MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450 Advanced R programming

Week 7: Object-oriented programming (part 2)

`arp.numbat.space`

Outline

Outline

1. Assignments
2. Programming paradigms
3. Object oriented programming

Assignment 2

- Questions?
- Due 19 April 2024

Outline

Programming paradigms

Functional programming (W5)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Programming paradigms

Functional programming (W5)

- Functions are created and used like any other object.
- Output should only depend on the function’s inputs.

Object-oriented programming (W6-W7)

- Functions are associated with object types.
- Methods of the same ‘function’ produce object-specific output.

Programming paradigms

Literate programming (W7)

- Natural language is interspersed with code.
- Aimed at prioritising documentation/comments.
- Now used to create reproducible reports/documents.

Programming paradigms

Literate programming (W7)

- Natural language is interspersed with code.
- Aimed at prioritising documentation/comments.
- Now used to create reproducible reports/documents.

Reactive programming (W8)

- Objects are expressed using code based on inputs.
- When inputs change, the object's value updates.

Outline

Object oriented programming

S3

- The OO system used by most of CRAN.
- Very simple (and ‘limited’) compared to other systems.

Object oriented programming

S3

- The OO system used by most of CRAN.
- Very simple (and 'limited') compared to other systems.

vctrs

- Builds upon S3 to make creating vectors easier.
- Good practices inherited by default.

S3 Recap: Objects and methods

Unlike most OO systems where methods belong to **objects/data**, S3 methods *belong* to ‘generic’ **functions**.

Recall that functions in R are objects like any other.

S3 Recap: Objects and methods

Unlike most OO systems where methods belong to **objects/data**, S3 methods *belong* to 'generic' **functions**.

Recall that functions in R are objects like any other.

Self awareness

In S3, there is no concept of 'self' since the relevant objects are available as function arguments.
However S3 is self-aware of registered methods, allowing `NextMethod()` to call the S3 method of the inherited class.

S3 Recap: S3 dispatch

To use S3, we call the generic function (e.g. `plot()`).

```
plot
```

```
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x55d378d131c8>
<environment: namespace:base>
```

S3 Recap: S3 dispatch

This function looks at the inputs and dispatches (uses) the appropriate method for the input variable class/type.

```
stats:::plot.density
```

```
function (x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
    zero.line = TRUE, ...)
{
    if (is.null(xlab))
        xlab <- paste("N =", x$n, "  Bandwidth =", formatC(x$bw))
    if (is.null(main))
        main <- sub("[.]default", "", deparse(x$call))
    plot.default(x, main = main, xlab = xlab, ylab = ylab, type = type,
        ...)
    if (zero.line)
        abline(h = 0, lwd = 0.25, col = "gray")
    invisible(NULL)
```

S3 Recap: S3 dispatch

If there isn't a registered method for the object, the default method for the generic will be used.

```
graphics:::plot.default
```

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
    panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,
    ...)
{
    localAxis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)
    localBox <- function(..., col, bg, pch, cex, lty, lwd) box(...)
    localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)
    localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)
    xlabel <- if (!missing(x))
        deparse1(substitute(x))
```

Creating an S3 generic

S3 generics are work like any ordinary function, but they include `UseMethod()` which calls the appropriate method.

Your turn!

Create an S3 generic called "reverse".
This function will reverse objects. For example,

- `reverse("stressed")` becomes `"desserts"`,
- `reverse(7919)` becomes `9197`,
- `reverse(1.9599)` becomes `9959.1`.

Writing S3 methods

An S3 method is an ordinary function with some constraints:

- The function's name is of the form `<generic>.<class>`,
- The function's arguments match the generic's arguments,
- The function is registered as an S3 method (for packages).

This looks like:

```
#' Documentation for the method
#' @method <generic> <class>
<generic>.<class> <- function(<generic args>, <method args>, ...) {
  # The code for the method
}
```

Writing S3 methods

Your turn!

Write methods for reversing `character`, `integer`, and `double` objects.

- `reverse("stressed")` becomes `"desserts"`,
- `reverse(7919L)` becomes `9197L`,
- `reverse(1.9599)` becomes `9959.1`.

Hint: `stringi::stri_reverse()` will reverse a string.

The `integer` and `double` methods should return an integer and double respectively.

Writing S3 defaults

What if we tried to reverse the current date; `reverse(Sys.Date())`?

Writing S3 defaults

What if we tried to reverse the current date; `reverse(Sys.Date())`?

Your turn!

Question: what should the default behaviour be?

Writing S3 defaults

What if we tried to reverse the current date; `reverse(Sys.Date())`?

> 🔥 Your turn!
>
> Question: what should the default behaviour be?

- Raise an error?
- Return a reversed string?
- Something else entirely?

Creating your own S3 objects

S3 methods are *(mostly)* dispatched based on the `class()`.

```r
class("stressed")
```

```
[1] "character"
```

```r
class(7919L)
```

```
[1] "integer"
```

```r
class(1.9599)
```

```
[1] "numeric"
```

Creating your own S3 objects

To create an S3 object, we add a class to an object.

This is usually done with `structure()`, for example:

```
e <- structure(list(numerator = 2721, denominator = 1001), class = "fraction")
e
```

```
$numerator
[1] 2721

$denominator
[1] 1001

attr(,"class")
[1] "fraction"
```

Creating your own S3 objects

The `structure()` function is usually only used within other functions made for end-users. For example,

- `lm()` returns a list with class `"lm"`, and
- `tibble()` returns a data.frame (list) with classes `"tbl_df"`, `"tbl"`, and `"data.frame"`.

Creating your own S3 objects

The `structure()` function is usually only used within other functions made for end-users. For example,

- `lm()` returns a list with class `"lm"`, and
- `tibble()` returns a data.frame (list) with classes `"tbl_df"`, `"tbl"`, and `"data.frame"`.

> **Your turn!**
>
> Create `fraction()`, which returns `fraction` objects. This function should check that the inputs are suitable

Creating your own S3 objects

The `fraction` class doesn't yet have any methods, so it inherits methods from its list type.

Usually we would create a method for printing S3 objects.

```
print.fraction <- function(x, ...) {
  paste(x$numerator, x$denominator, sep = "/")
}
e
```

```
2721/1001
```

Creating your own S3 objects

Your turn!

Create a `reverse()` method for the `fraction` object class, which inverts the numerator and denominator.

Finished early?
Write a method for converting a `fraction` into a number.

Creating your own S3 vectors (with vctrs)

The *vctrs* package is helpful for creating custom vectors.

It is built upon S3, so the same approach for creating S3 generics and S3 methods also applies to vctrs.

> **S3 or vctrs?**
>
> - Regular S3 is useful for creating singular objects
> - vctrs is useful for creating vectorised objects

Creating your own S3 vectors (with vctrs)

Why vctrs?

vctrs simplifies the complicated parts in creating vectors

- easy subsetting
- nice printing
- predictable recycling
- casting / coercion
- tidyverse compatibility

Examples of vctrs packages

Lots of vctrs including:

- IP addresses
- Spatial geometries
- Time
- uncertainty

https://github.com/krlmlr/awesome-vctrs

Some packages I’ve made that use vctrs

- distributional

  Distributions of various shapes in vectors
- mixtime

  Time points/intervals of various granularities in vectors
- graphvec

  Graph factors, storing graph edges between levels.
- fabletools

  Custom data frames ‘mable’, ‘fable’, and ‘dable’.

Creating a new vctr

The basic way to produce a vctr is with `vctrs::new_vctr()`.

Just like `structure()`, you provide an object and its new class.

```
attendance <- vctrs::new_vctr(c(80, 70, 75, 50), class = "percent")
attendance
```

```
<percent[4]>
[1] 80 70 75 50
```

Creating a new vctr

As with S3, functions provide ways for users to create vectors.

```
percent <- function(x) {
  vctrs::new_vctr(x, class = "percent")
}
attendance <- percent(c(80, 70, 75, 50))
attendance
```

```
<percent[4]>
[1] 80 70 75 50
```

Creating a new vctr

Don't forget to check the inputs, vctrs provides helpful functions to make this easier and provide informative errors.

```
percent <- function(x) {
  vctrs::vec_assert(x, numeric())
  vctrs::new_vctr(x, class = "percent")
}
percent("80%")
```

```
Error in `percent()`:
! `x` must be a vector with type <double>.
Instead, it has type <character>.
```

Creating a new vctr

It's useful to provide default arguments in this function which creates a length 0 vector (similar to how empty vectors are created with `numeric()` and `character()`).

```
percent <- function(x = numeric()) {
  vctrs::vec_assert(x, numeric())
  vctrs::new_vctr(x, class = "percent")
}
percent()
```

```
<percent[0]>
```

Creating a new vctr

While vctrs provides a nice `print` method, we need to specify how our vector should be formatted.

```
format.percent <- function(x, ...) {
  paste0(vctrs::vec_data(x), "%")
}
attendance
```

```
<percent[4]>
[1] 80% 70% 75% 50%
```

The rcrd type

A special type of vctr is a record (rcrd).

A record is a list containing equal length vectors, and its size is the length its vectors rather than its list.

> **Record indexing**
>
> Usually in R, indexing happens across the list. With the record type, indexing happens within the list's vectors.

The rcrd type

Length of a data frame

Usually the length of data refers to the number of rows, but in R it is the number of columns since it is a list.

```
length(mtcars)
```

```
[1] 11
```

In vctrs, data is a record so we get the number of rows.

```
vctrs::vec_size(mtcars)
```

```
[1] 32
```

Creating a new rcrd

A record is created with the `vctrs::new_rcrd()` function.

```
wallet <- vctrs::new_rcrd(
  list(amt = c(10, 38), unit = c("AU$", "¥")), class = "currency"
)
format.currency <- function(x, ...) {
  paste0(vctrs::field(x, "unit"), vctrs::field(x, "amt"))
}
wallet
```

```
<currency[2]>
[1] AU$10 ¥38
```

Creating a new rcrd

Your turn!

Rewrite the `fraction()` function to use the rcrd data type.

You will also need to update the methods:

- Obtain the numerator and denominator with `field()`.
- Replace the `print` method with a `format` method.
- Remove the `print.fraction` method with `rm()`.

The list_of type

`list_of()` vectors require list elements to be the same type.

It can be created with `list_of()`, or more easily converted to with `as_list_of()`. It behaves identically to `new_vctr()`.

```
vctrs::as_list_of(list(80, 70, 75, 50), .ptype = numeric())
```

```
<list_of<double>[4]>
[[1]]
[1] 80

[[2]]
[1] 70

[[3]]
[1] 75
```

Prototypes

Notice the `.ptype` when we used `as_list_of()`?

`ptype` is shorthand for prototype, which is a size-0 vector.

> **Prototype attributes!**
>
> Prototypes contains all relevant attributes of the object, such as class, dimension, and levels of factors.

Prototypes

Obtain prototypes of a vector with `vctrs::vec_ptype()`.

```r
vctrs::vec_ptype(1:10)
```

```
integer(0)
```

```r
vctrs::vec_ptype(rnorm(10))
```

```
numeric(0)
```

```r
vctrs::vec_ptype(factor(letters))
```

```
factor()
Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```r
vctrs::vec_ptype(attendance)
```

```
<percent[0]>
```

vctr, rcrd, or list_of?

Your turn!

What's better? The `vctr` type or `list_of`?

vctr, rcrd, or list_of?

Your turn!

What's better? The `vctr` type or `list_of`?

It depends! If your vector is based on…

- a single atomic vector (like `percent`) then `vctr`,
- two or more atomic vectors (like `fraction`), then `rcrd`,
- complicated objects (like `lm`), then `list_of`.

That's it! You have created a new vector for R!

Time to celebrate with a break!

Ask questions, try using your new vector in various ways.

Methods for vctrs

While our new vectors looks pretty and fits right in with our tidy tibbles, it isn't very useful yet.

Adding features

Since vctrs is built upon S3, the same approach for creating generic functions and methods applies to vctrs.

Methods for vctrs

While our new vectors looks pretty and fits right in with our tidy tibbles, it isn't very useful yet.

Adding features

Since vctrs is built upon S3, the same approach for creating generic functions and methods applies to vctrs.

However there are also some important **vector specific methods** which should be written to improve usability.

(Proto)typing

We saw earlier how R coerces vectors of different types.

```r
c("desserts", 10)
```

```
[1] "desserts" "10"
```

```r
c(pi, 0L)
```

```
[1] 3.14 0.00
```

```r
c(-1, TRUE, FALSE)
```

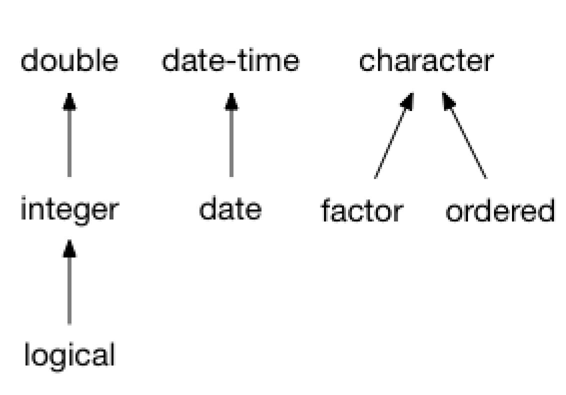
```
[1] -1  1  0
```

```r
c(1, Sys.Date())
```

```
[1]     1 19850
```

(Proto)typing

When combining or comparing vectors of different types, R will (usually) *coerce* to the 'richest' type.



(Proto)typing

vctrs doesn’t make any assumptions about how to coerce your vector, and instead raises an error.

```
library(vctrs)
vec_c(attendance, 0.8)
```

```
Error in `vec_c()`:
! Can't combine `..1` <percent> and `..2` <double>.
```

(Proto)typing

We can specify what the common ('richest') type is by writing `vctrs::vec_ptype2()` methods.

```r
#' @export
vec_ptype2.percent.double <- function(x, y, ...) {
  percent() # Prototype since this produces size-0
}
vctrs::vec_ptype2(attendance, 0.8)
```

```
<percent[0]>
```

```r
vctrs::vec_ptype2(0.8, attendance)
```

```
Error:
! Can't combine `0.8` <double> and `attendance` <percent>.
```

(Proto)typing

Common typing uses *double-dispatch*.

We need to define the common type in both directions.

```
#' @export
vec_ptype2.double.percent <- function(x, y, ...) {
  percent() # Prototype since this produces size-0
}
vctrs::vec_ptype2(attendance, 0.8)
```

```
<percent[0]>
```

```
vctrs::vec_ptype2(0.8, attendance)
```

```
<percent[0]>
```

(Proto)typing

Your turn!

Write methods that define the common (proto)type between `fraction` and `double` as `fraction -> double`.

Double dispatch

Unfortunately `c()` from base R can't (yet) be changed to support double-dispatch with S3. Usually this isn't a problem,

```
c(attendance, attendance)
```

```
<percent[8]>
[1] 80% 70% 75% 50% 80% 70% 75% 50%
```

```
c(attendance, 0.8)
```

```
<percent[5]>
[1] 80%  70%  75%  50%  0.8%
```

Double dispatch

but if your class isn't used in the first argument…

```r
c(0.8, attendance)
```

```
[1]  0.8 80.0 70.0 75.0 50.0
```

… your common (proto)type will be ignored!

Double dispatch

vctrs uses double dispatch when needed, and using `vctrs::vec_c()` fixes many coercion problems in R.

```
vctrs::vec_c(0.8, attendance)
```

```
<percent[5]>
[1] 0.8% 80%  70%  75%  50%
```

```
vctrs::vec_c(1, Sys.Date())
```

```
Error in `vctrs::vec_c()`:
! Can't combine `..1` <double> and `..2` <date>.
```

Double dispatch

Double dispatch inheritence

Double dispatch in vctrs doesn’t work with inheritance and so:

- `NextMethod()` can’t be used
- Default methods aren’t inherited/used.

Casting and coercion

> **!** Converting percentages
>
> Notice earlier how combining percentages with numbers gave the incorrect result?
> This is because we haven't written a method for converting numbers into percentages.

The `vctrs::vec_cast()` generic is used to convert/coerce ('cast') one type into another. Time to write more methods!

Casting and coercion

`vctrs::vec_cast()` also uses double dispatch.

```
vec_cast.double.percent <- function(x, to, ...) {
  vec_data(x)/100
}
vec_cast.percent.double <- function(x, to, ...) {
  percent(x*100)
}

vec_cast(0.8, percent())
```

```
<percent[1]>
[1] 80%
```

```
vec_cast(percent(80), double())
```

```
[1] 0.8
```

Casting and coercion

With both `vec_ptype2()` and `vec_cast()` methods for percentages and doubles it is now possible to combine them.

```
vctrs::vec_c(0.8, attendance)
```

```
<percent[5]>
[1] 80% 80% 70% 75% 50%
```

We can also use coercion to easily perform comparisons.

```
attendance > 0.7
```

```
[1]  TRUE FALSE  TRUE FALSE
```

Casting and coercion

Your turn!

Write a method for casting from a `fraction` to a `double`.

Does this work with `as.numeric()`?

Math and arithmetic

Methods also need to be written for math and arithmetic.

`vec_math()` implements mathematical functions like

```
mean(attendance)
```

```
<percent[1]>
[1] 68.75%
```

`vec_arith()` implements arithmetic operations like

```
attendance + percent(0.1)
```

```
Error in `vec_arith()`:
! <percent> + <percent> is not permitted
```

Math and arithmetic

Since attendance is a simple numeric, the default `vec_math` method works fine. The default `vec_math` function is essentially:

```
vec_math.percent <- function(.fn, .x, ...) {
  out <- vec_math_base(.fn, .x, ...)
  vec_restore(out, .x)
}
```

1. Apply the math to the underlying numbers
2. Restore the percentage class

Math and arithmetic

Unlike double dispatch in `vec_ptype2()` and `vec_cast()`, we currently need to implement our own secondary dispatch for `vec_arith()`.

```
vec_arith.percent <- function(op, x, y, ...) {
  UseMethod("vec_arith.percent", y)
}
vec_arith.percent.default <- function(op, x, y, ...) {
  stop_incompatible_op(op, x, y)
}
```

Math and arithmetic

Then we can create methods for arithmetic.

```
vec_arith.percent.percent <- function(op, x, y, ...) {
  out <- vec_arith_base(op, x, y)
  vec_restore(out, to = percent())
}
percent(40) + percent(20)
```

```
<percent[1]>
[1] 60%
```

Math and arithmetic

Then we can create methods for arithmetic.

```
vec_arith.percent.numeric <- function(op, x, y, ...) {
  out <- vec_arith_base(op, x, vec_cast(y, percent()))
  vec_restore(out, to = percent())
}
percent(40) + 0.3
```

```
<percent[1]>
[1] 70%
```

```
0.3 + percent(40)
```

```
Error in `vec_arith()`:
! <double> + <percent> is not permitted
```

Math and arithmetic

Then we can create methods for arithmetic.

```
vec_arith.numeric.percent <- function(op, x, y, ...) {
  out <- vec_arith_base(op, vec_cast(x, percent()), y)
  vec_restore(out, to = percent())
}
percent(40) + 0.3
```

```
<percent[1]>
[1] 70%
```

```
0.3 + percent(40)
```

```
<percent[1]>
[1] 70%
```

Math and arithmetic

Your turn!

Add support for math and arithmetic for the `fraction` class.

Hint: cast your fraction to a double and then use the base math/arith function, returning a double is fine.

Finished early?

Try to extend `vec_arith()` so that it retains the `fraction` class for `+`, `-`, `*`, `/` operations.