

Matching and Treatments in PGG

Treatments at the session-level

There are two main ways to implement treatments. You can administer treatments on the session-level so every subject in a session is in the same treatment. This is very easy to set up in oTree so we'll start with this. Later, we'll look at how to assign treatments within treatments, so subjects play different treatments in parallel.

Random Matching

By default, oTree fills groups by arrival time in each app and keeps the groups and player IDs within the groups constant across rounds of the same app. This is fine if the game is not repeated and arrival time is random (this works in the lab with subjects taking place on random machines). If the game is repeated sometimes repeated-game effects due to being able to identify one's partner(s) are not a problem or part of the design.

If we want to repeat the same game but with different partners, we have to set up a rematching procedure between rounds. Let's start with a simple matching procedure that shuffles all players and groups randomly so subjects cannot identify their partners between rounds.

We will write a function that sets up the matching for our app. Go to our pgg app's `__init__.py`. We already have blocks for `# MODELS` and `# PAGES`, let's add a new block `# FUNCTIONS` at the end. This is strictly speaking not necessary, but I prefer to have an orderly setup.

Set up a new function called `creating_session()`. This is a pre-defined oTree function that gets automatically called when creating a session. Whenever we create a new session, oTree creates all subsessions (i.e. rounds of an app) that the experiment will go through. Pass the `creating_session` the element of the subsession that is currently being created.

```
def creating_session(subsession: Subsession)
    pass
```

Now if we want to shuffle all groups and player randomly every round, we can use one of the pre-defined oTree functions: `group_randomly()`

```
def creating_session(subsession: Subsession):
    subsession.group_randomly()
```

Start the otree devserver and test if player IDs and groups are reshuffled between rounds.

Matching Procedures as Treatments

Let's use some more advanced matching procedures as treatments. We'll do shuffling positions within groups and rematching within matching silos. The idea is as follows: Random matching is the Baseline (default treatment if none other is enabled), and Group Shuffling and Matching Silos are treatments that can be enabled. Go back to SESSION_CONFIGS and add two new elements to the config: `shuffle_within_groups` and `matching_silos`. Also, rename the display name and increase the number of demo participants to 12 (so the matching actually makes sense).

```
SESSION_CONFIGS = [
    dict(
        name='course',
        display_name='WZB oTree course Baseline',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=False,
        matching_silos=False,
    ),
]
```

Save it and run otree devserver to check if the config works. Next, we create two more session configs that enable one treatment each.

```
SESSION_CONFIGS = [
    dict(
        name='course',
        display_name='WZB oTree course Baseline',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=False,
        matching_silos=False,
    ),
    dict(
        name='course',
        display_name='WZB oTree course Group Shuffling',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=True,
        matching_silos=False,
    ),
    dict(
        name='course',
        display_name='WZB oTree course Matching Silos',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=False,
        matching_silos=True,
    ),
]
```

Session configs are not saved to the database. In our output data file we should include the treatment that the subjects were in. So we go to our trusty `__init.py__` and we include a new player variable called `treatment`. I set it up as a string field because I want to store the name of the treatment as a text string.

```
class Player(BasePlayer):
    contribution = models.CurrencyField(
        label='How much do you want to put into the group account?',
        min=0,
        max=C.BUDGET
    )
    treatment = models.StringField()
```

Save it and run otree devserver to check if the config works.

Group Shuffling

In some applications where subjects' roles are dependent on the position/ID in their group (think, for example a sender-receiver game) it makes sense to reshuffle only the players within each group but keep the group itself intact. To achieve this, first import the standard Python random library at the top of `__init__.py`.

```
import random
```

Next, return to the `creating_session` function. First, we check if we are in the Group Shuffling treatment.

```
def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups']:
```

If so, we get the subsession's default group matrix and save it in the local variable `matching`.

```
def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups']:
        matching = subsession.get_group_matrix()
```

`get_group_matrix` is a function that returns the current matching of a subsession as a list of lists. On the higher level the list is a list of all the groups in the subsession. Each group is its own list of all the players in the group. For example, the `matching` variable could look as follows.

```
[
    [Player 1, Player 2, Player 3],
    [Player 4, Player 5, Player 6],
    [Player 7, Player 8, Player 9],
```

```
[Player 10, Player 11, Player 12],
]
```

Next, we use the random library's shuffle function. Shuffle takes all elements in a list and randomly rearranges them. Shuffling the matching variable itself is not enough -- this would simply rearrange the groups but keep the positions within each group the same. Instead, we have to loop through all groups and shuffle them. The shuffle function takes a list as input and automatically saves the shuffled list under the same name again.

```
for group in matching:
    random.shuffle(group)
```

Then, we could end up with a new matching variable like this.

```
[
    [Player 2, Player 1, Player 3],
    [Player 6, Player 4, Player 5],
    [Player 7, Player 8, Player 9],
    [Player 10, Player 12, Player 11],
]
```

Then, we tell oTree to use the reshuffled matching list as the new matching matrix.

```
subsession.set_group_matrix(matching)
```

Finally, we tell oTree to save the current treatment for every player in the group by writing to the treatment variable that we set up before.

```
players = subsession.get_players()
for player in players:
    player.treatment = 'shuffle within groups'
```

In the end, the top of our creating_session function should look as follows:

```
def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups']:
        matching = subsession.get_group_matrix()
        for group in matching:
            random.shuffle(group)
        subsession.set_group_matrix(matching)
        players = subsession.get_players()
```

```
for player in players:
    player.treatment = 'shuffle within groups'
```

Finally, we need to tell Python that if we are not in a treatment, we want to use random matching. We do that by including an else statement and moving the `subsession.group_randomly()` command one indent to the right. Also, we tell oTree to save this treatment as the baseline.

```
def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups']:
        matching = subsession.get_group_matrix()
        for group in matching:
            random.shuffle(group)
        subsession.set_group_matrix(matching)
        players = subsession.get_players()
        for player in players:
            player.treatment = 'shuffle within groups'
    else:
        subsession.group_randomly()
        players = subsession.get_players()
        for player in players:
            player.treatment = 'baseline'
```

Matching Silos & Working with LLMs

Suppose that we are working with 12 players in four groups of three. Between each round, we want to rematch players in the first two and the last two groups, so the matching silos' observations are independent. This can be quite cumbersome. Remember, we have a grouping structure like this:

```
[
    [Player 1, Player 2, Player 3],
    [Player 4, Player 5, Player 6],
    [Player 7, Player 8, Player 9],
    [Player 10, Player 11, Player 12],
]
```

Instead of figuring out manually what to do, we can use large language models to help us. The experimental logic is programmed in Python, and modern LLMs have been trained on a large set of Python problems found on the internet. So I went to Microsoft Copilot and described my problem.

"I am programming in Python. I have 12 players in four groups of three. They are represented by arrays in arrays like so:

```
matching = [ [Player 1, Player 2, Player 3], [Player 4, Player 5, Player 6], [Player 7, Player 8, Player 9], [Player 10, Player 11, Player 12], ]
```

I want to reshuffle the players, but I want to reshuffle them between the first two and the last two groups only. How can I do that?"

Copilot responded:

```
import random

# Your original list of players
matching = [
    ["Player 1", "Player 2", "Player 3"],
    ["Player 4", "Player 5", "Player 6"],
    ["Player 7", "Player 8", "Player 9"],
    ["Player 10", "Player 11", "Player 12"]
]

# Flatten the nested list
all_players = [player for group in matching for player in group]

# Split into two parts: first two groups and last two groups
first_two_groups = all_players[:6]
last_two_groups = all_players[6:]

# Shuffle each part independently
random.shuffle(first_two_groups)
random.shuffle(last_two_groups)

# Combine shuffled parts back into the original structure
shuffled_matching = [first_two_groups[:3], last_two_groups[:3]]

print(shuffled_matching) # This will give you the reshuffled players
```

Looks great, right? But if you go through the entire code, you'll hopefully find an error. The shuffled players variable is constructed in a wrong way and drops half the players! It only includes the first three players of first_two_groups but not the last three players. Also, it omits the first three players from last_two_groups. You can test the code by opening the terminal, typing python and entering the code line by line. If you want to exit the python prompt in the terminal, type quit().

So let's fix the shuffled_players line.

```
shuffled_players = [first_two_groups[:3], first_two_groups[3:],
                    last_two_groups[:3], last_two_groups[3:]]
```

Let's add the code to our matching procedure. We can create a new else if clause (elif) and add it after the first if and before the else.

```
def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups']:
        matching = subsession.get_group_matrix()
        for group in matching:
            random.shuffle(group)
        subsession.set_group_matrix(matching)
```

```

        players = subsession.get_players()
        for player in players:
            player.treatment = 'shuffle within groups'
    elif subsession.session.config['matching_silos']:

    else:
        subsession.group_randomly()
        players = subsession.get_players()
        for player in players:
            player.treatment = 'baseline'

```

Now we add the fixed code in the elif statement. We already imported random above, so we don't have to import it again. Mind the correct indentation! Also, don't forget to save the treatment to the treatment variable.

```

def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups']:
        matching = subsession.get_group_matrix()
        for group in matching:
            random.shuffle(group)
        subsession.set_group_matrix(matching)
        players = subsession.get_players()
        for player in players:
            player.treatment = 'shuffle within groups'
    elif subsession.session.config['matching_silos']:
        matching = subsession.get_group_matrix()
        # Flatten the nested list
        all_players = [player for group in matching for player in group]

        # Split into two parts: first two groups and last two groups
        first_two_groups = all_players[:6]
        last_two_groups = all_players[6:]

        # Shuffle each part independently
        random.shuffle(first_two_groups)
        random.shuffle(last_two_groups)

        # Combine shuffled parts back into the original structure
        shuffled_matching = [first_two_groups[:3], first_two_groups[3:],
last_two_groups[:3], last_two_groups[3:]]
        subsession.set_group_matrix(shuffled_matching)
        players = subsession.get_players()
        for player in players:
            player.treatment = 'matching silos'
    else:
        subsession.group_randomly()
        players = subsession.get_players()
        for player in players:
            player.treatment = 'baseline'

```

Adding another treatment dimension

For the public goods game we'll add another simple treatment variation. A standard treatment variation is changing the multiplier.

To start, go to settings.py and add a new entry called multiplier to our session dictionaries.

```
SESSION_CONFIGS = [  
    dict(  
        name='course',  
        display_name='WZB oTree course Baseline',  
        app_sequence=['consent', 'pgg'],  
        num_demo_participants=12,  
        shuffle_within_groups=False,  
        matching_silos=False,  
        multiplier=0.4,  
    ),  
    dict(  
        name='course',  
        display_name='WZB oTree course Group Shuffling',  
        app_sequence=['consent', 'pgg'],  
        num_demo_participants=12,  
        shuffle_within_groups=True,  
        matching_silos=False,  
        multiplier=0.4,  
    ),  
    dict(  
        name='course',  
        display_name='WZB oTree course Matching Silos',  
        app_sequence=['consent', 'pgg'],  
        num_demo_participants=12,  
        shuffle_within_groups=False,  
        matching_silos=True,  
        multiplier=0.4,  
    ),  
]
```

Now that we have defined the multiplier in the session config we have to change our code to switch from the multiplier that we initially defined in the constants to our new config multiplier. We can access the multiplier by asking the session element which value the multiplier entry has.

First, we ask the session for the config dictionary.

```
session.config
```

Then, we ask the config dictionary for the multiplier entry.

```
session.config['multiplier']
```


We should save the current multiplier in the database since it will vary between treatments, and we must be able to identify which condition subjects were in when later looking at the data table. The session config is not automatically saved -- we have to add this variable manually. Scroll up to the Group class and add a new FloatField model.

```
class Group(BaseGroup):
    total = models.CurrencyField()
    multiplier = models.FloatField()
```

We now have to store the current value of our value in the database. There are multiple ways to do this. We could just store the value when we calculate the payoffs. But personally, I like to write this kind of data to the database as soon as I start the session. So go to the creating_session function that we created before. We ask the subsession for a list of all the groups in the subsession, and for each group in the subsession we set the multiplier to the one from the config. Note that in the for-loop here we ask every group g what the current session is to access the config. Only write to the group multiplier variable after you do the matching procedure! If you do it before the matching procedure, you will create new groups, so you destroy the data that you wrote to the variable.

```
def creating_session(subsession: Subsession):

    ### ALL THE MATCHING PROCEDURES COME BEFORE ###

    groups = subsession.get_groups()
    for g in groups:
        g.multiplier = g.session.config['multiplier']
```

In the end, your code should look as follows:

```
def creating_session(subsession: Subsession):
    if subsession.session.config['shuffle_within_groups'] == True:
        matching = subsession.get_group_matrix()
        for group in matching:
            random.shuffle(group)
        subsession.set_group_matrix(matching)
        players = subsession.get_players()
        for player in players:
            player.treatment = 'shuffle within groups'
    elif subsession.session.config['matching_silos'] == True:
        matching = subsession.get_group_matrix()
        all_players = [player for group in matching for player in group]
        first_two_groups = all_players[:6]
        last_two_groups = all_players[6:]
        random.shuffle(first_two_groups)
        random.shuffle(last_two_groups)
        shuffled_matching = [first_two_groups[:3], first_two_groups[3:],
```

```

last_two_groups[:3], last_two_groups[3:]]
    subsession.set_group_matrix(shuffled_matching)
    players = subsession.get_players()
    for player in players:
        player.treatment = 'matching silos'
else:
    subsession.group_randomly()
    players = subsession.get_players()
    for player in players:
        player.treatment = 'baseline'
groups = subsession.get_groups()
for g in groups:
    g.multiplier = g.session.config['multiplier']

```

Next, we have to replace the multiplier in our payoff code. Go to ChoiceWaitPage and replace C.MULTIPLIER with group.multiplier. All together, our new function should look as follows:

```

class ChoiceWaitPage(WaitPage):
    @staticmethod
    def after_all_players_arrive(group: Group):
        players = group.get_players()
        contributions = [player.contribution for player in players]
        group.total = sum(contributions)
        payout = group.total * group.multiplier
        for p in players:
            p.payoff = C.BUDGET - p.contribution + payout

```

There is only one thing left -- deleting the MULTIPLIER constant since it is now superfluous. Run otree devserver and test the code. Check out that you can now manipulate the multiplier when choosing Sessions -> Create New Session -> Configure Session. We could use this to use set up treatments but it can be dangerous -- you might forget changing the treatment variable or there could be a typo. Instead, it is advisable to set up session configs for multiple treatments. Go back to settings.py and go to SESSIONS_CONFIGS. Next, add a new dictionary with a different multiplier and treatment in the name and display name. You should end up with something like this:

```

SESSION_CONFIGS = [
    dict(
        name='course',
        display_name='WZB oTree course Baseline',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=False,
        matching_silos=False,
        multiplier=0.4,
    ),
    dict(
        name='course',
        display_name='WZB oTree course Baseline High Multiplier',

```

```
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=False,
        matching_silos=False,
        multiplier=0.5,
    ),
    dict(
        name='course',
        display_name='WZB oTree course Group Shuffling',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=True,
        matching_silos=False,
        multiplier=0.4,
    ),
    dict(
        name='course',
        display_name='WZB oTree course Matching Silos',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=12,
        shuffle_within_groups=False,
        matching_silos=True,
        multiplier=0.4,
    ),
]
```

Now you can run demo sessions with the correct values for the treatment variables, and you can double-check without having to manipulate them when setting up a session in the Sessions tab.