Tidyevaluation cheat sheet

Brendan Clarke [1]
brendan.clarke2@nhs.scot
[1] *Digital Learning Lead, NHS Education for Scotland*

Data masking

Data masking is the term for the default tidyverse behaviour that allows us to refer to columns by unquoted names. So in tidyverse we can write:

```
ae_attendances %>%
  filter(type == 1)
```

This is an alternative to the base R syntax for specifying columns, with quoted column names in double square brackets:

```
ae_attendances[["type"]]
```

Note that we can use base R syntax inside some tidyverse functions:

```
ae_attendances %>%
    filter(ae_attendances[["type"]]
        == 1)
```

While data masking usually works well, allowing us to write cleaner code, there are some situations where it breaks down. This cheat sheet provides an overview of ways of avoiding data masking problems.

Pronouns

Used to disambiguate which kind of object we are referring to. **.data[[“type”]]** will refers to the *type* column from our data:

```
ae_attendances %>%
  filter(.data[["type"]] == 1)
```

.env[[“type”]] will refer to the *type* environmental variable:

```
type <- 2

ae_attendances %>%
   filter(type == .env[["type"]]) #
        filters cases where the
        type column == 2
```

Pronouns prevent ambiguity, allowing us to work with clarity. This example has 3 objects named *type*:

```
type <- 2 # global .env type = 2


type_org <- function(type, org) {
  ae_attendances %>%
          filter(.data[["type"]]  ==
        .env[["type"]]             &
        .data[["org_code"]]        ==
        .env[["org"]])
}

type_org(1, "RF4")
```

{{var}}

{{var}} takes an unquoted argument, and converts it to a data variable. Most useful for specifying cols from functions:

```
col_greater <- function(col, n) {
  ae_attendances %>%
    filter({{col}} >= n)
}
col_greater(breaches, 5000)
```

Under the surface, *{{var}}* defuses and injects the variable. You can also do the defuse and inject in separate steps:

```
col_greater_steps <- function(col,
        n) {


  new_col <- enquo(col)

  ae_attendances %>%
    filter(!!new_col >= n)
}

col_greater_steps(breaches, 7000)
```

:= (name injection)

= is fussy, and checks that it can evaluate the LHS of an expression. := is less fussy

Use := with embracing to make col names from unquoted args:

```
col_greater_maker <- function(col,
        n) {

  ae_attendances %>%
    select({{col}}) %>%
    filter({{col}} >= n) %>%
     rename("{{col}} over {{n}}" :=
        {{col}})
}
col_greater_maker(attendances,
        5000)
```

Use := with glue syntax to make col names from quoted stuff:

```
col_greater_maker <- function(col,
        n, name) {
  ae_attendances %>%
    select({{col}}) %>%
    filter({{col}} >= n) %>%
     rename("{name} over {{n}}" :=
        {{col}})
}
col_greater_maker(attendances,
        5000, "Bruce")
```

Quasiquotation

Use quasiquotation to deal with difficult cases that mix unquoted and quoted use

This code won’t work:

```
col_quas <- function(col, n) {
  ae_attendances %>%
    summarise(col = sum(col))
}
try(col_quas("attendances"))
```

This code gives the correct value, but the wrong name:

```
col_quas <- function(col, n) {
  ae_attendances %>%
              summarise(col   =
      sum(!!sym(col)))
}
try(col_quas("attendances"))
```

Everything is as it should be. *!!sym(col)* defuses and injects the var:

```
col_quas <- function(col) {
  ae_attendances %>%
        summarise(!!sym(col)  :=
      sum(!!sym(col)))
}
var <- "attendances"
col_quas(var)
```

NHS Education for Scotland