Nectric

WOMBAT 2024: Advanced R Tips & Tricks

Quirky R

workshop.nectric.com.au/advr-wombat24

Outline

Outline

Hello!

I’m Mitch!

I make lots of R packages, and teach lots of people!

Hello!

I'm Mitch!

I make lots of R packages, and teach lots of people!

Among other things…

- PhD candidate at Monash University
- Data consulting and workshops at Nectric
- Specialised in time series analysis
- Develops R packages (fable, vitae, etc.)
- Grows all the things (hobby permaculturist)

Workshop materials

Are all on the website:

https://workshop.nectric.com.au/advr-wombat24/

Workshop materials

Are all on the website:

https://workshop.nectric.com.au/advr-wombat24/

Here you'll find…

- these slides
- demonstrated code
- video recordings
- everything you'll need (for the workshop)

Today's goals (very ambitious!)

1. Understand (and embrace) the quirks of using R
2. 'Appreciate' how 'helpful' R tries to be
3. Use vctrs to avoid common problems with vectors
4. Learn functional programming
5. Write code that writes and runs code (metaprogramming)
6. Use non-standard evaluation for code design

Expectations

1. Follow the code of conduct
2. Be kind and respectful
3. Ask relevant questions any time
4. General Q&A during breaks
5. Make mistakes and learn!

Expectations

1. Follow the code of conduct
2. Be kind and respectful
3. Ask relevant questions any time
4. General Q&A during breaks
5. Make mistakes and learn!

Ask lots of questions!

We'll have the most fun exploring the depths of R together.

The first question

The first question

Your turn!

Why are you here?
What motivates you to learn ‘advanced R’ tips and tricks?

The first question

Your turn!

Why are you here?
What motivates you to learn ‘advanced R’ tips and tricks?

- improve your analysis code?

The first question

Your turn!

Why are you here?
What motivates you to learn ‘advanced R’ tips and tricks?

- improve your analysis code?
- make better R packages?

The first question

Your turn!

Why are you here?
What motivates you to learn ‘advanced R’ tips and tricks?

- improve your analysis code?
- make better R packages?
- something else?

Outline

R is weird!

Featured in Kelly Bodwin's useR! 2024 keynote "Keep R weird".

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1
- everything is a vectors (there are no scalars)

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling
- *functional* programming design

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling
- *functional* programming design
- lazy and non-standard evaluation

R is weird!

Most software developers (of other languages) are ***SHOCKED*** when they see all the 'weird' behaviour of R.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling
- *functional* programming design
- lazy and non-standard evaluation
- *lets you do* ***anything***

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often ‘helpful’ for data analysis.

- indexing from 1

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often ‘helpful’ for data analysis.

- indexing from 1
- everything is a vectors (there are no scalars)

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often ‘helpful’ for data analysis.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often ‘helpful’ for data analysis.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often 'helpful' for data analysis.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling
- *functional* programming design

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often 'helpful' for data analysis.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling
- *functional* programming design
- lazy and non-standard evaluation

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often 'helpful' for data analysis.

- indexing from 1
- everything is a vectors (there are no scalars)
- NA (missing values)
- object types, casting, recycling
- *functional* programming design
- lazy and non-standard evaluation
- *lets you do* ***anything***

R is QUIRKY!

I prefer to think of R as ***quirky***.

These quirks are often ‘helpful’ for data analysis.

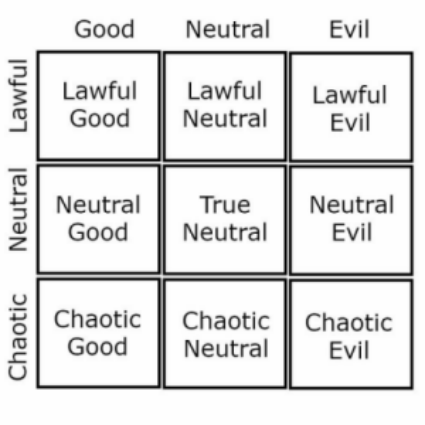
> **!** R’s ‘help’ can hurt!
>
> Unlike stricter languages, sometimes R’s helpful nature can cause *nasty* programming problems.

Workshop content

There’s a lot of fun things I can show you about R…

Workshop content

There's a lot of fun things I can show you about R...



Workshop content

! Chaotic evil

We can explore the ‘dark side’ and produce truly evil code…

Workshop content

❗ Chaotic evil

We can explore the ‘dark side’ and produce truly evil code…

💡 Lawful good

Or create lovely code which effortlessly solves problems.

The dark side

The dark side

R let's you do *almost* **anything**!

The dark side

R let's you do *almost* **anything**!

This includes (figuratively) shooting yourself in the foot.

The dark side

R let's you do *almost* **anything**!

This includes (figuratively) shooting yourself in the foot.

- active bindings

The dark side

R let's you do *almost* **anything**!

This includes (figuratively) shooting yourself in the foot.

- active bindings
- changing R itself

The dark side

R let's you do *almost* **anything**!

This includes (figuratively) shooting yourself in the foot.

- active bindings
- changing R itself
- https://github.com/romainfrancois/evil.R/

The dark side

R let's you do *almost* **anything**!

This includes (figuratively) shooting yourself in the foot.

- active bindings
- changing R itself
- https://github.com/romainfrancois/evil.R/
- `attach(structure(list(), class = "UserDefinedDatabase"))`

Workshop content

Today we'll learn **useful** tips and tricks for R.

- Avoid common mistakes
- Use powerful features

Workshop content

Today we'll learn **useful** tips and tricks for R.

- Avoid common mistakes
- Use powerful features

This workshop will focus on three R-centric topics:

- Vectorisation
- Functional programming
- Non-standard evaluation

Workshop content

Textbook reference

Much more Advanced R can be found in Hadley Wickham's Advanced R book. It's freely available online here: https://adv-r.hadley.nz/

Outline

Vectorisation

R's design around vectors is perfect for data.

Vectors are objects which store data (several datum) together.

Your turn!

What *types* of vectors ('data') do we have?

Types of vectors

There are two types of vectors in R:

- Atomic (single-type)
- List (mixed-type)

Types of vectors

Your turn!

Which of the following vectors are ‘atomic’ in R?

- Random numbers
- Today’s date
- A dataset (data.frame)
- A matrix
- $\sqrt{-1}$ (a complex number)
- `NULL`

Subsetting vectors: `x[i]`

```
letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
# What's the 13th letter?
letters[13L]
```

```
[1] "m"
```

```
# What's the last letter?
letters[length(letters)]
```

```
[1] "z"
```

Subsetting vectors: `x[i]`

Remember: indexing starts at 1!

```
letters[0L]
```

```
character(0)
```

Subsetting vectors: `x[i]`

Remember: indexing starts at 1!

```
letters[0L]
```

```
character(0)
```

Negative indices

Remember: R is weird!

```
letters[-1L]
```

```
 [1] "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
[20] "u" "v" "w" "x" "y" "z"
```

Subsetting vectors: `x[i]`

```
# What's the first three letters?
letters[1:3]
```

```
[1] "a" "b" "c"
```

Subsetting vectors: `x[i]`

```
# What's the first three letters?
letters[1:3]
```

```
[1] "a" "b" "c"
```

Safe sequences

Using `1:n` is unsafe in general code. `seq_len(n)` is safer.

```
# What's the first 'zero' letters?
n <- 0
letters[1:n]
```

```
[1] "a"
```

```
letters[seq_len(n)]
```

```
character(0)
```

Subsetting vectors: `x[i, j]`

When subsetting matrices (or arrays) we use multiple indices.

```
# Get the first row and third column
volcano[1L,3L]
```

```
[1] 101
```

Subsetting vectors: `x[i, j]`

When subsetting matrices (or arrays) we use multiple indices.

```
# Get the first row and third column
volcano[1L,3L]
```

```
[1] 101
```

Subsetting simplification

By default R will simplify matrices/arrays into 1-d vectors. It's often safer to prevent this with `drop = FALSE`.

Subsetting vectors: `x[i, j]`

```
# What's the first column?
volcano[,1L]
```

```
 [1] 100 101 102 103 104 105 105 106 107 108 109 110 110 111 114 116 118 120 120
[20] 121 122 122 123 124 123 123 120 118 117 115 114 115 113 111 110 109 108 108
[39] 107 107 107 108 109 110 111 111 112 113 113 114 115 115 114 113 112 111 111
[58] 112 112 112 113 114 114 115 115 116 116 117 117 116 114 112 109 106 104 102
[77] 101 100 100  99  99  99  99  98  98  97  97
```

```
# But with keeping the matrix
# (empty arguments for positioning is also quirky!)
volcano[,1L,drop=FALSE]
```

```
     [,1]
 [1,]  100
 [2,]  101
 [3,]  102
 [4,]  103
```

Subsetting vectors: `x[[i]]`

Your turn!

What's the difference between x[i] and x[[i]]?

This code gives the same result...

```
letters[13L]
```

```
[1] "m"
```

```
letters[[13L]]
```

```
[1] "m"
```

Subsetting (list) vectors: `x[[i]]`

`x[[i]]` is used to subset (list) vectors into their element's type.

Key differences:

- Only works for single indices `i`
- Drops the (list) structure of `x`

```
Orange[2L]
```

```
   age
1  118
2  484
3  664
4 1004
5 1231
6 1372
7 1582
```

```
Orange[[2L]]
```

```
 [1]  118  484  664 1004 1231 1372 1582  11
[16]  484  664 1004 1231 1372 1582  118  48
[31]  664 1004 1231 1372 1582
```

Subsetting (list) vectors: `x$col`

Often we use the list vector's names for subsetting.

```
Orange$age
```

```
 [1]  118  484  664 1004 1231 1372 1582  118  484  664 1004 1231 1372 1582  118
[16]  484  664 1004 1231 1372 1582  118  484  664 1004 1231 1372 1582  118  484
[31]  664 1004 1231 1372 1582
```

This also works for `x[["col"]]`.

```
Orange[["age"]]
```

```
 [1]  118  484  664 1004 1231 1372 1582  118  484  664 1004 1231 1372 1582  118
[16]  484  664 1004 1231 1372 1582  118  484  664 1004 1231 1372 1582  118  484
[31]  664 1004 1231 1372 1582
```

Subsetting (list) vectors: `x$col`

Often we use the list vector's names for subsetting.

Your turn!

What happens with the following code?

```
Orange["age"]
Orange["age",]
Orange[,"age"]
```

Subsetting (list) vectors: `x$col`

! Caution! R's eager to please.

`Orange["age",]` should probably error, but it doesn't. There was no rowname called "age", so it gives a 'missing' row.

What does `Orange[NA,]` do?

What about `Orange$a` and `Orange[["a"]]`? What if we also had a column called 'alpine'?

Subsetting (list) vectors: `x$col`

A tibble is stricter than `data.frame` (it also looks nicer).

By being less 'helpful', it is (a bit) safer.

Your turn!

Convert `Orange` into a tibble with `as_tibble()`, then try various subsets.

```
library(dplyr)
orange_trees <- as_tibble(Orange)
orange_trees$a
orange_trees["age",]
orange_trees[NA,]
```

Combining vectors: c(x, y)

Vectors are combined with `c()`, short for ‘combine’.

```
c(1, 2, 3)
```

```
[1] 1 2 3
```

! Confusing combinations

What happens when you combine vectors of different types?
Try it!

Combining vectors: vec_c(x, y)

The vctrs package makes combining vectors much stricter when you use `vec_c()`.

This is used widely in tidyverse packages now, to make data analysis in the tidyverse safer than base R.

> **Your turn!**
>
> Use `vec_c()` from `{vctrs}` to combine different vectors. What works, and what errors (safely)?

Casting vectors: as_*(), vec_cast()

This vector converting process is known as ‘casting’.

Explicit casting with `as.numeric()`, `as.Date()` or `vec_cast()` is good practice.

Casting vectors: as_*(), vec_cast()

This vector converting process is known as ‘casting’.

Explicit casting with `as.numeric()`, `as.Date()` or `vec_cast()` is good practice.

Parsing data from text

It is also safer to explicitly specify column types when reading in data.
The readr package writes this code for you - just copy it!

Recycling: vec_recycle()

What happens when you use two vectors of different length?

```
x <- 1:10
b <- 2
b^x
```

Recycling: vec_recycle()

What happens when you use two vectors of different length?

```
x <- 1:10
b <- 2
b^x
```

```
 [1]    2    4    8   16   32   64  128  256  512 1024
```

So helpful!

R ‘recycles’ `b` to be the same length as `x`.
This aspect of R’s vectorisation is great since we don’t need to write a loop.

Recycling: vec_recycle()

What if we're calculating the revenue of fruit sales...

```
fruit <- c("apple", "banana", "kiwi")
sales <- c(10, 3, 8)
price <- c(2.99, 4.39)
sales*price
```

```
Warning in sales * price: longer object length is not a multiple of shorter
object length
```

```
[1] 29.90 13.17 23.92
```

! Reckless recycling

R 'helpfully' recycles everything, regardless of if their lengths match. At least it warned us something was amiss!

Recycling: vec_recycle()

It is safer to only recycle length 1 vectors, which is done in the tidyverse via `vec_recycle()`. If you're ...

- writing packages recycle safely with `vec_recycle()`.
- undertaking analysis be careful of mismatched vector lengths (using data.frame/tibble helps)

Distribution statistics

The p/d/q/r functions in R are notoriously bad at recycling. My {distributional} package has much safer behaviour.