd

- No checks for correctness.

```r
# Create a factor
f <- factor(letter)
#> Error in factor(letter): object 'letter' not found
class(f)
#> [1] "factor"
print(f)
#> [1] a b c
#> Levels: a b c

# Turn it into a date (?!)
class(f) <- "Date"

# Unsurprisingly this doesn't work very well
print(f)
#> [1] "1970-01-02" "1970-01-03" "1970-01-04"
```

- R doesn't stop you from shooting yourself in the foot, so don't aim the gun at your toes and pull the trigger!

# Constructors



- S3 has no a formal definition of a class, so no built-in way to ensure that objects of a given class have the same structure.
- Consistency of the structure enforced with **constructors**.
- Should follow three principles:
  - Be called `new_myclass()`.
  - Have one argument for the base object, and one for each attribute.
  - Check the type of the base object and the types of each attribute.

```r
new_Date <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "Date")
}

new_Date(c(-1, 0, 1))
#> [1] "1969-12-31" "1970-01-01" "1970-01-02"
```

# Constructors cont'd



```r
new_difftime <- function(x = double(), units = "secs") {
  stopifnot(is.double(x))
  units <- match.arg(units, c("secs", "mins", "hours", "days", "weeks"))

  structure(x,
    class = "difftime",
    units = units
  )
}

new_difftime(c(1, 10, 3600), "secs")
#> Time differences in secs
#> [1]    1   10 3600
new_difftime(52, "weeks")
#> Time difference of 52 weeks
```

- Intended audience: developers.
  - Means you can keep them simple.
  - Don't need to optimize error messages for public consumption.
  - OK to trade a little safety in return for performance, avoid potentially time-consuming checks.

# Helpers



- Audience: users.
- Goal: make their life as easy as possible.
- A helper should always:
  - Have the same name as the class, e.g. `myclass()`.
  - Finish by calling the constructor.
  - Create error messages tailored towards an end-user.
  - Have an interface with carefully chosen default values and useful conversions.
- The last bullet is the trickiest!

# Helpers via input coercion



- Our `difftime` constructor is very strict.

```
new_difftime(1:10)
#> Error in new_difftime(1:10): is.double(x) is not TRUE
```

- Just coerces the input to a double.

```
difftime <- function(x = double(), units = "secs") {
  x <- as.double(x)
  new_difftime(x, units = units)
}

difftime(1:10)
#> Time differences in secs
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

# Helpers via decomposition



```r
POSIXct <- function(x, tzone = "") {
  as.POSIXct(x, tz = tzone, origin = "1970-01-01")
}

POSIXct(365*86400*30, tzone = "America/New_York")
#> [1] "1999-12-24 19:00:00 EST"

POSIXct <- function(year = integer(),
                    month = integer(),
                    day = integer(),
                    hour = 0L,
                    minute = 0L,
                    sec = 0,
                    tzone = "") {
  ISOdatetime(year, month, day, hour, minute, sec, tz = tzone)
}

POSIXct(1999, 12, 24, 19, tzone = "America/New_York")
#> [1] "1999-12-24 19:00:00 EST"
```

# Generics and methods

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

- The job of an S3 generic: **method dispatch**, i.e. find the specific implementation for a class.
- Performed by `UseMethod()`.
- Most generics are very simple.

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x55f8e591dfa8>
#> <environment: namespace:base>
```

- Creating your own generic is similarly simple.

```
my_new_generic <- function(x) {
  UseMethod("my_new_generic")
}
```

(If you wonder why we have to repeat `my_new_generic` twice, read Section 6.2.3.)

# Method dispatch



- How does `UseMethod()` work?
  - Basically creates a vector of method names,
  - And then looks for each potential method in turn.

```
x <- Sys.Date()
s3_dispatch(print(x))
#> => print.Date
#>  * print.default
```

- The output:
  - => indicates the method that is called, here `print.Date()`
  - * indicates a method that is defined, but not called, here `print.default()`.
- The essence of method dispatch is simple, but gets more complicated to encompass inheritance and more.

# Finding methods

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

```
sloop::s3_methods_generic("mean")
#> # A tibble: 7 x 4
#>   generic class       visible source
#>   <chr>   <chr>       <lgl>   <chr>
#> 1 mean    Date        TRUE    base
#> 2 mean    default     TRUE    base
#> 3 mean    difftime    TRUE    base
#> 4 mean    POSIXct     TRUE    base
#> 5 mean    POSIXlt     TRUE    base
#> 6 mean    quosure     FALSE   registered S3method
#> 7 mean    vctrs_vctr  FALSE   registered S3method
sloop::s3_methods_class("ordered")
#> # A tibble: 7 x 4
#>   generic       class   visible source
#>   <chr>         <chr>   <lgl>   <chr>
#> 1 as.data.frame ordered TRUE    base
#> 2 is_vector_s3  ordered FALSE   registered S3method
#> 3 Ops           ordered TRUE    base
#> 4 relevel       ordered FALSE   registered S3method
#> 5 scale_type    ordered FALSE   registered S3method
#> 6 Summary       ordered TRUE    base
#> 7 type_sum      ordered FALSE   registered S3method
```

# Inheritance



- S3 classes can share behavior through **inheritance**.
- Powered by three ideas.
- The class can be a character *vector*.

```
class(ordered("x"))
#> [1] "ordered" "factor"
```

```
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
```

- If a method is not found for the class in the first element of the vector, R looks for a method for the second class (and so on).

```
s3_dispatch(print(ordered("x")))
#>    print.ordered
#> => print.factor
#>  * print.default
```

```
s3_dispatch(print(Sys.time()))
#> => print.POSIXct
#>    print.POSIXt
#>  * print.default
```

- A method can delegate work by calling `NextMethod()`.

```
s3_dispatch(ordered("x")[1])
#>    [.ordered
#> => [.factor
#>    [.default
#> -> [ (internal)
```

```
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

# NextMethod()

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- The hardest part of inheritance to understand!
- An example for the most common use case: [.

```
new_secret <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "secret")
}
print.secret <- function(x, ...) {
  print(strrep("x", nchar(x)))
  invisible(x)
}

(x <- new_secret(c(15, 1, 456)))
#> [1] "xx"  "x"   "xxx"
```

- Works, but the default [ method doesn't preserve the class.

```
s3_dispatch(x[1])
#>    [.secret
#>    [.default
#> => [ (internal)
x[1]
#> [1] 15
```

# NextMethod() cont'd



- What is the issue with the following?

```
`[.secret` <- function(x, i) new_secret(x[i])
```

- A naive solution:

```
`[.secret` <- function(x, i) {
  x <- unclass(x)
  new_secret(x[i])
}
x[1]
#> [1] "xx"
```

- Or better:

```
`[.secret` <- function(x, i) {
  new_secret(NextMethod())
}
x[1]
#> [1] "xx"
s3_dispatch(x[1])
#> => [.secret
#>    [.default
#> -> [ (internal)
```

# Allowing subclassing



- To allow subclasses:
  - Parent constructor with `...` and `class` arguments.
  - Subclass constructor calls it with additional arguments.

```
new_secret <- function(x, ..., class = character()) {
  stopifnot(is.double(x))

  structure(
    x,
    ...,
    class = c(class, "secret")
  )
}
new_supersecret <- function(x) {
  new_secret(x, class = "supersecret")
}
print.supersecret <- function(x, ...) {
  print(rep("xxxxx", length(x)))
  invisible(x)
}
new_supersecret(c(15, 1, 456))
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

# Outline

Columbia University
in the city of New York

# References



Chambers, John M. 1998. *Programming with Data: A Guide to the S Language*. Springer.

———. 2016. *Extending R*. CRC Press.

Chambers, John M, and Trevor J Hastie. 1992. *Statistical Models in S*. Wadsworth & Brooks/Cole Advanced Books & Software.

Chang, Winston. 2017. *R6: Classes with Reference Semantics*. https://r6.r-lib.org.