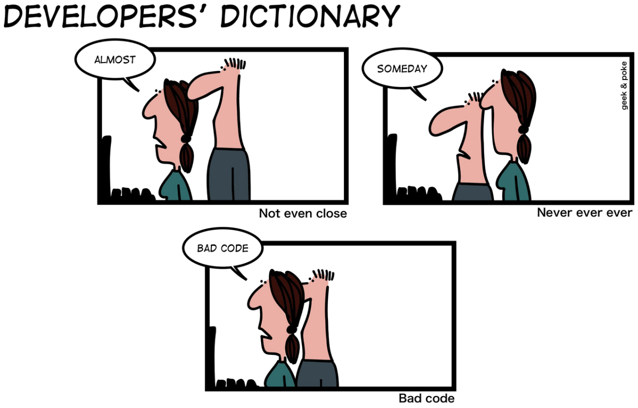
oTree Concepts #2 - Tutorial #2 - Bots.

Juan Cabral - jbc.develop@gmail.com

Jan, 2018

Tutorial #2: Trust Game.



- Open your console (Powershell, terminal, or any flaored pyton console)
- Open an editor (PyCharm, SublimeText, Kate, Atom. . .)
- Follow Me!

Tutorial #2: Trust Game.

- Now let's create a 2-player Trust game, and learn some more features of oTree.
  - To start, Player 1 receives 10 points;
  - Player 2 receives nothing.
  - Player 1 can send some or all of his points to Player 2.
  - Before P2 receives these points they will be tripled.
  - Once P2 receives the tripled points he can decide to send some or all of his points to P1.

Tutorial #2: Trust Game.

Define models.py

- First we define our app's constants. The endowment is 10 points and the donation gets tripled.
- There are 2 critical data points to record: the "sent" amount from P1, and the "sent back" amount from P2.
- Also, let's define the payoff function in the Group class.

Tutorial #2: Trust Game.

Define the templates and views

We need 3 pages:

1. P1's “Send” page
2. P2's “Send back” page
3. “Results” page that both users see.
4. It would also be good if game instructions appeared on each page so that players are clear how the game works.
5. This game has 2 wait pages:
   - 5.1 P2 needs to wait while P1 decides how much to send
   - 5.2 P1 needs to wait while P2 decides how much to send back
   - 5.3 After the second wait page, we should calculate the payoffs. So, we use `after_all_players_arrive`.
6. Then we define the page sequence.

Tutorial #2: Trust Game.

Settings and run

- Add an entry to `SESSION_CONFIGS` in `settings.py`
- Reset the database and run.

oTree Concepts #2.

Groups

- oTree's group system lets you divide players into groups and have players interact with others in the same group. This is often used in multiplayer games.
- To set the group size, go to your app's **`models.py`** and set **`Constants.players_per_group`**.

```
class Constants(BaseConstants):
    ...
    players_per_group = 2
```

oTree Concepts #2 - Multiplayer Games

Groups

- If all players should be in the same group, or if it's a single-player game, set it to **`None`**:

```
class Constants(BaseConstants):
    # ...
    players_per_group = None
```

- In this case, **`self.group.get_players()`** and **`self.subsession.get_players()`** has the same behavior.
- Each player has an attribute **`id_in_group`**, which will tell you if it is player 1, player 2, etc.

oTree Concepts #2 - Multiplayer Games

Getting players

Group objects have the following methods:

- **`get_players()`**: Returns a list of the players in the group (ordered by `id_in_group`).
- **`get_player_by_id(n)`**: Returns the player in the group with the given `id_in_group`.

oTree Concepts #2 - Multiplayer Games

Getting players

- **`get_player_by_role(r)`**: Returns the player with the given role. If you use this method, you must define the role method. For example:

```python
class Group(BaseGroup):
    def set_payoff(self):
        buyer = self.get_player_by_role('buyer')

class Player(BasePlayer):
    def role(self):
        if self.id_in_group == 1:
            return 'buyer'
        return 'seller'
```

oTree Concepts #2 - Multiplayer Games

Getting other players

`Player` objects have methods **`get_others_in_group()`** and **`get_others_in_subsession()`** that return a **`list`** of the other players in the group and subsession. For example, with 2-player groups you can get the partner of a player:

```
class Player(BasePlayer):

    def get_partner(self):
        return self.get_others_in_group()[0]
```

oTree Concepts #2 - Multiplayer Games

Group matching - Fixed matching

- By default, in each round, players are split into groups of size **`Constants.players_per_group`**.
- They are grouped sequentially – for example:

  if there are 2 players per group, then P1 and P2 would be grouped together, and so would P3 and P4, and so on.

- **`id_in_group`** is also assigned sequentially within each group.
- This means that by default, the groups are the same in each round, and even between apps that have the same **`players_per_group`**.
- If you want to rearrange groups, you can use the next techniques.

oTree Concepts #2 - Multiplayer Games

Group matching - **`group_randomly()`**

- Subsessions have a method **`group_randomly()`** that shuffles players randomly, so they can end up in any group, and any position within the group.
- For example, this will group players randomly each round:

```
class Subsession(BaseSubsession):
    def creating_session(self):
        self.group_randomly()
```

oTree Concepts #2 - Multiplayer Games

Group matching - `group_randomly()`

- If you would like to shuffle players between groups but keep players in fixed roles, use **`group_randomly(fixed_id_in_group=True)`**:

```
class Subsession(BaseSubsession):
    def creating_session(self):
        self.group_randomly(fixed_id_in_group=True)
```

oTree Concepts #2 - Multiplayer Games

Group matching - `group_like_round()`

- To copy the group structure from one round to another round, use the **`group_like_round(n)`** method.
- The argument to this method is the round number whose group structure should be copied.
- In the below example, the groups are shuffled in round 1, and then subsequent rounds copy round 1's grouping structure.

```python
class Subsession(BaseSubsession):

    def creating_session(self):
        if self.round_number == 1:
            # <some shuffling code here>
        else:
            self.group_like_round(1)
```

oTree Concepts #2 - Multiplayer Games

Group matching - **get_group_matrix()**

- Subsessions have a method called **get_group_matrix()** that return the structure of groups as a matrix, i.e. a list of lists, with each sublist being the players in a group, ordered by **id_in_group**.
- The following lines are equivalent.

```
matrix = self.get_group_matrix()
# === is equivalent to ===
matrix = [
    group.get_players()
    for group in self.get_groups()]
```

oTree Concepts #2 - Multiplayer Games

Group matching - `set_group_matrix()`

- **`set_group_matrix()`** lets you modify the group structure in any way you want.
- You can modify the list of lists returned by `get_group_matrix()`, using regular Python list operations, and then pass this modified matrix to **`set_group_matrix()`**.

oTree Concepts #2 - Multiplayer Games

Group matching - `set_group_matrix()`

- Here is how this would look in `creating_session`:

```python
class Subsession(BaseSubsession):
    def creating_session(self):
        matrix = self.get_group_matrix()
        for row in matrix:
            row.reverse()
        self.set_group_matrix(matrix)
```

oTree Concepts #2 - Multiplayer Games

Group matching - **`set_group_matrix()`**

- You can also pass a matrix of integers. It must contain all integers from **1** to the number of players in the subsession.
- Each integer represents the player who has that id_in_subsession. For example:

```
>>> new_structure = [[1, 3,  5],
...                  [7, 9, 11],
...                  [2, 4,  6],
...                  [8,10, 12]]
>>> self.set_group_matrix(new_structure)
>>> self.get_group_matrix()
```

- You can even use **`set_group_matrix`** to make groups of uneven sizes.

oTree Concepts #2 - Multiplayer Games

Group matching - **`group.set_players()`**

- If you just want to rearrange players within a group, you can use the method on **`group.set_players()`** that takes as an argument a list of the players to assign to that group, in order.
- For example, if you want players to be reassigned to the same groups but to have roles randomly shuffled around within their groups (e.g. so player 1 will either become player 2 or remain player 1), you would do this:

```
class Subsession(BaseSubsession):

    def creating_session(self):
        for group in self.get_groups():
            players = group.get_players()
            players.reverse()
            group.set_players(players)
```

oTree Concepts #2 - Multiplayer Games

Group matching - Shuffling during the session

- If your shuffling logic needs to depend on something that happens after the session starts, you should do the shuffling in a wait page instead of in `creating_session`
- For example, let's say you want to randomize groups in round 2 only if a certain result happened in round 1. You need to make a WaitPage with wait_for_all_groups=True and put the shuffling code in after_all_players_arrive:

```python
class ShuffleWaitPage(WaitPage):
    wait_for_all_groups = True

    def after_all_players_arrive(self):
        if some_condition:
            self.subsession.group_randomly()
```

oTree Concepts #2 - Multiplayer Games

Group matching - Shuffling during the session

- You should also use is_displayed() so that this method only executes once. For example:

```
class ShuffleWaitPage(WaitPage):
    wait_for_all_groups = True

    def after_all_players_arrive(self):
        # [...shuffle groups for round 1]
        subsessions = self.subsession.in_rounds(
            2, Constants.num_rounds)
        for subsession in subsessions:
            subsession.group_like_round(1)

    def is_displayed(self):
        return self.round_number == 1
```

oTree Concepts #2 - Wait pages

- Wait pages are necessary when one player needs to wait for others to take some action before they can proceed.
- If you have a WaitPage in your sequence of pages, then oTree waits until all players in the group have arrived at that point in the sequence, and then all players are allowed to proceed.

```
class NormalWaitPage(WaitPage):
    pass
```

oTree Concepts #2 - Wait pages

- If your subsession has multiple groups playing simultaneously, and you would like a wait page that waits for all groups (i.e. all players in the subsession), you can set the **`attribute wait_for_all_groups = True`** on the wait page, e.g.:

```
class AllGroupsWaitPage(WaitPage):
    wait_for_all_groups = True
```

oTree Concepts #2 - Wait pages

Methods - `after_all_players_arrive()`

- Any code you define here will be executed once all players have arrived at the wait page.
  For example, this method can determine the winner and set each player's payoff.

```
class ResultsWaitPage(WaitPage):
    def after_all_players_arrive(self):
        self.group.set_payoffs()
```

WARNING

- you can't reference **`self.player`** inside **`after_all_players_arrive`**, because the code is executed once for the entire group, not for each individual player.
- However, you can use **`self.player`** in a wait page's **`is_displayed`**.

oTree Concepts #2 - Wait pages

Methods - `is_displayed()`

- Works the same way as with regular pages. If this returns **`False`** then the player skips the wait page.
- If some or all players in the group skip the wait page, then **`after_all_players_arrive()`** may not be run.

oTree Concepts #2 - Wait pages

Methods - `group_by_arrival_time`

- If you set **`group_by_arrival_time = True`** on a `WaitPage`, players will be grouped in the order they arrive at that wait page:

```
class MyWaitPage(WaitPage):
    group_by_arrival_time = True
```

For example, if **`players_per_group = 2`**, the first 2 players to arrive at the wait page will be grouped together, then the next 2 players, and so on.
This is useful in sessions where some participants might drop out in something like consent pages.

oTree Concepts #2 - Wait pages

Methods - `group_by_arrival_time`

If a game has multiple rounds, you may want to only group by arrival time in round 1:
class MyWaitPage(WaitPage): group_by_arrival_time = True

```
def is_displayed(self):
    return self.round_number == 1
```

If you do this, then subsequent rounds will keep the same group structure as round 1. Otherwise, players will be re-grouped by their arrival time in each round.

oTree Concepts #2 - Wait pages

Methods - `group_by_arrival_time`

Notes:

- `id_in_group` is not necessarily assigned in the order players arrived at the page.
- `group_by_arrival_time` can only be used if the wait page is the first page in `page_sequence`
- If you use `is_displayed` on a page with `group_by_arrival_time`, it should only be based on the round number. **Don't use** `is_displayed` to show the page to some players but not others.
- If you need further control on arranging players into groups, use `get_players_for_group()`.

References

- http://otree.readthedocs.io/en/latest/
- http://blog.easylearning.guru/implementing-mtv-model-in-python-django/
- https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93co
- https://en.wikipedia.org/wiki/Django_(web_framework)
- https://www.quora.com/What-is-a-Full-Stack-Web-framework