## Prepare your work folder

```
# Update the Materials folder to the latest version of our GitHub
cd your-path/Programming/Materials
git pull

# Create a Week-6 folder in Assignments
cd your-path/Programming/Assignments
mkdir Week-6
cd Week-6

# Copy the contents of Materials/Week-6 into Assignments/Week-6
cp -R ../../Materials/Week-6/* .
```

# Expyriment: Counterbalancing

**Programming Psychology Experiments (CORE-1)**

Barbu Revencu & Maxime Cauté

Session 6 | 15 October 2025

## The plan for today

1. **Assignment** discussion
2. **Program your first experiment**
3. **Counterbalancing**

# Assignment 3 Discussion

# blindspot.py

```python
INSTRUCTION_TEMPLATE = """
While looking at the cross with your {eye_closed} eye closed, adjust the circle's
position (using your keyboard arrows) and size (1: make smaller, 2: make bigger) until
you can no longer see it.\n
When the circle becomes invisible, press SPACE.\n
Press any key to begin.
"""


def make_instructions(eye):
    eye_closed = "left" if eye == "right" else "right"
    instructions = stimuli.TextScreen(
            text=INSTRUCTION_TEMPLATE.format(eye_closed=eye_closed),
            text_justification=0, heading="Instructions"
        )
    instructions.preload()
    return instructions
```

# blindspot.py

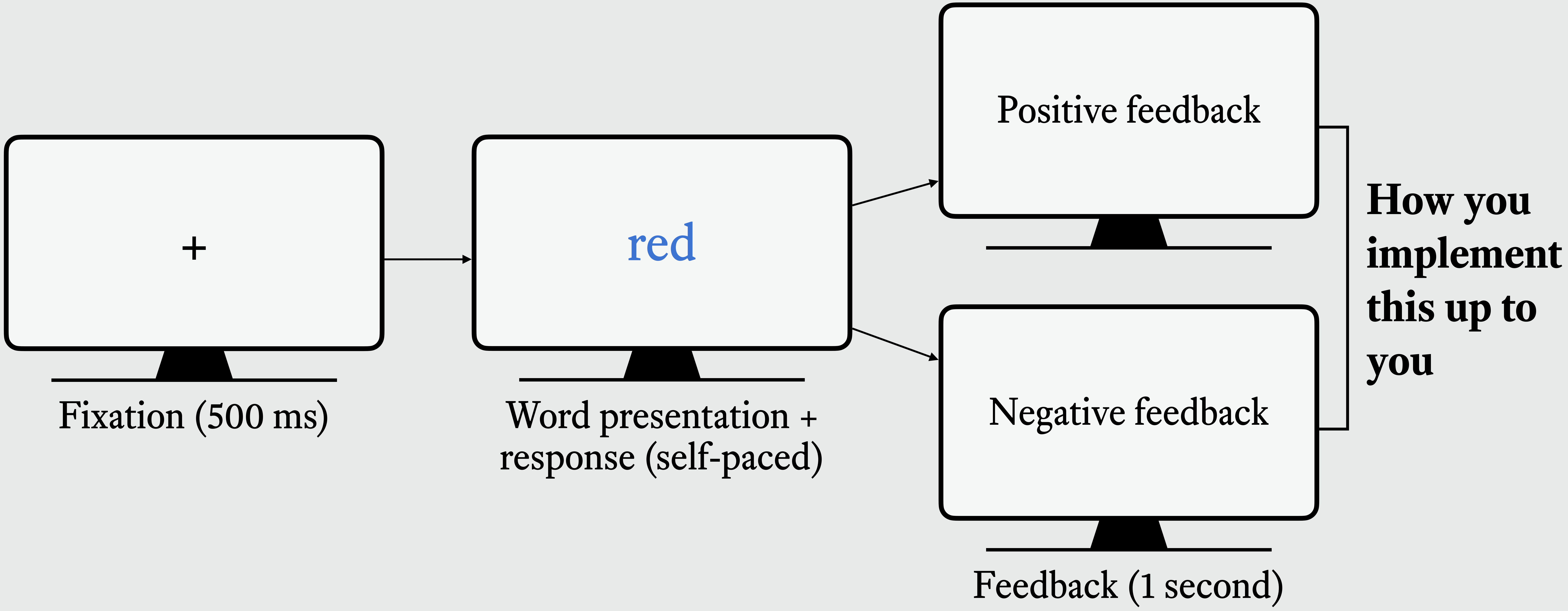
```python
from expyriment.misc.constants import K_SPACE, K_1, K_2, K_DOWN, K_UP, K_LEFT, K_RIGHT

ADJUST_SIZE = 5
STEP_MOVE = 5

KEYMAP = {
    K_1: ("1", "radius", -ADJUST_RADIUS),
    K_2: ("2", "radius", +ADJUST_RADIUS),
    K_DOWN: ("down", "move", (0, -STEP_MOVE)),
    K_UP: ("up", "move", (0, +STEP_MOVE)),
    K_LEFT: ("left", "move", (-STEP_MOVE, 0)),
    K_RIGHT: ("right", "move", (+STEP_MOVE, 0))
}
```

# blindspot.py

```
def run_trial(eye, radius=75):
   fixation = stimuli.FixCross(..., position=([300, 0] if eye == "left" else [-300, 0]))
   circle = make_circle(radius)

   make_instructions(eye).present(); exp.keyboard.wait()

   while True:
      draw([fixation, circle])

      key, _ = exp.keyboard.wait(KEYS)
      if key == K_SPACE: break
      keypressed, action, change = KEYMAP.get(key)

      if action == "move": circle.move(change)
      else:
         radius = max(1, radius + change)
         circle = make_circle(radius, circle.position)

      x, y = circle.position
      exp.data.add([eye, keypressed, radius, x, y])
```

## First experiment: Stroop effect

# Exercise 1: Stroop task



# Exercise 1: Stroop task

Open `stroop.py` and modify it such that:

- Participants decide via their keyboard whether word meaning and text color match (COLORS: red, blue, green, orange) **with feedback**
- For each trial in the `exp` loop, randomly choose a trial type (match/mismatch), one color word and one color for the text: The `random.choice()` function returns one element at random from a list
- There should be **32 trials** in total, equally divided into **2 blocks**

# stroop.py

```python
TRIAL_TYPES = ["match", "mismatch"]
COLORS = ["red", "blue", "green", "orange"]

stims = {w: {c: stimuli.TextLine(w, text_colour=c) for c in COLORS} for w in COLORS}
load([stims[w][c] for w in COLORS for c in COLORS])

for block in range(1, N_BLOCKS + 1):
    for trial in range(1, N_TRIALS_IN_BLOCK + 1):
        trial_type = random.choice(TRIAL_TYPES)
        word = random.choice(COLORS)
        color = word if trial_type == "match" else random.choice([c for c in COLORS if
                c != word])
        run_trial(block, trial, trial_type, word, color)
```

## A note on feedback

**Asymmetrical feedback** is fine: Show a message only when the participant makes a mistake (they can infer when they were correct)

Make sure that the feedback does not bias participants' next response: Showing a red screen is bad if the color on the next trial is red; displaying "**g**ood!" is bad if the word on the next trial is "**g**reen"

In the particular case of the Stroop task, **sounds**, **emojis**, or a **longer sentence** ("That was wrong!") would all be good options

# Is this a good experiment?

The ideal experiment **minimizes differences between conditions** to only those driven by the research question

Any **additional differences introduce confounds** that undermine the validity of the conclusions

How does this apply to our Stroop task?

## Problems in the current Stroop task

**Problem 1**: The number of words, colors, and word–color pairings is not matched, so what if the effect emerges only for a *subset* of words/colors/pairings that happen to be selected?

**Problem 2**: The number of trial types is not matched—what if participants answer faster on the *more frequent* trial type?

## Counterbalancing: Overview

# Full factorial design

To avoid **confounds**, experimental factors are **crossed**, so that each factor and each factor combination appears equally often: **balanced design**

**Binary factors** are very easy to work with ($2^n$ possibilities), but this idea generalizes to factors with any number of levels

$n = 3$

$$
\begin{array}{c} \\ ([1]^4 + [0]^4)^1 \end{array}
\begin{array}{ccc} \mathbf{A} & \mathbf{B} & \mathbf{C} \\ \hline \end{array}
$$

$$
([1]^4 + [0]^4)^1 \quad \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad ([1]^1 + [0]^1)^4
$$

**Column $i$: $([1]^{2^{n-i}} + [0]^{2^{n-i}})^{2^{i-1}}$**

# Full factorial design in Python

```
# Highly recommended reading: https://docs.python.org/3/library/itertools.html
import itertools

letters = ("A", "B", "C", "D")
sides = ("left", "center", "right")
sizes = (50, 100, 150, 200)

for comb in itertools.product(letters, sides, sizes): # Cartesian product of the 3 sets
    print(comb)

>>> ("A", 'left', 50)
>>> ("A", 'left', 100)
...
>>> ("D", 'right', 150)
>>> ("D", 'right', 200)
```

## Limitations of the full factorial design

For logical and practical purposes, it is **sometimes impossible to use a full factorial design**

**Logical**: If there are two conditions that must be administered sequentially, no subject can go through both orders A–B and B–A

**Practical**: If the number of trials required for full factorization is huge

# Latin squares

## Cycled

*each condition appears equally often in each*

| S1 | S2 | S3 | S4 |
|---|---|---|---|
| **A** | B | C | D |
| B | C | D | **A** |
| C | D | **A** | B |
| D | **A** | B | C |

## Balanced

| S1 | S2 | S3 | S4 |
|---|---|---|---|
| **A** | B | C | D |
| B | C | D | **A** |
| D | **A** | B | C |
| C | D | **A** | B |

*each condition also precedes and follows every other condition equally often*

```
design.permute.latin_square(["A", "B", "C", "D"], permutation_type='cycled')
[['A', 'B', 'C', 'D'], ['B', 'C', 'D', 'A'], ['C', 'D', 'A', 'B'], ['D', 'A', 'B', ‘C']]

design.permute.latin_square(["A", "B", "C", "D"], permutation_type='balanced')
[[‘A', 'B', 'D', 'C'], ['B', 'C', 'A', 'D'], ['C', 'D', 'B', 'A'], ['D', 'A', 'C', 'B']]
```

# Even distribution

| LETTER | SIDE | SIZE |
|---|---|---|
| A | left | 50 |
| A | left | 100 |
| A | left | 150 |
| A | left | 200 |
| | … | |
| D | left | 50 |
| D | left | 100 |
| D | left | 150 |
| D | left | 200 |
| | … | |

+ COLOR(r, g)

| LETTER | SIDE | SIZE | COLOR |
|---|---|---|---|
| A | left | 50 | r |
| A | left | 100 | g |
| A | left | 150 | g |
| A | left | 200 | r |
| | … | | |
| D | left | 50 | g |
| D | left | 100 | r |
| D | left | 150 | r |
| D | left | 200 | g |
| | … | | |

## Counterbalancing in the Stroop task

# Full factorial design in Stroop task

**Crossing** our experimental factors:

WORD (*red*, *blue*, *green*, *orange*) × COLOR (r, b, g, o) × TRIAL TYPE (match, mismatch) = 32 trials

**Problem**: Nonsensical trials (e.g., *red*–r–mismatch)

The 3 factors are **not** independent: TRIAL TYPE can be determined from WORD and COLOR

| WORD | COLOR | TRIAL TYPE |
|---|---|---|
| *red* | r | match |
| *red* | r | mismatch |
| *red* | b | match |
| *red* | b | mismatch |
| *red* | g | match |
| *red* | g | mismatch |
| *red* | o | match |
| *red* | o | mismatch |
|  | … |  |

# Dropping out trial type

**Possible solution 1**: Cross the word and color factors only and determine trial type afterward

WORD (*red*, *blue*, *green*, *orange*) × COLOR (r, b, g, o) = 16 trials

**Problem**: Imbalance in trial types (3 times as many mismatch trials)

| WORD | COLOR | TRIAL TYPE |
|---|---|---|
| *red* | r | match |
| *red* | b | mismatch |
| *red* | g | mismatch |
| *red* | o | mismatch |
| | … | |

# Forcing TRIAL TYPE equality

**Possible solution 2**: For each word, multiply the corresponding congruent trials by 3 to get 3 congruent and 3 incongruent trials for each word

**Problem**: One condition has higher variability (MATCH: only *red*–red trials, MISMATCH: 3 colored *red* strings)

| WORD | COLOR | TRIAL TYPE |
|---|---|---|
| *red* | r | match |
| *red* | r | match |
| *red* | r | match |
| *red* | b | mismatch |
| *red* | g | mismatch |
| *red* | o | mismatch |
| | … | |
| *blue* | r | mismatch |
| | … | |
| *green* | r | mismatch |
| | … | |
| *orange* | r | mismatch |

# Fix mismatch assignment

**Possible solution 3**: For each word, choose a unique mismatch color and fix that throughout the experiment

**Problem**: Not all factors are tested, which may limit the generalizability of the conclusion based on the effect

| WORD | COLOR | TRIAL TYPE |
|---|---|---|
| *red* | r | match |
| *red* | b | mismatch |
| *blue* | b | match |
| *blue* | r | mismatch |
| *green* | g | match |
| *green* | o | mismatch |
| *orange* | o | match |
| *orange* | g | mismatch |
|  | … |  |

# Counterbalance *across* subjects

**Possible solution 4**: Across subjects, iterate through all possible word–color mismatches

For this, we need to use a subset of the permutations of (*red*, *blue*, *green*, *orange*): those in which **no element appears in the original position**: (*blue*, *red*, *orange*, *green*) works; (*blue*, *green*, *red*, *orange*) doesn't

| WORD | COLOR | TRIAL TYPE |
|---|---|---|
| *red* | r | match |
| *red* | o | mismatch |
| *blue* | b | match |
| *blue* | g | mismatch |
| *green* | g | match |
| *green* | b | mismatch |
| *orange* | o | match |
| *orange* | r | mismatch |
|  | … |  |

# Derangements

**$n = 1$; $!n = 0$**

| 1 |
|---|
| — |

**$n = 2$; $!n = 1$**

| 1 | 2 |
|---|---|
| 2 | 1 |

**$n = 3$; $!n = 2$**

| 1 | 2 | 3 |
|---|---|---|
| 2 | 3 | 1 |
| 3 | 1 | 2 |

**$n = 4$; $!n = 9$**

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 2 | **1** | 4 | 3 |
| 2 | 4 | **1** | 3 |
| 2 | 3 | 4 | **1** |
| 3 | 4 | **1** | 2 |
| 3 | **1** | 4 | 2 |
| 3 | 4 | 2 | **1** |
| 4 | 3 | 2 | **1** |
| 4 | 3 | **1** | 2 |
| 4 | **1** | 2 | 3 |

**Number of permutations** $n!$

**Number of derangements**

$$!n = (n - 1) \times (!(\mathrm{n} - 2) + !(n - 1))$$

## Counterbalancing in the `expyriment` script

# Counterbalancing for one subject

```
# Helper for obtaining derangements in python
def derangements(lst):
    ders = []
    for perm in itertools.permutations(lst):
        if all(original != perm[idx] for idx, original in enumerate(lst)):
            ders.append(lst)

COLORS = ["red", "blue", "green", “orange"]
mappings = derangements(COLORS) # The 9 derangements

# A list of dictionaries for the trials
trials = (
  [{"trial_type": "match", "word": c, "color": c} for c in COLORS] +
  [{"trial_type": "mismatch", "word": w, "color": c} for w,c in zip(COLORS, mappings[0])]
)
```

# Iterating over derangements across subjects

```
COLORS = ["red", "blue", "green", “orange"]
mappings = derangements(COLORS) # The 9 derangements

subject_id = 1, 2, ..., n
order = (subject_id – 1) % len(mappings) # modulo arithmetic
mapping = mappings[order] # Choose based on subject ID

# A list of dictionaries for the trials
trials = (
  [{"trial_type": "match", "word": c, "color": c} for c in COLORS] +
  [{"trial_type": "mismatch", "word": w, "color": c} for w, c in zip(COLORS, mapping)]
)
```

# At this point

## Subject 1

| WORD | COLOR | TRIAL TYPE |
|---:|---:|---:|
| *red* | r | match |
| *blue* | b | match |
| *green* | g | match |
| *orange* | o | match |
| *red* | b | mismatch |
| *blue* | r | mismatch |
| *green* | o | mismatch |
| *orange* | g | mismatch |

## Subject 2

| WORD | COLOR | TRIAL TYPE |
|---:|---:|---:|
| *red* | r | match |
| *blue* | b | match |
| *green* | g | match |
| *orange* | o | match |
| *red* | b | mismatch |
| *blue* | g | mismatch |
| *green* | o | mismatch |
| *orange* | r | mismatch |

…

# Extending the number of trials

```python
N_BLOCKS = 2
N_TRIALS_IN_BLOCK = 16

# A list of dictionaries for the base trials
base = (
  [{"trial_type": "match", "word": c, "color": c} for c in COLORS] +
  [{"trial_type": "mismatch", "word": w, "color": c} for w, c in zip(COLORS, mapping)]
)

# Create 32 trials, divided in 2 blocks, in a nested list
block_repetitions = N_TRIALS_IN_BLOCK // len(base)
blocks = []

for b in range(1, N_BLOCKS + 1):
   b_trials = base_mappings * block_repetitions # 16 trials
   trials = [{"block_id": b, "trial_id": i, **t} for i, t in enumerate(b_trials, 1)]
   blocks.append(trials) # len(blocks) = 2; len(trials) = 16
```

# Problem: `trials` is too orderly

```
for b in range(1, N_BLOCKS + 1):
   b_trials = base_mappings * block_repetitions
   random.shuffle(b_trials)
   trials = [{"block_id": b, "trial_id": i, **t} for i,
t in enumerate(b_trials, 1)]
   blocks.append(trials)
```

## Subject 1: Randomized

| WORD | COLOR | TRIAL TYPE |
|---|---|---|
| *blue* | r | mismatch |
| *blue* | b | match |
| *orange* | o | match |
| *red* | r | match |
| *orange* | g | mismatch |
| *green* | o | mismatch |
| *green* | g | match |
| *red* | r | match |
| *red* | b | mismatch |
| | … | |

# The `expyriment` loop

```python
""" Experiment """
control.start(subject_id=subject_id)

present_instructions(INSTR_START)
for block_id, block in enumerate(blocks, 1):
    for trial in block:
        run_trial(**trial)
    if block_id != N_BLOCKS:
        present_instructions(INSTR_MID)
present_instructions(INSTR_END)

control.end()
```

## Is this a good experiment?

**Problem 1**: The number of words, colors, and word–color pairings is not matched, so what if the effect emerges only for a *subset* of words/colors/pairings that happen to be selected?

## Solved by counterbalancing

**Problem 2**: The number of trial types is not matched—what if participants answer faster on the *more frequent* trial type?

**Problem 3**: If mismatch trials are harder than match trials, what can we conclude?

## Is this a good experiment?

**Problem 1**: The number of words, colors, and word-color pairings is not matched, so what if the effect emerges only for a *subset* of words/colors/pairings that happen to be selected?

**Solved by counterbalancing**

**Problem 2**: The number of trial types is not matched—what if participants answer faster on the *more frequent* trial type?

**Problem 3**: If mismatch trials are harder than match trials, what can we conclude?

**Solve by changing the responses**

## Counterbalancing outside the `expyriment` script

## Another way of handling the counterbalancing

**So far**: We did the counterbalancing inside the `expyriment` script

**Also possible** (and sometimes better): Prepare a full counterbalancing sheet outside `expyriment` and have it read in `expyriment` at the beginning of the experiment

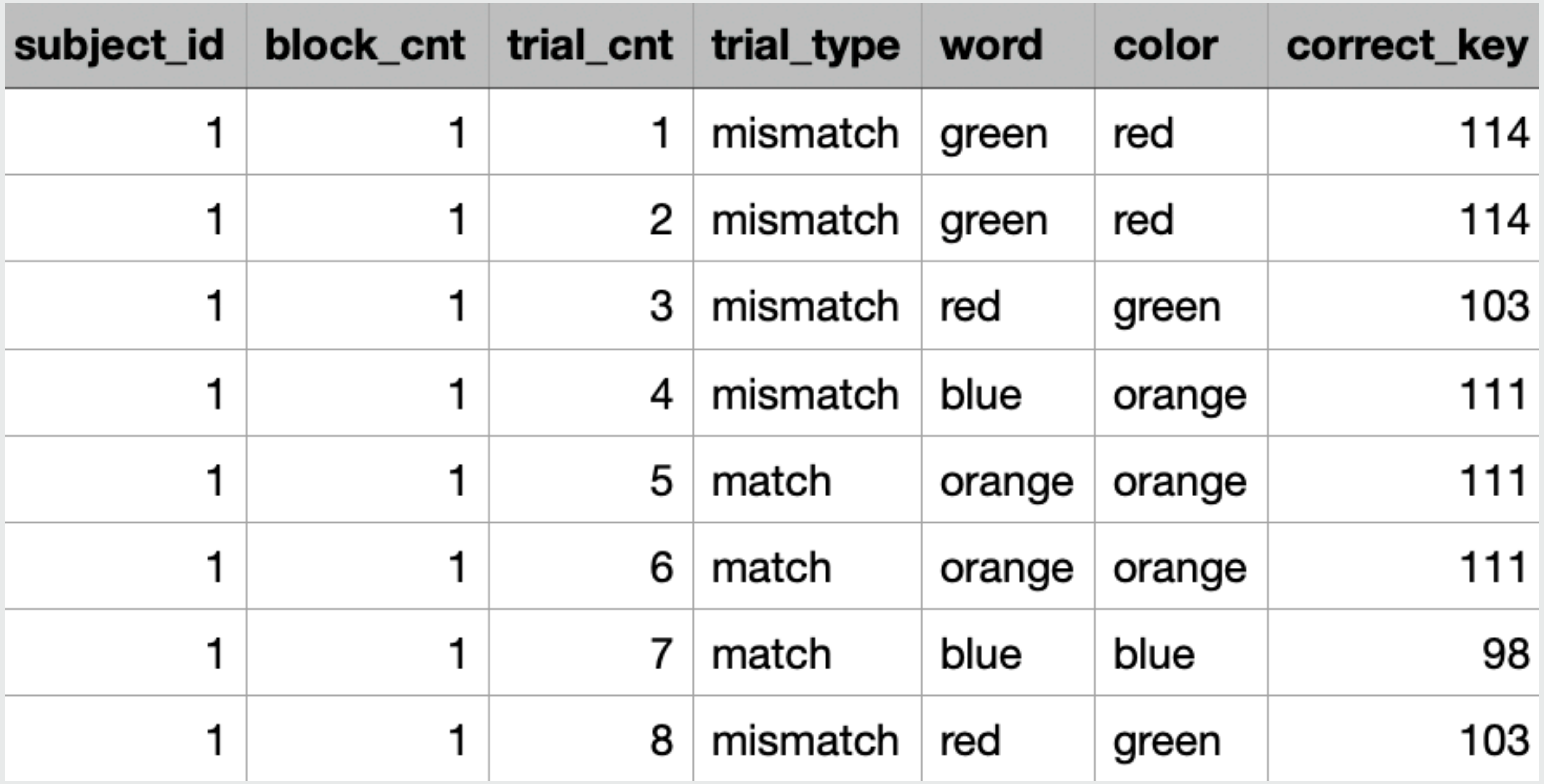
# Implementation

```python
def subject_trials(subject_id):
    mismatch = MISMATCHES[(subject_id - 1) % len(MISMATCHES)]
    base = [{"word": w, "color": w} for w in COLORS] +
           [{"word": w, "color": c} for w, c in zip(COLORS, mismatch)]

    block_reps = N_TRIALS_IN_BLOCK // len(base)
    trials = []

    for b_index in range(1, N_BLOCKS + 1):
        block = base * block_reps
        random.shuffle(block)
        for t_index, trial in enumerate(block, 1):
            trials.append({
                "subject_id": subject_id, "block_id": b_index, "trial_id": t_index,
                "trial_type": "match" if trial["word"] == trial["color"] else "mismatch",
                "word": trial[“word"], "color": trial["color"],
                "correct_key": ord(trial[“color"][0])})

    return trials
```

## Implementation

```
import csv

all_trials = [trial for id in range(1, N_SUBJECTS + 1) for trial in subject_trials(id)]

csv_cols = ["subject_id", "block_id", "trial_id", "trial_type", "word", "color",
"correct_key"]

with open("cb.csv", "w", newline="") as f: # Opens a file "cb.csv" in write mode
    w = csv.DictWriter(f, fieldnames=csv_cols) # Creates a csv writer for writing dicts
    w.writeheader() # Writes the first row (column names)
    w.writerows(all_trials) # Writes everything else
```

# Output



| subject_id | block_cnt | trial_cnt | trial_type | word | color | correct_key |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | mismatch | green | red | 114 |
| 1 | 1 | 2 | mismatch | green | red | 114 |
| 1 | 1 | 3 | mismatch | red | green | 103 |
| 1 | 1 | 4 | mismatch | blue | orange | 111 |
| 1 | 1 | 5 | match | orange | orange | 111 |
| 1 | 1 | 6 | match | orange | orange | 111 |
| 1 | 1 | 7 | match | blue | blue | 98 |
| 1 | 1 | 8 | mismatch | red | green | 103 |

## Reading the csv in `expyriment`

```python
with open("cb.csv", "r") as f: # Opens the file "cb.csv" in read mode
    reader = csv.DictReader(f) # Creates a csv reader for reading to dictionaries
    trials = [row for row in reader if row["subject_id"] == str(subject_id)]
    # Type conversion necessary since the csv entries are read as strings

for trial in trials:
    if trial["trial_id"] == 1 and trial["block_id"] != 1:
        present_instructions(INSTR_MID)
    run_trial(**trial)
```

## Counterbalancing with `expyriment` functions

# Between-subject factors

`expyriment` offers several tools for doing it all natively

For instance, `expyriment` can implement between-subject factors

While color assignment is technically not a factor of interest, this can still be used to counterbalance variable assignments across subjects

```
exp = design.Experiment(name="Stroop")
exp.add_bws_factor("assignment", derangements(COLORS)) # There are 9 conditions now

subject_id = 1 # Expyriment administers the conditions automatically based on subject_id
COLOR_CROSSINGS = exp.get_permuted_bws_factor_condition("assignment", subject_id)
dict_colors = dict(zip(COLORS, COLOR_CROSSINGS))
```

# The Block class

Then, `expyriment.design` has two classes, **`Block`** and **`Trial`**, that can help with further counterbalancing and organizing as well

```python
TRIAL_TYPES = ["match", "mismatch"]
COLORS = ["red", "green", "blue", "orange"]
FACTORS = {"trial_type": TRIAL_TYPES, "word": COLORS}

block = design.Block("Block 1")
block.add_trials_full_factorial(FACTORS, copies=1)
block.shuffle_trials(method=0, max_repetitions=None, n_segments=1)
```

# The Trial class

Once the full-factorial trials have been added to Block 1, they can be further modified (e.g., adding stimuli, defining other parameters)

```python
for trial in block.trials:
    trial_type = trial.get_factor("trial_type")
    word = trial.get_factor("word")

    color = word if trial_type == "match" else dict_colors[word]
    trial.set_factor("color", color)
    trial.set_factor("correct_key", ord(color[0]))

    trial.add_stimulus(stimuli.TextLine(word, text_colour=color))
    trial.preload_stimuli()

exp.add_block(block)
```

# Accessing trial properties in `Trial`

```python
def run_trial(block_id, trial_id, trial):
    trial_type, word, color, correct_key = (
        trial.factor_dict[k] for k in ("trial_type", "word", "color", "correct_key")
    )

    present_for(fixation, t=500)
    trial.stimuli[0].present()
    key, rt = exp.keyboard.wait(KEYS)
    correct = key == correct_key
    present_for(feedback_correct if correct else feedback_incorrect, t=1000)

    exp.data.add([block_id, trial_id, trial_type, word, color, key, rt, correct])
```

# Running the loop

```python
present_instructions(INSTR_START)

for block_id, block in enumerate(exp.blocks, 1): # exp has blocks
    for trial_id, trial in enumerate(block.trials, 1): # block has trials
        run_trial(block_id, trial_id, trial)

    if block_id != N_BLOCKS:
        present_instructions(INSTR_MID)

present_instructions(INSTR_END)
```

## Summary

Even the seemingly trivial Stroop task poses nontrivial design problems

When a full factorial is not possible, researchers need to use their best judgment to decide how to balance the design: **randomization**, **latin squares**, balancing **across participants**

These decisions often need to be made **on a case-by-case basis**, depending on the research question under investigation

# One possible algorithm

**Step 1**: Determine the fully crossed factors and generate the corresponding dataframe

**Step 2**: Determine how you will distribute the remaining factors over the rows and counterbalance across subjects

**Step 3**: Shuffle the trials (±constraints: e.g., maximum number of repetitions)

| | | Fully crossed | | | Counterb. | |
|---|---|---|---|---|---|---|
| ID | #T | X | Y | Z | U | V |
| 2 | 1 | 0 | 1 | 0 | 1 | 1 |
| 2 | 2 | 1 | 1 | 0 | 0 | 0 |
| 2 | 3 | 1 | 1 | 1 | 1 | 0 |
| 2 | 4 | 0 | 1 | 1 | 0 | 1 |
| 2 | 5 | 0 | 0 | 0 | 0 | 0 |
| 2 | 6 | 1 | 0 | 1 | 0 | 1 |
| 2 | 7 | 1 | 0 | 0 | 1 | 1 |
| 2 | 8 | 0 | 0 | 1 | 1 | 0 |
| | | | … | | | |

## Exercise 2: Stroop task 2.0

Create a new script, `stroop_balanced.py`, that modifies `stroop.py` as follows:

- Have participants decide **the color the onscreen word is written in**
- Choosing the method you think is best, **balance the design**
- The Stroop experiment should have 128 trials, divided in 8 blocks

## Taking stock

## You already know a lot

How to create and customize stimuli

How to draw them on-screen and present them for a specific duration

How to record key presses and reaction times

How to store this data for offline analysis

How to prepare the counterbalancing sheet

# You already know a lot

**Typical structure of an expyriment script**

1. Import functions/constants from `expyriment` and other modules (`math`, `random`)
2. Define useful constants and helper functions
3. Define global settings of the experiment
4. Initialize and preload stimuli
5. Define trial structure (constants vs variables) and data collection
6. Run experiment by looping over the counterbalanced trials and blocks

## You already know a lot

**The good news**: This is a huge step toward being able to autonomously code a full experiment

**The bad news**: The best way to make good progress is to get your hands dirty by programming experiments from scratch—you will encounter all sorts of real, idiosyncratic problems and you will have to figure them out on your own

**So now it's your turn to put this knowledge to work**

# Coding projects

**Implement a published experiment** in teams of 4 members (9–10 projects)

During the midterm break, **form the teams and choose a paper**

We prepared a selection that you can access in this spreadsheet (PDFs in this folder), but you can also opt for another paper if you want (just discuss this with us on Discord)

Once you've formed the teams and chosen a paper, **enter your choice** and team members in the spreadsheet

# Coding projects

**WEEK 7**: Short presentation (7 minutes) in which each team presents **the research question** that the experiment aims to answer, **why** they find it interesting (or not), the **methods**, a **short plan** (or pseudocode) for the implementation, and the **problems** they expect to encounter

**WEEK 8**: Short presentation (7 minutes) in which each team **explains their code** and shows **a short demo** of the experiment

*Note.* Each team should choose **different presenters** for the two weeks

## Implementation

Each project should have **a public GitHub repository** created by one of the team members, to which the others should be invited to contribute

**Divide labor**: Global settings of the experiment | Trial structure | Stimuli | Counterbalancing | Data organization | Presentation

*Note*. Some experiments involve techniques (e.g., mouse clicks, audio sounds) that we have not covered directly: Use the `expyriment` documentation and don't hesitate to ask questions on Discord

## Push your work to GitHub