OBJECT-ORIENTED PROGRAMMING

Introduction

In the following five chapters you'll learn about object-oriented programming (OOP). OOP is a little more challenging in R than in other languages because:

There are multiple OOP systems to choose from. In this book, I'll focus on the three that I believe are most important: S3, R6, and S4. S3 and S4 are provided by base R. R6 is provided by the R6 package, and is similar to the Reference Classes, or RC for short, from base R.

There is disagreement about the relative importance of the OOP systems. I think S3 is most important, followed by R6, then S4. Others believe that S4 is most important, followed by RC, and that S3 should be avoided. This means that different R communities use different systems.

S3 and S4 use generic function OOP which is rather different from the encapsulated OOP used by most languages popular today. We'll come back to precisely what those terms mean shortly, but basically, while the underlying ideas of OOP are the same across languages, their expressions are rather different. This means that you can't immediately transfer your existing OOP skills to R.

Generally in R, functional programming is much more important than object-oriented programming, because you typically solve complex problems by decomposing them into simple functions, not simple objects. Nevertheless, there are important reasons to learn each of the three systems:

S3 allows your functions to return rich results with user-friendly display and programmer-friendly internals. S3 is used throughout base R, so it's important to master if you want to extend base R functions to work with new types of input.

R6 provides a standardised way to escape R's copy-on-modify semantics. This is particularly important if you want to model objects that exist independently of R. Today, a common need for R6 is to model data that comes from a web API, and where changes come from inside or outside of R.

S4 is a rigorous system that forces you to think carefully about program design. It's particularly well-suited for building large systems that evolve over time and will receive contributions from many programmers. This is why it is used by the Bioconductor project, so another reason to learn S4 is to equip you to contribute to that project.

The goal of this brief introductory chapter is to give you some important vocabulary and some tools to identify OOP systems in the wild. The following chapters then dive into the details of R's OOP systems:

Chapter "Base types" details the base types which form the foundation underlying all other OO system.

Chapter "S3" introduces S3, the simplest and most commonly used OO system.

Chapter "R6" discusses R6, a encapsulated OO system built on top of environments.

Chapter "S4" introduces S4, which is similar to S3 but more formal and more strict.

Chapter "Trade-offs" compares these three main OO systems. By understanding the trade-offs of each system you can appreciate when to use one or the other.

This book focusses on the mechanics of OOP, not its effective use, and it may be challenging to fully understand if you have not done object-oriented programming before. You might wonder why I chose not to provide more immediately useful coverage. I have focussed on mechanics here because they need to be well described somewhere (writing these chapters required a considerable amount of reading, exploration, and synthesis on my behalf), and using OOP

effectively is sufficiently complex to require book-length treatment; there's simply not enough room in Advanced R to cover it in the depth required.

OOP systems

Different people use OOP terms in different ways, so this section provides a quick overview of important vocabulary. The explanations are necessarily compressed, but we will come back to these ideas multiple times.

The main reason to use OOP is polymorphism (literally: many shapes). Polymorphism means that a developer can consider a function's interface separately from its implementation, making it possible to use the same function form for different types of input. This is closely related to the idea of encapsulation: the user doesn't need to worry about details of an object because they are encapsulated behind a standard interface.

To be concrete, polymorphism is what allows summary() to produce different outputs for numeric and factor variables:

```
diamonds <- ggplot2::diamonds

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

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

You could imagine summary() containing a series of if-else statements, but that would mean only the original author could add new implementations. An OOP system makes it possible for any developer to extend the interface with implementations for new types of input.

To be more precise, OO systems call the type of an object its class, and an implementation for a specific class is called a method. Roughly speaking, a class defines what an object is and methods describe what that object can do. The class defines the fields, the data possessed by every instance of that class. Classes are organised in a hierarchy so that if a method does not exist for one class, its parent's method is used, and the child is said to inherit behaviour. For example, in R, an ordered factor inherits from a regular factor, and a generalised linear model inherits from a linear model. The process of finding the correct method given a class is called method dispatch.

There are two main paradigms of object-oriented programming which differ in how methods and classes are related. In this book, we'll borrow the terminology of Extending R (Chambers 2016) and call these paradigms encapsulated and functional:

In encapsulated OOP, methods belong to objects or classes, and method calls typically look like object.method(arg1, arg2). This is called encapsulated because the object encapsulates both data (with fields) and behaviour (with methods), and is the paradigm found in most popular languages.

In functional OOP, methods belong to generic functions, and method calls look like ordinary function calls: generic(object, arg2, arg3). This is called functional because from the outside it looks like a regular function call, and internally the components are also functions.

With this terminology in hand, we can now talk precisely about the different OO systems available in R.

OOP in R

Base R provides three OOP systems: S3, S4, and reference classes (RC):

S3 is R's first OOP system, and is described in Statistical Models in S (Chambers and Hastie 1992). S3 is an informal implementation of functional OOP and relies on common conventions rather than ironclad guarantees. This makes it easy to get started with, providing a low cost way of solving many simple problems.

S4 is a formal and rigorous rewrite of S3, and was introduced in Programming with Data (Chambers 1998). It requires more upfront work than S3, but in return provides more guarantees and greater encapsulation. S4 is implemented in the base methods package, which is always installed with R.

(You might wonder if S1 and S2 exist. They don't: S3 and S4 were named according to the versions of S that they accompanied. The first two versions of S didn't have any OOP framework.)

RC implements encapsulated OO. RC objects are a special type of S4 objects that are also mutable, i.e., instead of using R's usual copy-on-modify semantics, they can be modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve in the functional OOP style of S3 and S4.

A number of other OOP systems are provided by CRAN packages:

R6 (Chang 2017) implements encapsulated OOP like RC, but resolves some important issues. In this book, you'll learn about R6 instead of RC, for reasons described in Section 14.5.

R.oo (Bengtsson 2003) provides some formalism on top of S3, and makes it possible to have mutable S3 objects.

proto (Grothendieck, Kates, and Petzoldt 2016) implements another style of OOP based on the idea of prototypes, which blur the distinctions between classes and instances of classes (objects). I was briefly enamoured with prototype based programming (Wickham 2011) and used it in ggplot2, but now think it's better to stick with the standard forms.

Apart from R6, which is widely used, these systems are primarily of theoretical interest. They do have their strengths, but few R users know and understand them, so it is hard for others to read and contribute to your code.

sloop

Before we go on I want to introduce the sloop package:

```
library(sloop)
```

The sloop package (think "sail the seas of OOP") provides a number of helpers that fill in missing pieces in base R. The first of these is sloop::otype(). It makes it easy to figure out the OOP system used by a wild-caught object:

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

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

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

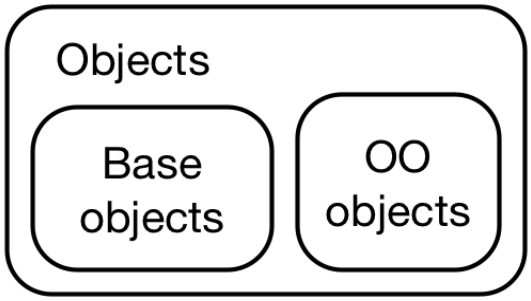
Use this function to figure out which chapter to read to understand how to work with an existing object.

BASE TYPES

Introduction

To talk about objects and OOP in R we first need to clear up a fundamental confusion about two uses of the word "object". So far in this book, we've used the word in the general sense captured by John Chambers' pithy quote: "Everything that exists in R is an object". However, while everything is an object, not everything is object-oriented. This confusion arises because the base objects come from S, and were developed before anyone thought that S might need an OOP system. The tools and nomenclature evolved organically over many years without a single guiding principle.

Most of the time, the distinction between objects and object-oriented objects is not important. But here we need to get into the nitty gritty details so we'll use the terms base objects and OO objects to distinguish them.



Base versus OO objects

To tell the difference between a base and OO object, use is.object() or sloop::otype():

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

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

Technically, the difference between base and OO objects is that OO objects have a "class" attribute:

```
attr(1:10, "class")
#> NULL

attr(mtcars, "class")
#> [1] "data.frame"
```

You may already be familiar with the class() function. This function is safe to apply to S3 and S4 objects, but it returns misleading results when applied to base objects. It's safer to use sloop::s3_class(), which returns the implicit class that the S3 and S4 systems will use to pick methods.

```
x <- matrix(1:4, nrow = 2)
class(x)
#> [1] "matrix"
sloop::s3_class(x)
#> [1] "matrix"  "integer" "numeric"
```

Base types

While only OO objects have a class attribute, every object has a base type:

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

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

Base types do not form an OOP system because functions that behave differently for different base types are primarily written in C code that uses switch statements. This means that only R-core can create new types, and creating a new type is a lot of work because every switch statement needs to be modified to handle a new case. As a consequence, new base types are rarely added. The most recent change, in 2011, added two exotic types that you never see in R itself, but are needed for diagnosing memory problems. Prior to that, the last type added was a special base type for S4 objects added in 2005.

In total, there are 25 different base types. They are listed below, loosely grouped according to where they're discussed in this book. These types are most important in C code, so you'll often see them called by their C type names. I've included those in parentheses.

Vectors, include types NULL (NILSXP), logical (LGLSXP), integer (INTSXP), double (REALSXP), complex (CPLXSXP), character (STRSXP), list (VECSXP), and raw (RAWSXP).

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

Functions, include types closure (regular R functions, CLOSXP), special (internal functions, SPECIALSXP), and builtin (primitive functions, BUILTINSXP).

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

Environments, have type environment (ENVSXP).

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

The S4 type (S4SXP), is used for S4 classes that don't inherit from an existing base type.

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

Language components, include symbol (aka name, SYMSXP), language (usually called calls, LANGSXP), and pairlist (used for function arguments, LISTSXP) types.

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

expression (EXPRSXP) is a special purpose type that's only returned by parse() and expression(). Expressions are generally not needed in user code.

The remaining types are esoteric and rarely seen in R. They are important primarily for C code: externalptr (EXTPTRSXP), weakref (WEAKREFSXP), bytecode (BCODESXP), promise (PROMSXP), ... (DOTSXP), and any (ANYSXP).

You may have heard of mode() and storage.mode(). Do not use these functions: they exist only to provide type names that are compatible with S.

Numeric type

Be careful when talking about the numeric type, because R uses "numeric" to mean three slightly different things:

In some places numeric is used as an alias for the double type. For example as.numeric() is identical to as.double(), and numeric() is identical to double().

(R also occasionally uses real instead of double; NA_real_ is the one place that you're likely to encounter this in practice.)

In the S3 and S4 systems, numeric is used as a shorthand for either integer or double type, and is used when picking methods:

```
sloop::s3_class(1)
#> [1] "double"  "numeric"
sloop::s3_class(1L)
#> [1] "integer" "numeric"
```

is.numeric() tests for objects that behave like numbers. For example, factors have type "integer" but don't behave like numbers (i.e. it doesn't make sense to take the mean of factor).

```
typeof(factor("x"))
#> [1] "integer"
is.numeric(factor("x"))
#> [1] FALSE
```

In this book, I consistently use numeric to mean an object of type integer or double.

S3

Introduction

S3 is R's first and simplest OO system. S3 is informal and ad hoc, but there is a certain elegance in its minimalism: you can't take away any part of it and still have a useful OO system. For these reasons, you should use it, unless you have a compelling reason to do otherwise. S3 is the only OO system used in the base and stats packages, and it's the most commonly used system in CRAN packages.

S3 is very flexible, which means it allows you to do things that are quite ill-advised. If you're coming from a strict environment like Java this will seem pretty frightening, but it gives R programmers a tremendous amount of freedom. It may be very difficult to prevent people from doing something you don't want them to do, but your users will never be held back because there is something you haven't implemented yet. Since S3 has few built-in constraints, the key to its successful use is applying the constraints yourself. This chapter will therefore teach you the conventions you should (almost) always follow.

The goal of this chapter is to show you how the S3 system works, not how to use it effectively to create new classes and generics. I'd recommend coupling the theoretical knowledge from this chapter with the practical knowledge encoded in the vctrs package.

Basics

An S3 object is a base type with at least a class attribute (other attributes may be used to store other data). For example, take the factor. Its base type is the integer vector, it has a class attribute of "factor", and a levels attribute that stores the possible levels:

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

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

You can get the underlying base type by unclass()ing it, which strips the class attribute, causing it to lose its special behaviour:

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

An S3 object behaves differently from its underlying base type whenever it's passed to a generic (short for generic function). The easiest way to tell if a function is a generic is to use sloop::ftype() and look for "generic" in the output:

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

A generic function defines an interface, which uses a different implementation depending on the class of an argument (almost always the first argument). Many base R functions are generic, including the important print():

```
print(f)
#> [1] a b c
#> Levels: a b c
```

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

Beware that str() is generic, and some S3 classes use that generic to hide the internal details. For example, the POSIXlt class used to represent date-time data is actually built on top of a list, a fact which is hidden by its str() method:

```
time <- strptime(c("2017-01-01", "2020-05-04 03:21"), "%Y-%m-%d")
str(time)
#>  POSIXlt[1:2], format: "2017-01-01" "2020-05-04"

str(unclass(time))
#> List of 9
#>  $ sec  : num [1:2] 0 0
#>  $ min  : int [1:2] 0 0
#>  $ hour : int [1:2] 0 0
#>  $ mday : int [1:2] 1 4
#>  $ mon  : int [1:2] 0 4
#>  $ year : int [1:2] 117 120
#>  $ wday : int [1:2] 0 1
#>  $ yday : int [1:2] 0 124
#>  $ isdst: int [1:2] 0 0
#>  - attr(*, "tzone")= chr "UTC"
```

The generic is a middleman: its job is to define the interface (i.e. the arguments) then find the right implementation for the job. The implementation for a specific class is called a method, and the generic finds that method by performing method dispatch.

You can use sloop::s3_dispatch() to see the process of method dispatch:

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

We'll come back to the details of dispatch, for now note that S3 methods are functions with a special naming scheme, generic.class(). For example, the factor method for the print() generic is called print.factor(). You should never call the method directly, but instead rely on the generic to find it for you.

Generally, you can identify a method by the presence of . in the function name, but there are a number of important functions in base R that were written before S3, and hence use . to join words. If you're unsure, check with sloop::ftype():

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

Unlike most functions, you can't see the source code for most S3 methods just by typing their names. That's because S3 methods are not usually exported: they live only inside the package, and are not available from the global environment. Instead, you can use sloop::s3_get_method(), which will work regardless of where the method lives:

```
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: 0x596bea8>
#> <environment: namespace:stats>
```

Classes

If you have done object-oriented programming in other languages, you may be surprised to learn that S3 has no formal definition of a class: to make an object an instance of a class, you simply set the class attribute. You can do that during creation with structure(), or after the fact with class<-():

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

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

You can determine the class of an S3 object with class(x), and see if an object is an instance of a class using inherits(x, "classname").

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

The class name can be any string, but I recommend using only letters and _. Avoid . because (as mentioned earlier) it can be confused with the . separator between a generic name and a class name. When using a class in a package, I recommend including the package name in the class name. That ensures you won't accidentally clash with a class defined by another package.

S3 has no checks for correctness which means you can change the class of existing objects:

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
print(mod)
#>
#> Call:
#> lm(formula = log(mpg) ~ log(disp), data = mtcars)
#>
#> Coefficients:
#> (Intercept)    log(disp)
#>       5.381       -0.459

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

# Unsurprisingly this doesn't work very well
print(mod)
#> Error in as.POSIXlt.Date(x): (list) object cannot be coerced to type
#> 'double'
```

If you've used other OO languages, this might make you feel queasy, but in practice this flexibility causes few problems. R doesn't stop you from shooting yourself in the foot, but as long as you don't aim the gun at your toes and pull the trigger, you won't have a problem.

To avoid foot-bullet intersections when creating your own class, I recommend that you usually provide three functions:

A low-level constructor, new_myclass(), that efficiently creates new objects with the correct structure.

A validator, validate_myclass(), that performs more computationally expensive checks to ensure that the object has correct values.

A user-friendly helper, myclass(), that provides a convenient way for others to create objects of your class.

You don't need a validator for very simple classes, and you can skip the helper if the class is for internal use only, but you should always provide a constructor.

Constructors

S3 doesn't provide a formal definition of a class, so it has no built-in way to ensure that all objects of a given class have the same structure (i.e. the same base type and the same attributes with the same types). Instead, you must enforce a consistent structure by using a constructor.

The constructor 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.

I'll illustrate these ideas by creating constructors for base classes that you're already familiar with. To start, lets make a constructor for the simplest S3 class: Date. A Date is just a double with a single attribute: its class is "Date". This makes for a very simple constructor:

```
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"
```

The purpose of constructors is to help you, the developer. That means you can keep them simple, and you don't need to optimise error messages for public consumption. If you expect users to also create objects, you should create a friendly helper function, called class_name(), which I'll describe shortly.

A slightly more complicated constructor is that for difftime, which is used to represent time differences. It is again built on a double, but has a units attribute that must take one of a small set of values:

```
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
```

The constructor is a developer function: it will be called in many places, by an experienced user. That means it's OK to trade a little safety in return for performance, and you should avoid potentially time-consuming checks in the constructor.