Constructing experimental designs with the `edibble` R-package

Presenter: *Emi Tanaka*

Department of Econometrics and Business Statistics,
Monash University, Melbourne, Australia

emi.tanaka@monash.edu

@statsgen

9 Nov 2021 @ Applications of Statistical Procedures in Biological Data

Table of Contents

1 Experimental design *basics*

2 *Current state* of experimental design tools

3 *Software design* for an everyday user

4 The **grammar of experimental designs** with `edibble`

These slides made using R powered by HTML/CSS/JS can be found at

emitanaka.org/slides/stats4bio2021/edibble

1

Experimental design *basics*

Experiment

Essential scientific endeavors to collect data to explore, understand or verify phenomena.

Experimental data

The gold standard in data collection.

(provided that experimental design is satisfactory)

Comparative experiments

Collecting data to compare the effects of different conditions under a ***controlled environment*** with the goal of drawing generalisable conclusions

Designing comparative experiments

> *... to identify data-collection schemes that achieve sensitivity and specificity requirements despite biological and technical variability, while keeping time and resource costs low.*
>
> *— Krzywinski & Altman (2014)*

Planning the controlled environment such that there is a higher confidence that effects can be attributed to selected conditions

Krzywinski, M., Altman, N. Designing comparative experiments. *Nat Methods* 11, 597–598 (2014). https://doi.org/10.1038/nmeth.2974

Basic terminology in comparative experiments modified versions of Bailey (2008)

A **treatment** $(\mathcal{T})$ is the entire description of the condition applied to an experimental unit.

Experimental unit (Ω) is the smallest unit that the treatment can be independently applied to.

Observational unit (Ω_o) is the smallest unit in which the response will be measured on.

- Not to be confused with responses $\boldsymbol{Y}$.
- May or may not be the same as experimental unit.

Block, also called **cluster**, is the unit that group some other units (e.g. experimental units) such that the units within the same block (cluster) are more alike (homogeneous).

A **design** $(D : \Omega \to \mathcal{T})$ is the allotment of treatments to particular set of units.

A **plan** or **layout** is the design translated into actual units. *Randomisation* is usually involved in the translation process.

Bailey, R. (2008). Design of Comparative Experiments (Cambridge Series in Statistical and Probabilistic Mathematics). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511611483

Experimental structures as defined by Bailey (2008)

Unit structure means meaningful ways of dividing up experimental units (Ω) and observational units (Ω_o).

For example:

- **Unstructured**
- **Blocking**

Treatment structure means meaningful ways of dividing up $\mathcal{T}$.

For example:

- **Unstructured**: no grouping within $\mathcal{T}$
- **Factorial**: all combinations of at least two factors
- **Factorial + control**

Bailey, R. (2008). Design of Comparative Experiments (Cambridge Series in Statistical and Probabilistic Mathematics). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511611483

Unreplicated experiments

- **Experimental units**: 3 cows
- **Observational units**: 3 cows
- **Observation**: milk yield
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows
- **Replication**: 1 each

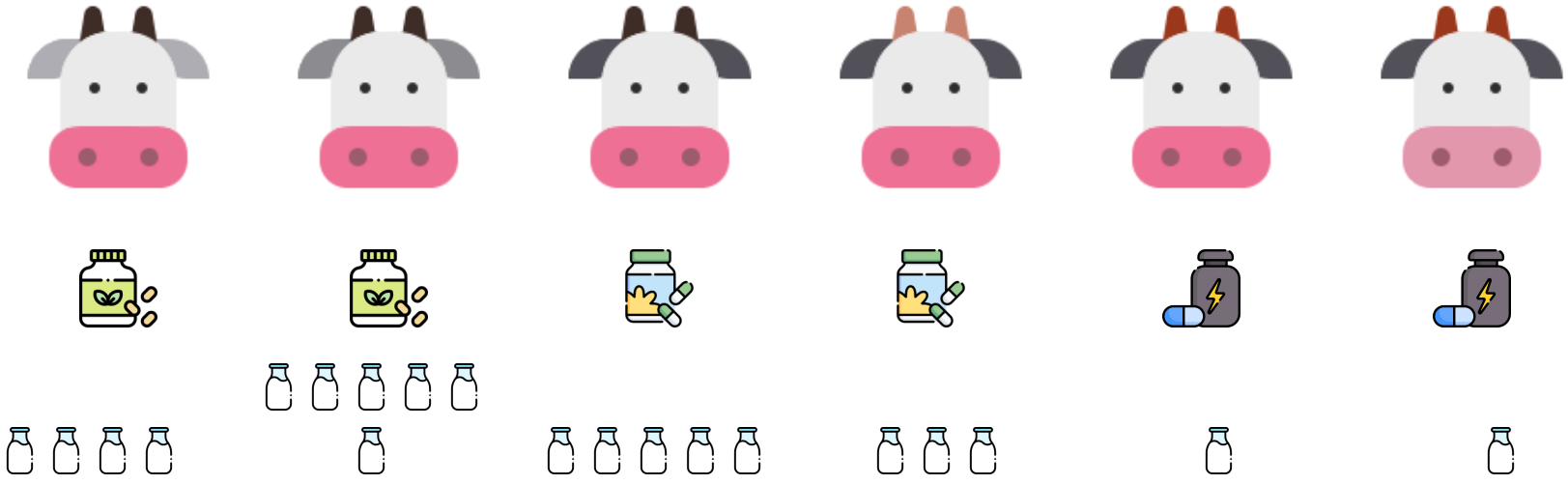
Conclusion: produces most therefore is the most effective supplement for higher milk yield from cows out of the three supplements tested

❓ How confident will you be of this conclusion?

- No individual experimental units are the same

  (with some exceptions)

Treatment replications



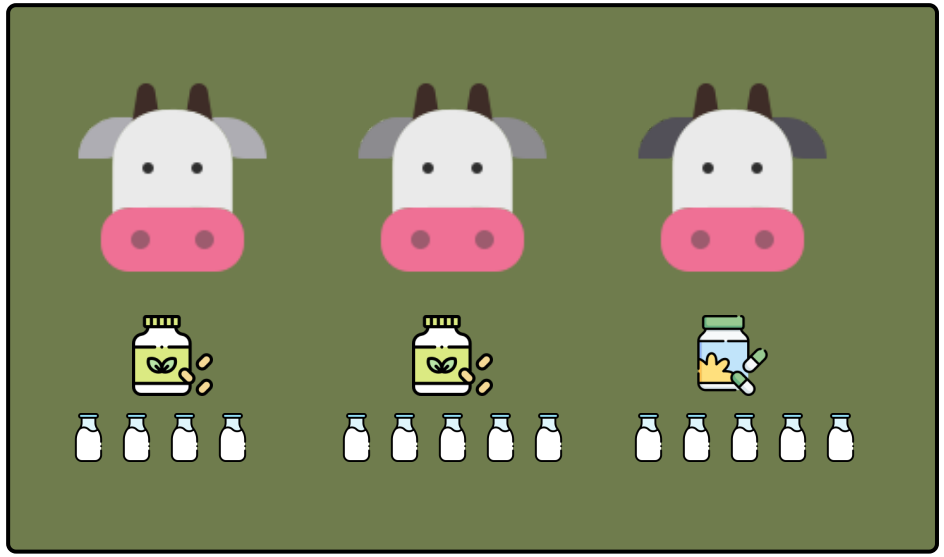
- **Experimental units**: 6 cows
- **Observational units**: 6 cows
- **Observation**: milk yield
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows
- **Replication**: 2 each

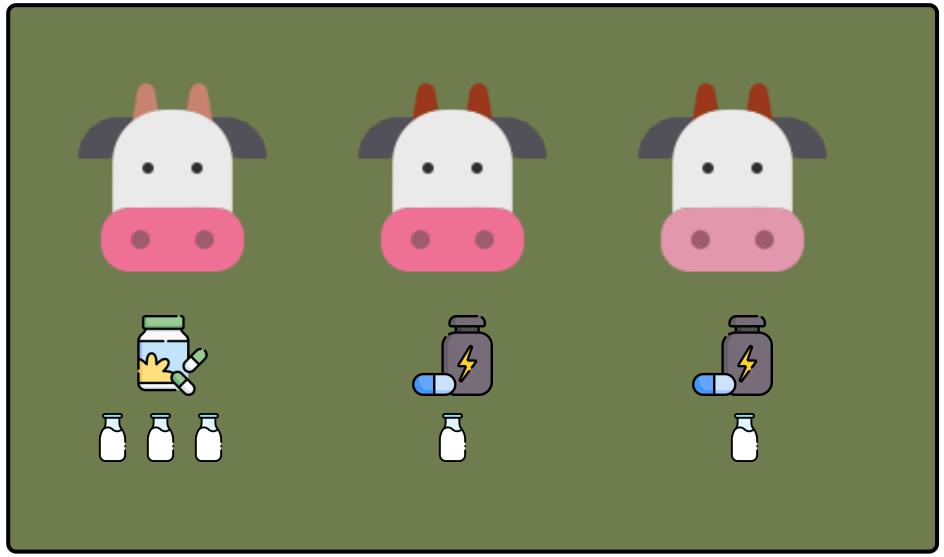
Conclusion: produces most *on average* therefore is the most effective supplement for higher milk yield from cows out of the three supplements tested

❓ How confident will you be of this conclusion now?

- Treatment replications here allow us to estimate experimental unit (or error) variation

Plan ❶ Treatment allocation for nested unit structure



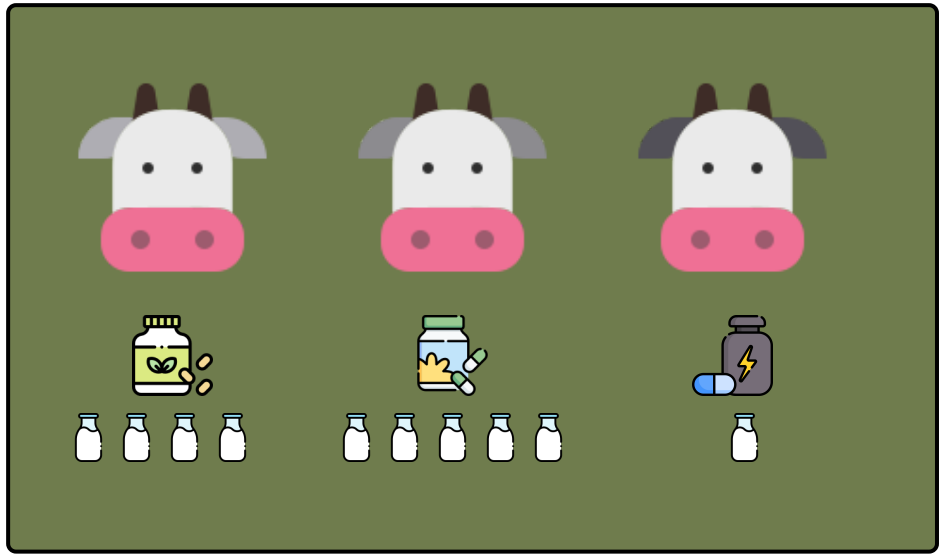


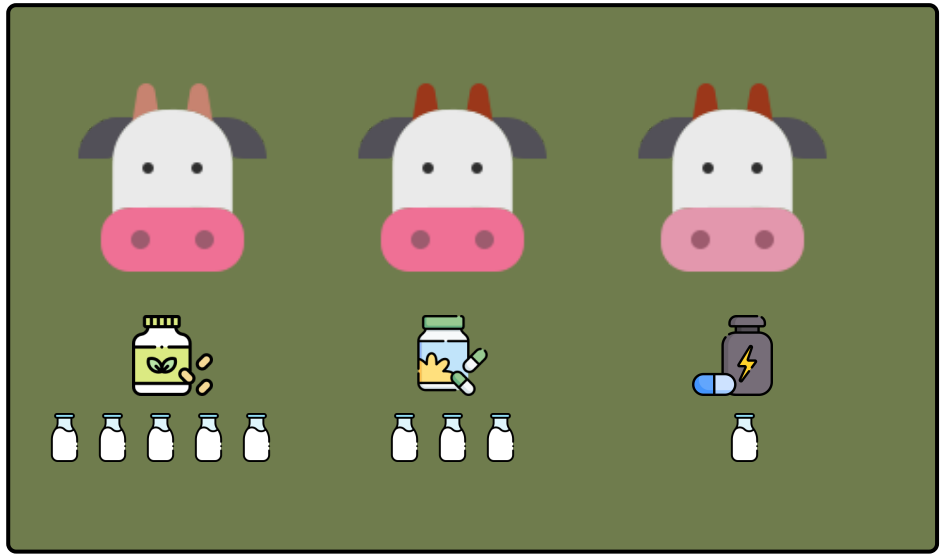
- **Units**: 2 pens with 3 cows each
- **Observational units**: 6 cows
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows

❓ Are the treatment means of say,  and  comparable?

❓ How would you distribute the treatments?

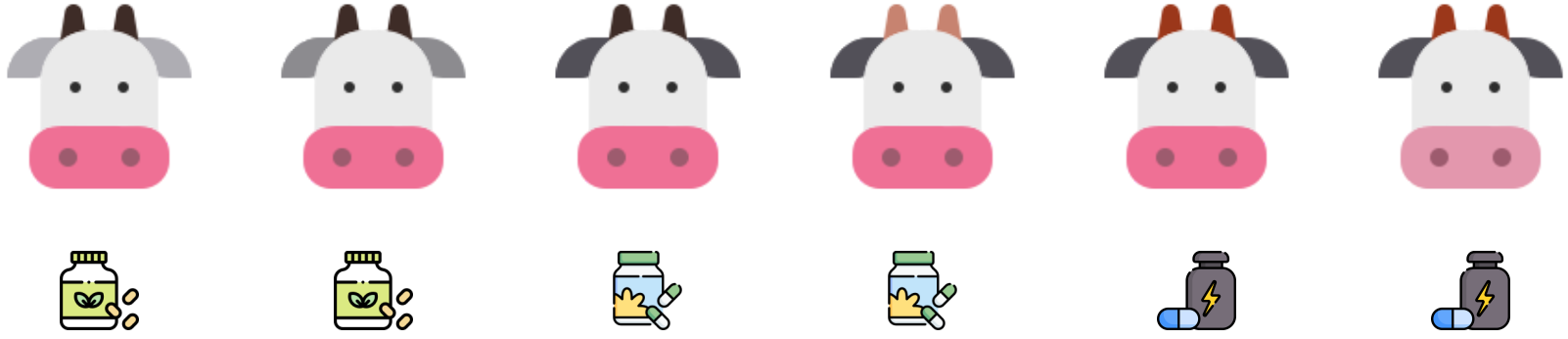
Plan ❷ Treatment allocation for nested unit structure





- Every treatment appears once in each pen
- This is a better design since each treatment appears in every pen so you can be more confident that the treatment means are not due to the conditions of particular pens

Systematic designs



- **Experimental units**: 6 cows
- **Observational units**: 6 cows
- **Observation**: milk yield
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows
- **Replication**: 2 each
- **Assignment**: systematic order

❓ What could go wrong with this?

- The order of the experimental units may be ***confounded*** with some extraneous factor
- Like say, the order of the experimental units was determined by the speed (fast to slow) of the cow to get to the feed
- This means that the more active cows are given  and leasat active ones are given 

Randomised designs

- **Experimental units**: 6 cows
- **Observational units**: 6 cows
- **Observation**: milk yield
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows
- **Replication**: 2 each
- **Assignment**: random order

- Randomisation protects you against bias and potential unwanted confounding with extraneous factors
- Bias comes in many forms: obvious to not-so obvious, known to unknown, and so on.
- Randomisation doesn't mean it'll completely shield you from all biases!
- You can get a systematic order by chance! ❗ This doesn't mean you should keep on randomising your design until get the layout you want! You should instead add another unit structure before randomisation.

Factorial treatment structure ❶

- **Experimental units**: 12 plots
- **Observational units**: 12 plots
- **Observation**: wheat yield
- **Treatments**: combination of:
  - Water: irrigated or rain-fed
  - Fertilizer: type A or type B
- **Allotment**:
  - Water → plots
  - Fertilizer → plots
- **Assignment**: random order

❓ How many treatments are there?

Treatment	Replication
	3
	3
	3
	3

Treatment factor	Count
	6
	6
	6
	6

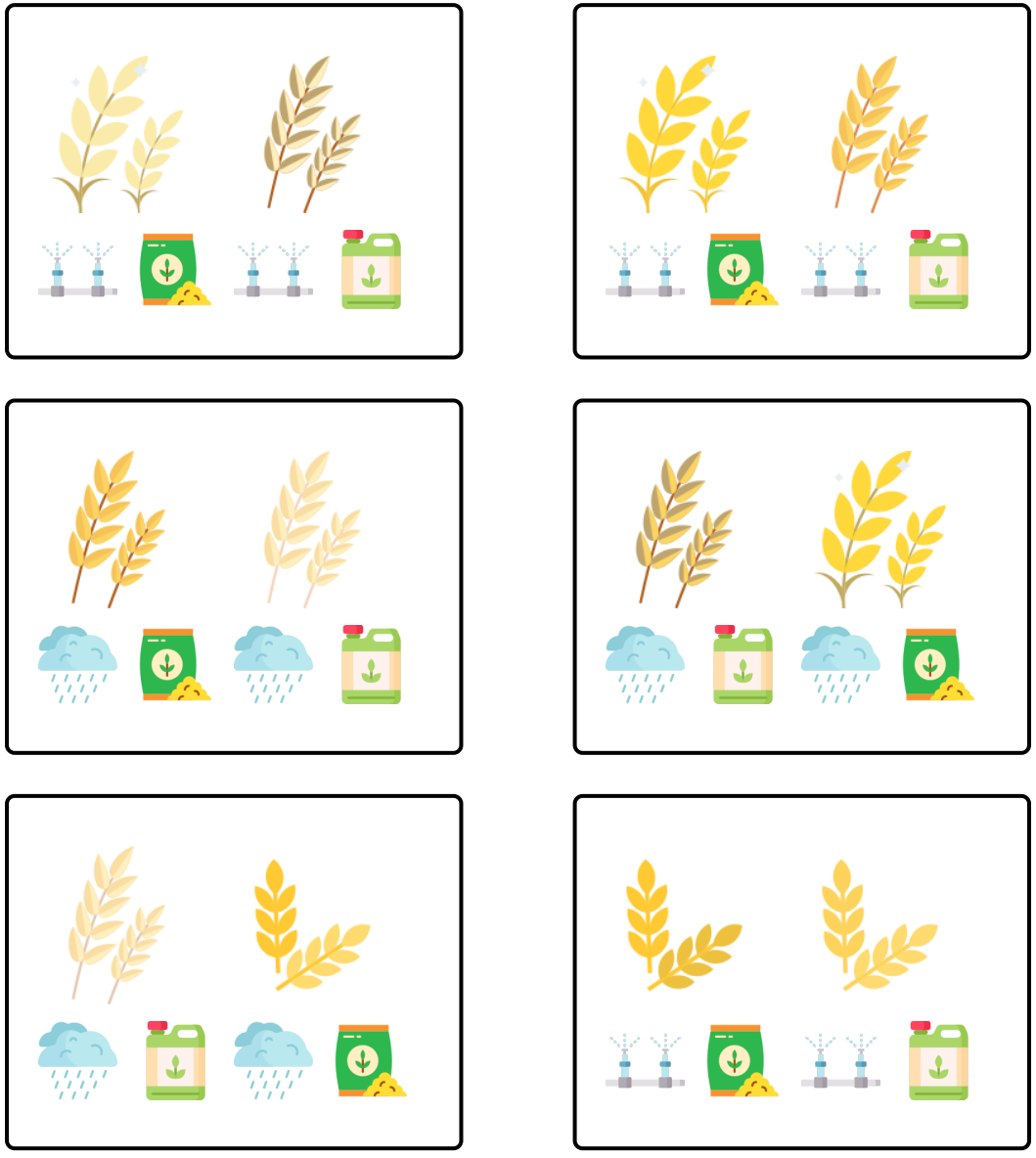
Factorial treatment structure ②

- **Experimental units**: 12 plots
- **Observational units**: 12 plots
- **Observation**: wheat yield
- **Treatments**: combination of:
  - Water: irrigated or rain-fed
  - Fertilizer: type A or type B
- **Allotment**:
  - Water and fertilizer → plots
- **Assignment**: random order

Treatment	Replication
	3
	3
	3
	3

Treatment factor	Count
	6
	6
	6
	6

Factorial treatment structure, nested unit structure, and treatment constraint



- **Units**: 6 strips with 2 plots each
- **Observational units**: 12 plots
- **Observation**: wheat yield
- **Treatments**: combination of:
  - Water: irrigated or rain-fed
  - Fertilizer: type A or type B
- **Allotment**:
  - Water → strip
  - Fertilizer → plot
- **Assignment**: random order

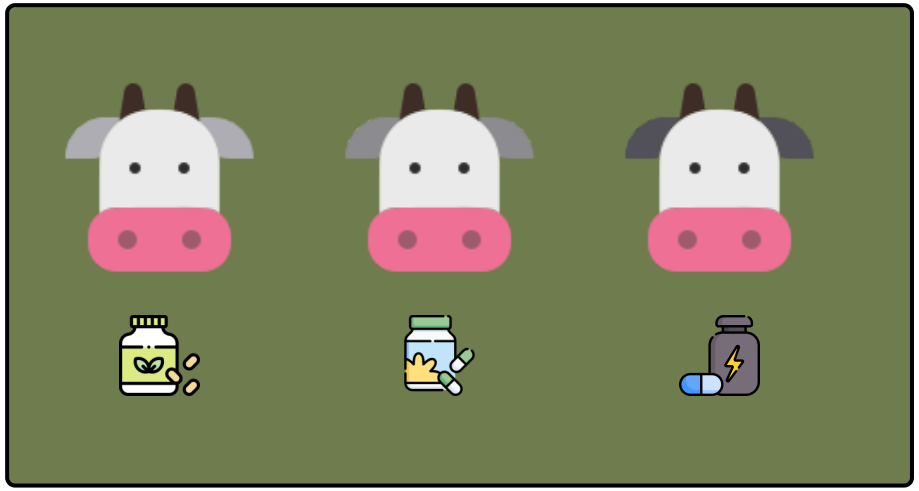
Some classical "named" experimental designs

- **A** Completely Randomised Design
- **B** Randomised Complete Block Design
- **C** Factorial Design
- **D** Split-Plot Design

A Completely Randomised Design (CRD)

- **Experimental units**: 6 cows
- **Observational units**: 6 cows
- **Observation**: milk yield
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows
- **Replication**: 2 each
- **Assignment**: random order

B Randomised Complete Block Design (RCBD)

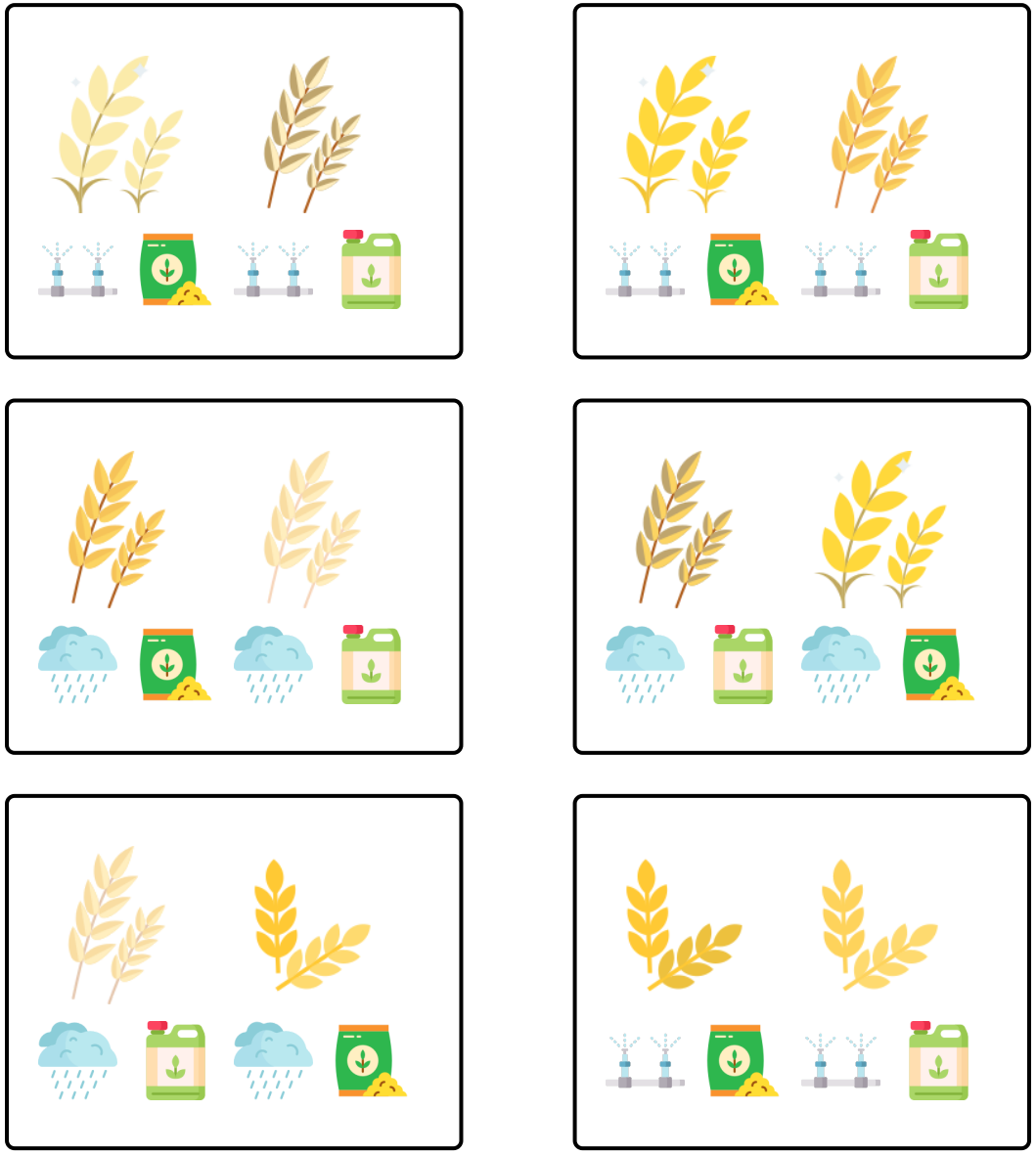


- **Units**: 2 pens with 3 cows each
- **Observational units**: 6 cows
- **Treatments**: 3 types of supplements
- **Allotment**: supplements → cows, with restriction such that each treatment appears once in each pen
- **Assignment**: random order

C Factorial Design

- **Experimental units**: 12 plots
- **Observational units**: 12 plots
- **Observation**: wheat yield
- **Treatments**: combination of:
  - Water: irrigated or rain-fed
  - Fertilizer: type A or type B
- **Allotment**:
  - Water and fertilizer → plots
- **Assignment**: random order

D Split-Plot Design



- **Units**: 6 strips with 2 plots each
- **Observational units**: 12 plots
- **Observation**: wheat yield
- **Treatments**: combination of:
  - Water: irrigated or rain-fed
  - Fertilizer: type A or type B
- **Allotment**:
  - Water → strip
  - Fertilizer → plot
- **Assignment**: random order

2

Current state of experimental design tools

CRAN Task View of Design of Experiments

& Analysis of Experimental Data

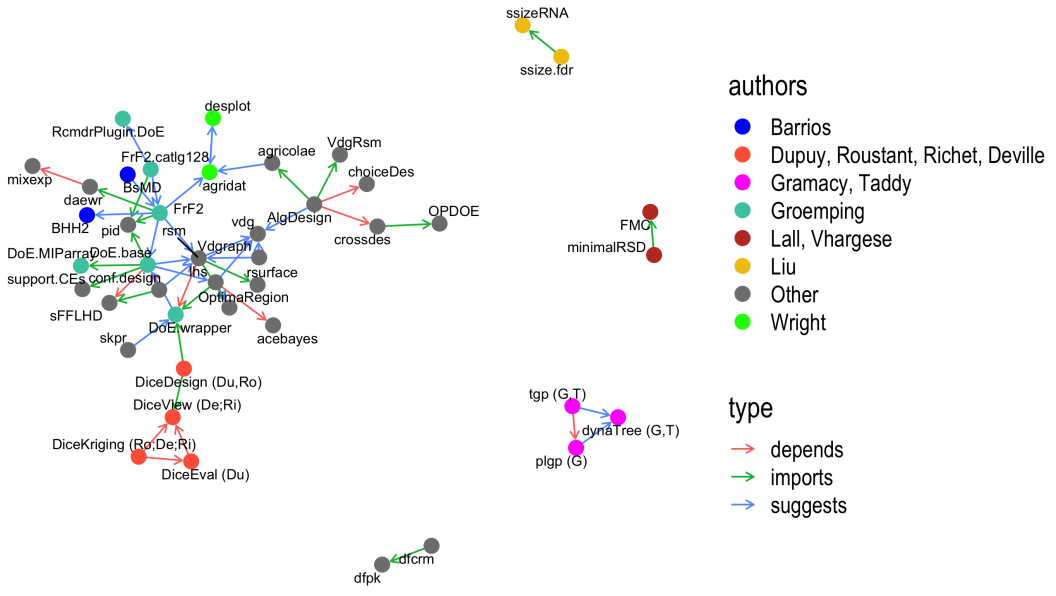
contains

109 R-packages

based on the `ctv` package version 0.8.5

In contrast, only a handful of libraries exist in Python
(namely `pyDOE`, `pyDOE2`, `dexpy`, `experimenter` and `GPdoemd`).

Design of experiments area appear to have the least collaboration

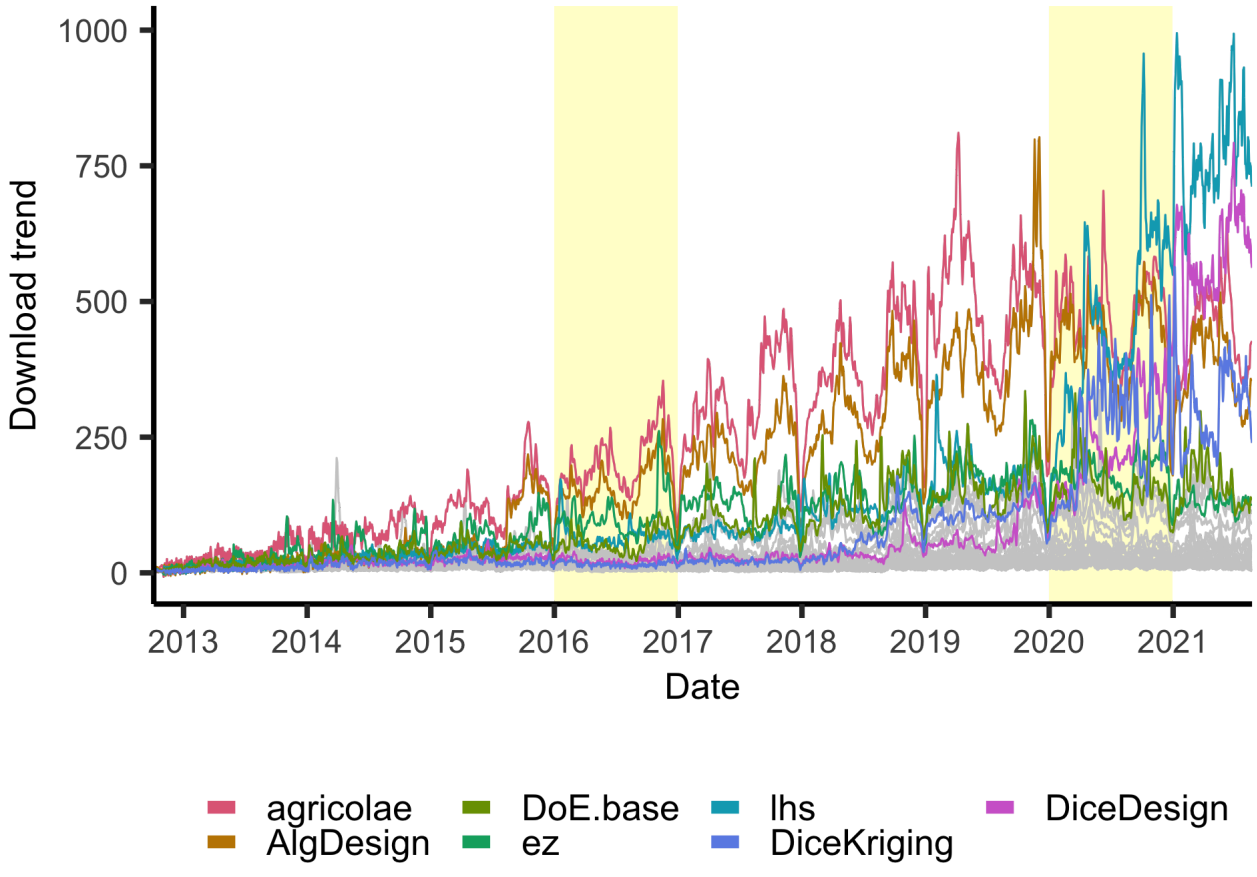


Thanks to Dewi Lestari Amaliah for the graph!

Authors tend to work in silos → limited knowledge sharing across silos perhaps

Topic	# of packages	% of packages connected within topic	Averag # c author
Analysis of Pharmacokinetic Data	18	16.67	3
Hydrological Data and Modeling	96	21.88	3
Design of Experiments (DoE) & Analysis of Experimental Data	109	24.77	2

Top downloaded R-packages in the design of experiments



Top 5 in 2016

Package	Downloads
agricolae	73,521
AlgDesign	57,037
ez	37,488
lhs	23,518
DoE.base	20,651

Top 5 in 2020

Package	Downloads
agricolae	171,813
lhs	165,415
AlgDesign	153,582
DiceKriging	92,287
DiceDesign	88,160

`agricolae` is one of the top downloaded

(total download based on logs from the RStudio CRAN mirror scrubbed by Danyang Dai)

agricolae

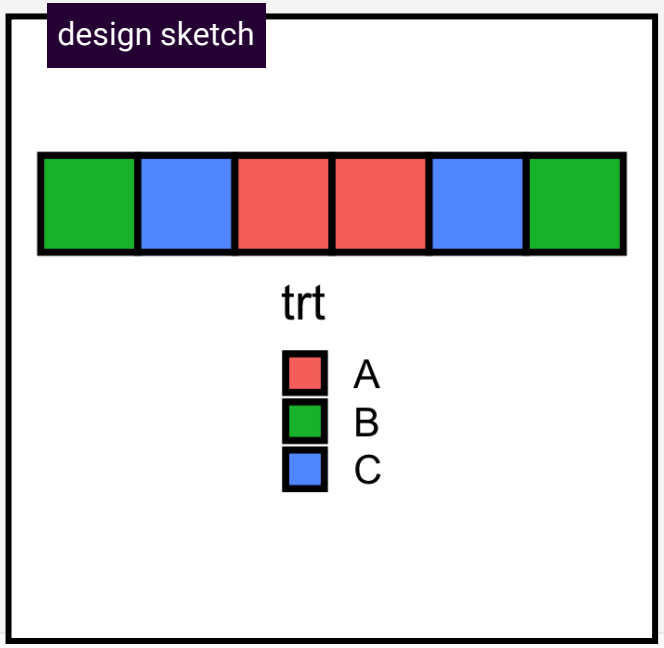
a case of classical named randomised designs

A agricolae::design.crd

Completely randomised design for $t = 3$ treatments with 2 replicates each

```
trt <- c("A", "B", "C")
agricolae::design.crd(trt = trt, r = 2)
```

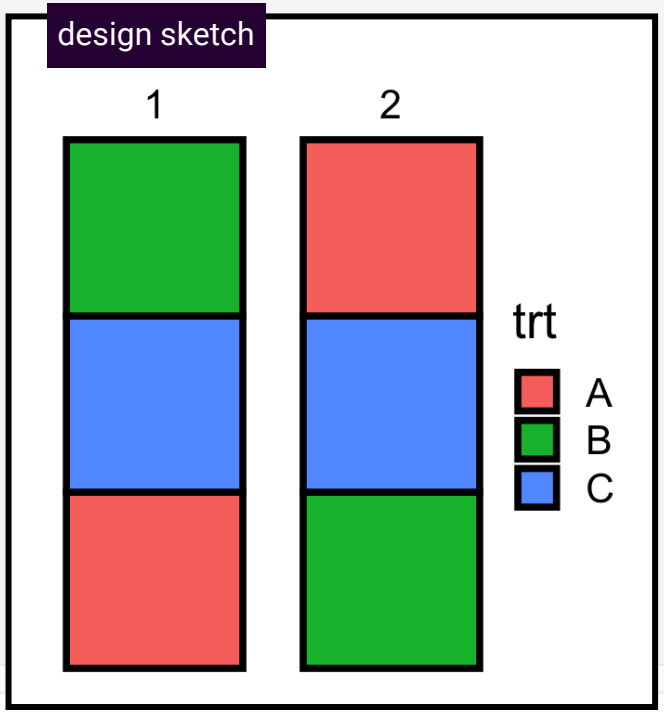
```
## $parameters
## $parameters$design
## [1] "crd"
##
## $parameters$trt
## [1] "A" "B" "C"
##
## $parameters$r
## [1] 2 2 2
##
```



B agricolae::design.rcbd

Randomised complete block design for $t = 3$ treatments with 2 blocks

```
trt <- c("A", "B", "C")
agricolae::design.rcbd(trt = trt, r = 2)

## $parameters
## $parameters$design
## [1] "rcbd"
##
## $parameters$trt
## [1] "A" "B" "C"
##
## $parameters$r
## [1] 2
##
```

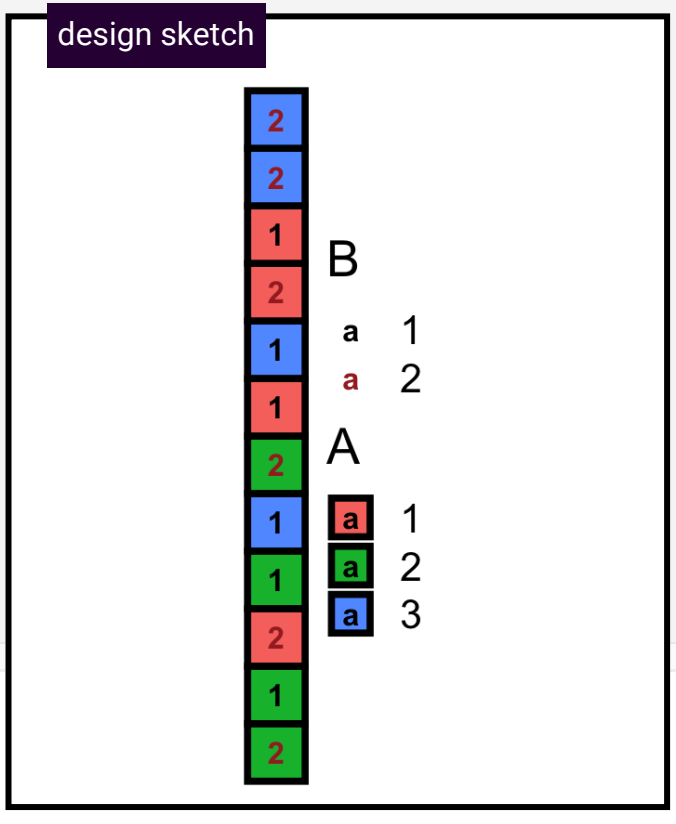


C agricolae::design.ab()

Factorial design for $t = 3 \times 2$ treatments with 2 replication for each treatment

```
agricolae::design.ab(trt = c(3, 2), r = 2, design = "crd")
```

```
## $parameters
## $parameters$design
## [1] "factorial"
##
## $parameters$trt
## [1] "1 1" "1 2" "2 1" "2 2" "3 1" "3 2"
##
## $parameters$r
## [1] 2 2 2 2 2 2
##
```

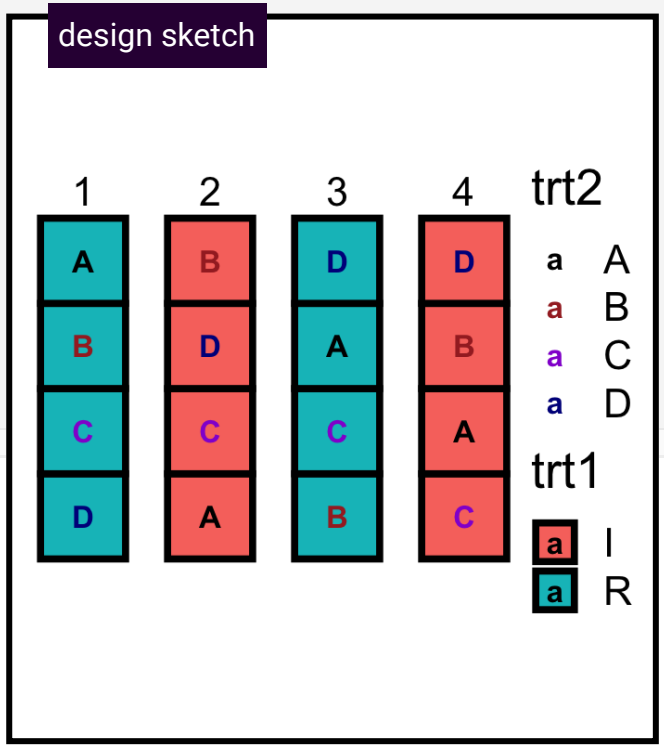


Note *not* A/B testing!

D agricolae::design.split()

Split-plot design for $t = 2 \times 4$ treatments with 2 replication for each treatment

```
trt1 <- c("I", "R"); trt2 <- LETTERS[1:4]
agricolae::design.split(trt1 = trt1, trt2 = trt2, r = 2, design = "crd")

## $parameters
## $parameters$design
## [1] "split"
##
## $parameters[[2]]
## [1] TRUE
##
## $parameters$trt1
```



> *“Good design considers units and treatments first, and then allocates treatments to units. It does not choose from a menu of named designs.*
>
> *—Rosemary Bailey (2008)*

AlgDesign

a case of optimised (model-based) designs

AlgDesign::gen.factorial()

- First, a **helper function** to create the treatment (and replicate) combinations:

```
dat <- AlgDesign::gen.factorial(levels = 3,
                                nVars = 3,
                                center = FALSE,
                                varNames = c("irrigation",
                                             "fertilizer",
                                             "variety"),
                                factors = "all")
dat
```

```
##    irrigation fertilizer variety
## 1           1          1       1
## 2           2          1       1
## 3           3          1       1
## 4           1          2       1
## 5           2          2       1
## 6           3          2       1
```

AlgDesign::optFederov

- Optimum design with 14 trials using Federov's exchange algorithm

```
AlgDesign::optFederov(frml = ~ ., # assume additive effects
                      data = dat,
                      nTrials = 14,
                      criterion = "D")

## $D
## [1] 0.2343815
##
## $A
## [1] 6.25
##
## $Ce
```

AlgDesign::optBlock()

- An optimal design with 3 blocks of size 9.

```
AlgDesign::optBlock(frml = ~ ., # assume additive effects
                    withinData = dat,
                    blocksizes = rep(9, 3),
                    criterion = "D")

## $D
## [1] 0.1924501
##
## $diagonality
## [1] 0.866
##
## $Blocks
```

What were the experiments about?

Context is key in experimental design

Units and allocation are often *implicitly* understood

3

Software design

for an everyday user

Benefits of programming

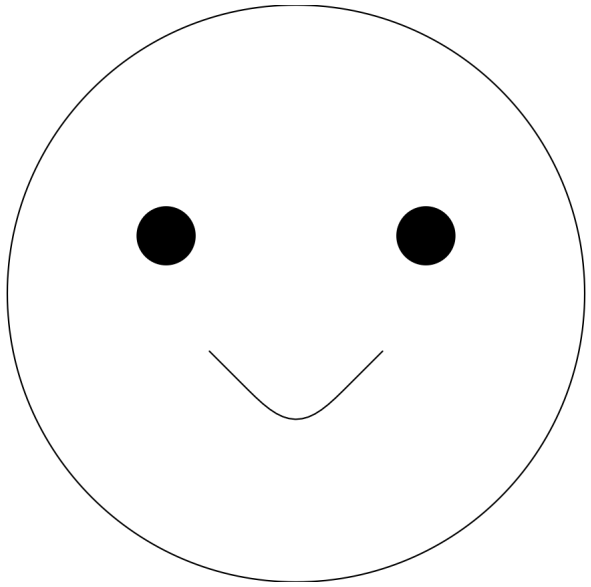
1. Computational reproducibility
2. Allows greater flexibility
3. Can promote higher order thinking *if the software is designed with the user in mind*

Software design for users

- A user interacts with the **software interface**
- The interface design can make a *huge difference to an everyday user*

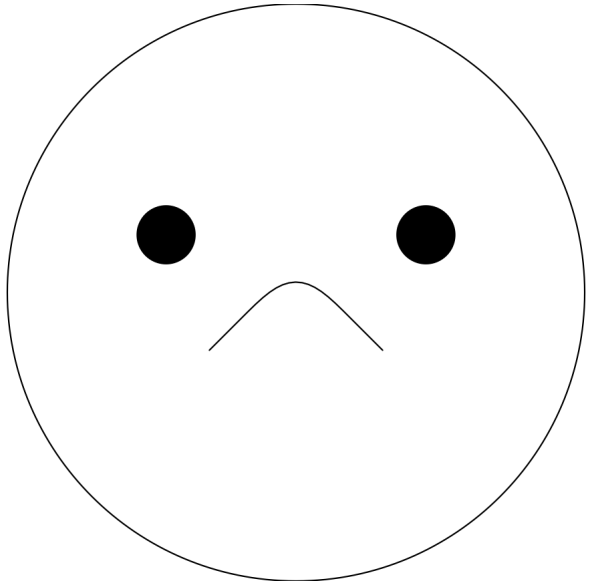
Drawing a happy face

```
library(grid)
# face shape
grid.circle(x = 0.5, y = 0.5, r = 0.5)

# eyes
grid.circle(x = c(0.35, 0.65),
            y = c(0.6, 0.6),
            r = 0.05,
            gp = gpar(fill = "black"))

# mouth
grid.curve(x1 = 0.4, y1 = 0.4,
           x2 = 0.6, y2 = 0.4,
           square = FALSE)
```



Drawing a sad face

```r
library(grid)
# face shape
grid.circle(x = 0.5, y = 0.5, r = 0.5)

# eyes
grid.circle(x = c(0.35, 0.65),
            y = c(0.6, 0.6),
            r = 0.05,
            gp = gpar(fill = "black"))

# mouth
grid.curve(x1 = 0.4, y1 = 0.4,
           x2 = 0.6, y2 = 0.4,
           square = FALSE,
           curvature = -1)
```



Use functions to draw faces

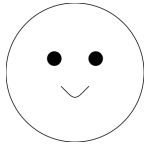
```r
face1 <- function() {
  grid::grid.circle(x = 0.5, y = 0.5, r = 0.5)
  grid::grid.circle(x = c(0.35, 0.65),
                    y = c(0.6, 0.6),
                    r = 0.05,
                    gp = gpar(fill = "black"))
  grid::grid.curve(x1 = 0.4, y1 = 0.4,
                   x2 = 0.6, y2 = 0.4,
                   square = FALSE)
}

face2 <- function() {
  grid::grid.circle(x = 0.5, y = 0.5, r = 0.5)
  grid::grid.circle(x = c(0.35, 0.65),
                    y = c(0.6, 0.6),
                    r = 0.05,
                    gp = gpar(fill = "black"))
  grid::grid.curve(x1 = 0.4, y1 = 0.4,
                   x2 = 0.6, y2 = 0.4,
                   square = FALSE,
                   curvature = -1)
}
```

```r
face1()
```

```r
face2()
```

```r
face1()
```

```r
face1()
```



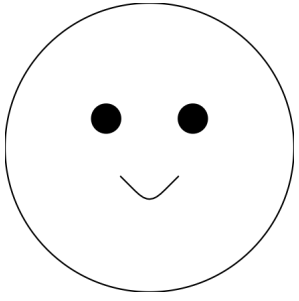
```r
face2()
```

```r
face1()
```

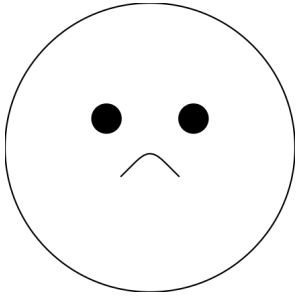
😃 Drawing faces ❸ Human-centered design

Adapt computational systems for human use with syntactic sugar

```
face1()
```



```
face2()
```



```
face3()
```

?

Alternative function names:

```
face_happy()
```

```
face_sad()
```

```
face_angry()
```

❓ Now what do you expect for the output?

- Functions are named after emotions
- Emotion is a surrogate for describing the entire face

What if you want to draw a face that is winking? ☺

... with a grin? 😉

... or with the tongue out? 😜

The differences between facial features are small, but you need an **entire new function** that contains instructions for the *whole* face and a **new function name**.

❓ How would you design the system to draw faces?

Let's reframe how we think

Let's reframe how we think

https://github.com/emitanaka/portrait

```
library(portrait)
```

- Let's ***reframe*** how we think
- A face is made up of:
  - eyes
  - mouth
  - shape

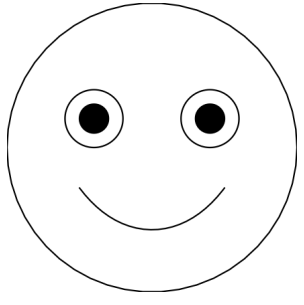
Let's reframe how we think

https://github.com/emitanaka/portrait

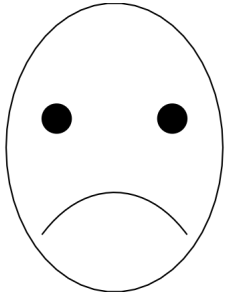
```r
library(portrait)
```

- Let's ***reframe*** how we think
- A face is made up of:
  - eyes
  - mouth
  - shape

```r
face(eyes = "googly",
     mouth = "smile",
     shape = "round")
```



```r
face(eyes = "round",
     mouth = "sad",
     shape = "oval")
```



- We can easily make large number of faces with a single function
- It makes users think about faces based on facial features

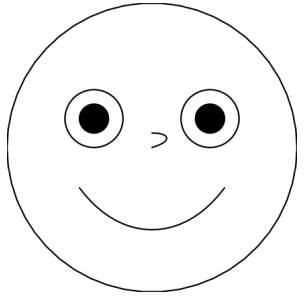
? But what about hair, nose and other facial features?

Let's reframe how we think

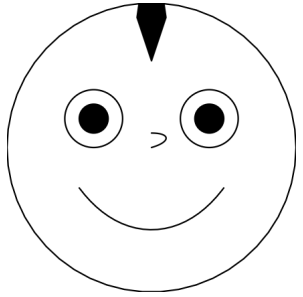
- A face is made up of:
  - eyes
  - mouth
  - shape
  - **hair** 👈
  - **nose** 👈

- Adding more arguments:

```
face(eyes = "googly",
     mouth = "smile",
     shape = "round",
     hair = "none",
     nose = "simple")
```



```
face(eyes = "googly",
     mouth = "smile",
     shape = "round",
     hair = "mohawk",
     nose = "simple")
```

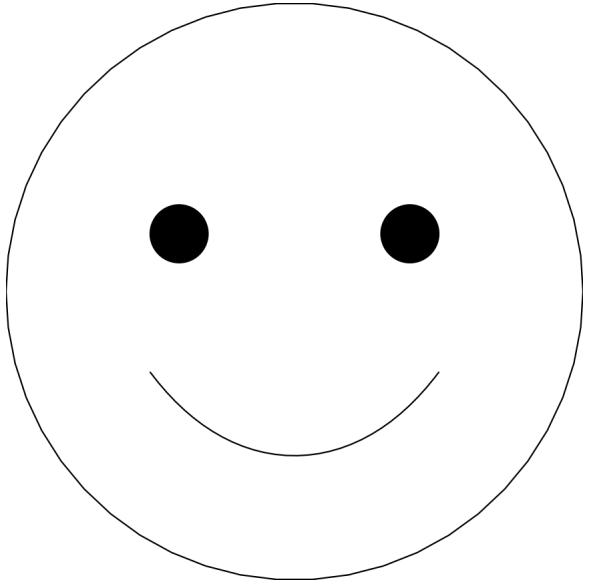


❓ But about other facial features?

Rethink everything as an object

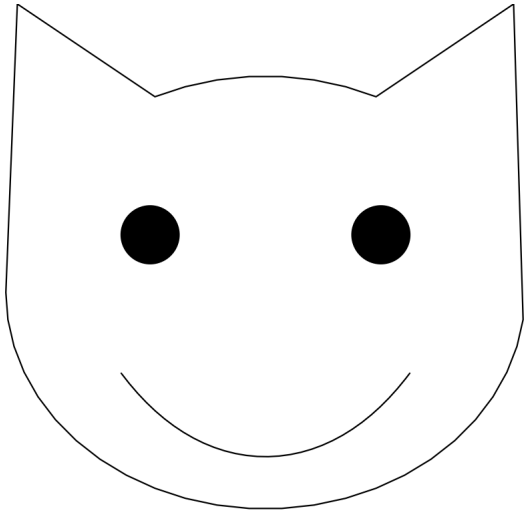
Rethink everything as an object

```r
library(portrait)
face()
```



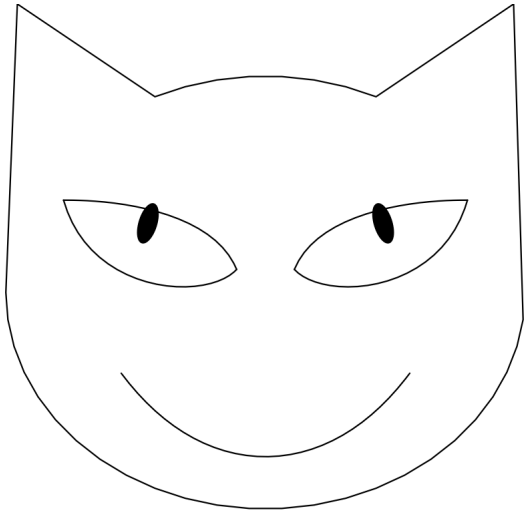
Rethink everything as an object

```
library(portrait)
face() +
  cat_shape()
```



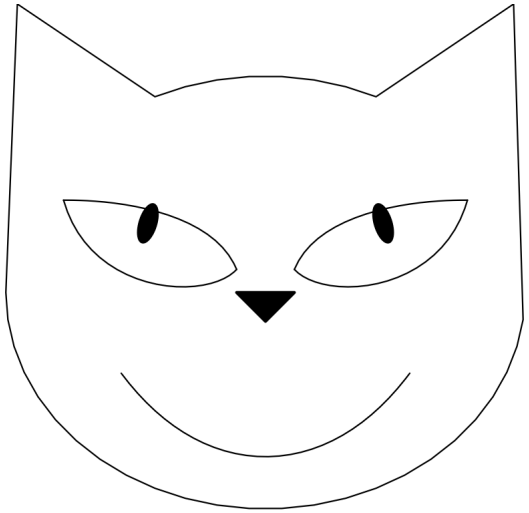
Rethink everything as an object

```
library(portrait)
face() +
  cat_shape() +
  cat_eyes()
```



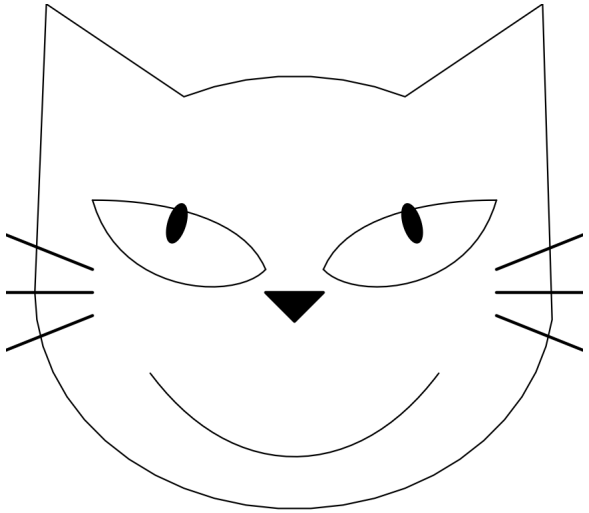
Rethink everything as an object

```
library(portrait)
face() +
  cat_shape() +
  cat_eyes() +
  cat_nose()
```



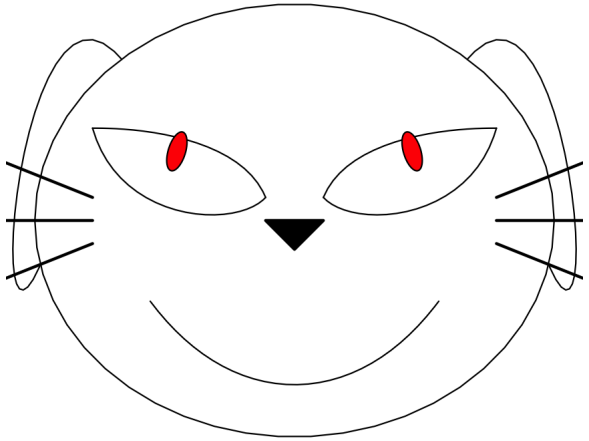
Rethink everything as an object

```
library(portrait)
face() +
  cat_shape() +
  cat_eyes() +
  cat_nose() +
  cat_whiskers()
```



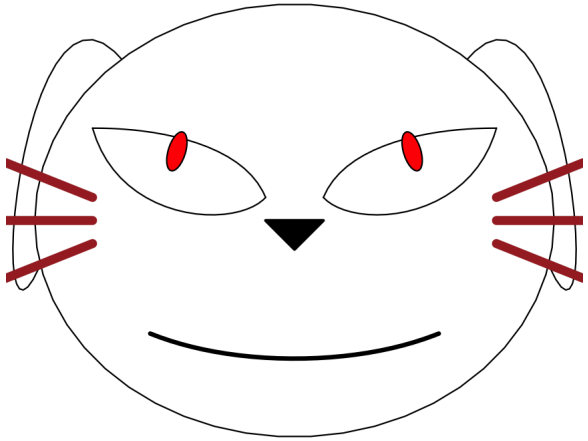
Rethink everything as an object

```
library(portrait)
face() +
  dog_shape() +
  cat_eyes(fill = "red") +
  cat_nose() +
  cat_whiskers()
```



Rethink everything as an object

```
library(portrait)
face() +
  dog_shape() +
  cat_eyes(fill = "red") +
  cat_nose() +
  cat_whiskers(size = 6,
               color = "brown") +
  sketch_mouth(smile = 0.3,
               size = 3)
```



Software design for everyday users

1. Imperative instructions for the computer ➞ more work for humans
2. Recipe functions ➞ One function to draw one complete face
3. Syntactic syntax ➞ Make it easier for humans to read code
4. A function with multiple arguments ➞
   One function to draw multiple complete faces
5. Finite number of functions to draw
   infinite possible *incomplete* and complete faces

The tool ***you choose*** to use can **enforce a certain way of thinking** and may restrict you on what you can do.

4

The grammar of experimental designs with `edibble`

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment")
```

```
My experiment
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4)
```

```
My experiment
└─wholeplot (4 levels)
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4) %>%
  set_units(subplot = nested_in(wholeplot, 2))
```

```
My experiment
└─wholeplot (4 levels)
  └─subplot (8 levels)
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4,
            subplot = nested_in(wholeplot, 2))
```

```
My experiment
└─wholeplot (4 levels)
  └─subplot (8 levels)
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4,
            subplot = nested_in(wholeplot, 2)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B"))
```

```
My experiment
├─wholeplot (4 levels)
│  └─subplot (8 levels)
├─water (2 levels)
└─fertilizer (2 levels)
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  set_units(wholeplot = 4,
            subplot = nested_in(wholeplot, 2))
```

```
My experiment
├─water (2 levels)
├─fertilizer (2 levels)
└─wholeplot (4 levels)
  └─subplot (8 levels)
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_trts(water = c("irrigated", "rainfed")) %>%
  set_units(wholeplot = 4) %>%
  set_trts(fertilizer = c("A", "B")) %>%
  set_units(subplot = nested_in(wholeplot, 2))
```

```
My experiment
├─water (2 levels)
├─wholeplot (4 levels)
│  └─subplot (8 levels)
└─fertilizer (2 levels)
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4,
            subplot = nested_in(wholeplot, 2)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  allot_trts(water ~ wholeplot,
             fertilizer ~ subplot)
```

```
My experiment
├─wholeplot (4 levels)
│  └─subplot (8 levels)
├─water (2 levels)
└─fertilizer (2 levels)
Allotment:
• water ~ wholeplot
• fertilizer ~ subplot
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4,
            subplot = nested_in(wholeplot, 2)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  allot_trts(water ~ wholeplot,
             fertilizer ~ subplot) %>%
  assign_trts("random", seed = 1)
```

```
My experiment
├─wholeplot (4 levels)
│  └─subplot (8 levels)
├─water (2 levels)
└─fertilizer (2 levels)
Allotment:
• water ~ wholeplot
• fertilizer ~ subplot
Assignment: random
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("My experiment") %>%
  set_units(wholeplot = 4,
            subplot = nested_in(wholeplot, 2)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  allot_trts(water ~ wholeplot,
             fertilizer ~ subplot) %>%
  assign_trts("random", seed = 1) %>%
  serve_table()
```

```
# An edibble: 8 x 4
  wholeplot   subplot     water fertilizer
  <unit(4)> <unit(8)>  <trt(2)>   <trt(2)>
1 wholeplot1  subplot1 irrigated          A
2 wholeplot1  subplot2 irrigated          B
3 wholeplot2  subplot3 irrigated          A
4 wholeplot2  subplot4 irrigated          B
5 wholeplot3  subplot5 rainfed            B
6 wholeplot3  subplot6 rainfed            A
7 wholeplot4  subplot7 rainfed            B
8 wholeplot4  subplot8 rainfed            A
```

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("Modified design") %>%
  set_units(block = 2,
            subplot = nested_in(block, 4)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  allot_trts(water:fertilizer ~ subplot) %>%
  assign_trts("random", seed = 1) %>%
  serve_table()
```

```
# An edibble: 8 x 4
      block   subplot     water fertilizer
  <unit(2)> <unit(8)>  <trt(2)>   <trt(2)>
1    block1  subplot1 irrigated          A
2    block1  subplot2 irrigated          B
3    block1  subplot3 rainfed            B
4    block1  subplot4 rainfed            A
5    block2  subplot5 rainfed            A
6    block2  subplot6 irrigated          B
7    block2  subplot7 rainfed            B
8    block2  subplot8 irrigated          A
```

- The resulting design is what we call "randomised complete block design"

The *grammar of experimental design* with `edibble`

```r
library(edibble)
start_design("Modified design") %>%
  set_units(block = 2,
            subplot = nested_in(block, 4)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  allot_trts(water:fertilizer ~ subplot) %>%
  assign_trts("random", seed = 1) %>%
  set_rcrds_of(subplot = c("yield", "disease"),
               block = "manager") %>%
  serve_table()
```

```
# An edibble: 8 x 7
     block   subplot     water fertilizer  yield disease manager
  <unit(2)> <unit(8)>  <trt(2)>   <trt(2)> <rcrd>  <rcrd>  <rcrd>
1    block1  subplot1 irrigated          A      ■       ■       ■
2    block1  subplot2 irrigated          B      ■       ■       x
3    block1  subplot3 rainfed            B      ■       ■       x
4    block1  subplot4 rainfed            A      ■       ■       x
5    block2  subplot5 rainfed            A      ■       ■       ■
6    block2  subplot6 irrigated          B      ■       ■       x
7    block2  subplot7 rainfed            B      ■       ■       x
8    block2  subplot8 irrigated          A      ■       ■       x
```

- The functions are reminiscent of the fundamental experimental terminology ▸

The *grammar of experimental design* with `edibble`

```r
out <- start_design("Modified design") %>%
  set_units(block = 2,
            subplot = nested_in(block, 4)) %>%
  set_trts(water = c("irrigated", "rainfed"),
           fertilizer = c("A", "B")) %>%
  allot_trts(water:fertilizer ~ subplot) %>%
  assign_trts("random", seed = 1) %>%
  set_rcrds_of(subplot = c("yield", "disease"),
               block = "manager") %>%
  expect_rcrds(yield = to_be_numeric(with_value(">=", 0)),
               disease = to_be_factor(levels = c("none", "moderate",  "severe"))
  serve_table()
```

```r
export_design(out, file = "design-layout.xlsx", overwrite = TRUE)
```

- The exported file has data validation features embedded

There are more (not-well documented) features in `edibble`

More on those on Thursday!

- Our understanding of experimental design is **growing** and so the tool should **evolve** with better understanding
- The idea for `edibble` was conceived early 2019, the code base was released publicly on 31st Dec 2020.
- Since its initial public realease, underlying structure in `edibble` has evolved drastically for the better
- The development of a good tool is a community effort so...

Get in touch!

- The purpose of `edibble` is to help you plan experiments better
- `edibble` gets better with feedback

  Slides: emitanaka.org/slides/stats4bio2021/edibble

  Package documentation: edibble.emitanaka.org

  Source code: github.com/emitanaka/edibble

  emi.tanaka@monash.edu @statsgen
- Feature requests or issues with `edibble`? Submit or upvote here: github.com/emitanaka/edibble/issues, send me an email or tell me now!