A Simple Public Goods Game

Basic Design

Let's move to an interactive game and set up a simple repeated public goods game. In such a game subjects receive a budget and they have to decide how much of the budget to allocate between a private account and a group account. First, consider which design choices we have to make for a basic design of our baseline treatment. We will add treatments later.

- Constants
  - Group size n = 3
  - Budget B = 10
  - Multiplier m = 0.4
  - Number of repetitions = 2
- Variables
  - Player i's allocation x_i
  - Total amount in group account
- Pages of the Experiment
  - Choice to make a choice how much to allocate.
  - ChoiceWaitPage to wait for other players in the group to make their choice
  - Feedback to learn about the payoffs in the last round
  - FinalFeedback to learn about the total earnings from all rounds
- Payoff function
  - In a single round of the experiment, a subject earns profits depending on their own choice x_i and all players' choices x_j.
  - $\pi_i (x_i, x_j) = B - x_i + m \sum_{j}x_j$

App Setup

Open the terminal and make sure that you are in the oTree folder.

```
otree startapp pgg
```

Constants

Navigate to __init__.py and locate the constants class C. It is a pure convention that constants are written in upper case to distinguish them from variables. We want three players per group and the game will be repeated two times. We set the budget to 10 and the multiplier to 0.4.

```
class C(BaseConstants):
    NAME_IN_URL = 'pgg'
    PLAYERS_PER_GROUP = 3
    NUM_ROUNDS = 2
    BUDGET = cu(10)
    MULTIPLIER = 0.4
```

cu() is short for currency. Using this function lets oTree understand that the amount in question are the currency unit in use. oTree will use this information to automatically format this constant as a currency amount when we show it to subjects.

Variables

Next, we set up the variables. Each player decides how much to allocate to the group account. We already used cu() to set up BUDGET as a currency amount, so we will set up a variable called contribution as a CurrencyField. Navigate to the Player class definition and add the contribution variable.

```
class Player(BasePlayer):
    contribution = models.CurrencyField()
```

Since this is a form that subjects will use we should set up a label for the form and some constraints. CurrencyField comes with some sensible defaults -- a subject must enter a number. If a subject tries to submit a string, they will get an error. But by default, subjects could enter any number. We want to constrict the action set by setting a min and max amount. We set min to zero and max to the budget. The budget is a constant, which we can access by calling C.BUDGET. Again, remember to separate the inputs to CurrencyField by commas.

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

Every player chooses their contributions, which are then summed up in the group account. So we define a variable called total on the group level. Again, we use a CurrencyField. In this case we do not have to set a min, max or label because this is not a variable that subjects directly interact with.

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

Pages

On the Choice page subjects decide how much of their budget to put into the group account. Just like in the consent page we have to tell oTree to expect the player to send information about the contribution variable. Scroll to the PAGES section and add a new Choice class.

```
class Choice(Page):
    form_model = 'player'
    form_fields = ['contribution']
```

After Choice, subjects have to wait for the other members of the group before payoffs can be calculated. So we define a wait page. By default, a player is stuck at this wait page until all other members of their group also arrive at this page. When all members have arrived, subjects automatically continue to the next page.

```
class ChoiceWaitPage(WaitPage):
    pass
```

Afterwards, there are two results pages. They are not interactive so we just declare simple pages.

```
class Feedback(Page):
    pass


class FinalFeedback(Page):
    pass
```

At the end, we set up the page sequence. The page sequence is a list of all the pages we declared previously.

```
page_sequence = [Choice, ChoiceWaitPage, Feedback, FinalFeedback]
```

Finally, we set up the three normal pages. Our wait page is automatically generated and does not require any additional input. Right-click on the pgg folder, select New File and create three files: Choice.html, Feedback.html and FinalFeedback.html.

Let's start with Choice.html. Again, we set up a title block. Since we repeat the game we can display in the title that a subject is currently on game X out of Y. You can call information about oTree objects by referencing them directly in the html code within double curly brackets. We already know that the total number of rounds is a constant that we can reference by calling C.NUM_ROUNDS. But what about the current round? oTree player objects automatically count which round they are in, so we can simply ask the player model which round it is in by calling player.current_round. Putting it all together we have a title block

```
{{ block title }}
    Round {{ player.round_number }} out of {{ C.NUM_ROUNDS }}
{{ endblock }}
```

Next we add a content block with a simple paragraph reminding subjects how much budget (C.BUDGET) they have available. Since budget is a currency amount it is automatically displayed as currency. Below we include the form. We could either tell oTree to include all formfields, as we did in the consent app, by calling {{ formfields }}, or we can reference a specific form to be displayed here by calling {{ formfield 'contribution'

}}. If we had different forms, we could display them in different places on the page. Finally, we add the submit button.

```
{{ block content }}
    <p>You have a budget of {{ C.BUDGET }}.</p>
    {{ formfield 'contribution' }}
    {{ next_button }}
{{ endblock }}
```

Next, let's define a simple feedback form. Open Feedback.html, set up a simple title that again counts the round and a submit button. We'll add the rest of the feedback later.

```
{{ block title }}
    Feedback {{ player.round_number }} out of {{ C.NUM_ROUNDS }}
{{ endblock }}

{{ block content }}
    {{ next_button }}
{{ endblock }}
```

Same for FinalFeedback.html

```
{{ block title }}
    Final Feedback
{{ endblock }}

{{ block content }}
    {{ next_button }}
{{ endblock }}
```

Testing the Setup

Before we continue we'll test if the setup works. We already have a session config, so we just add the public good game to our app sequence. Navigate to settings.py.

```
SESSION_CONFIGS = [
    dict(
        name='course',
        display_name='WZB oTree Course',
        app_sequence=['consent', 'pgg'],
        num_demo_participants=3,
    ),
]
```

Since we are already in settings.py, let's take a look at the currency definition. In our game we only declared a currency, but what the currency actually means is defined in settings.py. There are two lines to consider. REAL_WORLD_CURRENCY_CODE defines what currency is used for payments. USE_POINTS defines whether subjects first earn points that are later exchanged for actual money, or whether all choices are in actual money already. One wrinkle here is that oTree points are automatically rounded to the next integer, while currency is rounded two decimal places. With 10 units of currency and a multiplier of 0.4 there would be substantial amounts of rounding error with points, so either we scale up the points such that the rounding does not make a difference anymore or we use currency, where rounding matters less. Let's use Euro directly, without an experimental point currency in between.

```
REAL_WORLD_CURRENCY_CODE = 'EUR'
USE_POINTS = False
```

Save all the files and run otree devserver. It is a good idea to make changes incrementally and stop to check if the code works. Errors are easier to fix when there is one error rather than ten. Check that the Choice page looks as expected, the input form only accepts values within the defined range, the wait page is working and that the data is recorded in the data tab.

Calculating Payoffs

We record subjects' choices but we don't calculate any payoffs yet. A good moment to calculate payoffs is the wait page. Return to __init__.py and locate the ChoiceWaitPage class. oTree has a useful pre-defined function to do some calculations as soon as all players have arrived, aptly called after_all_players_arrive. We start by invoking @staticmethod -- it's a tool to attach a function to a class without having to tell the function exactly what class it is attached to.

```
class ChoiceWaitPage(WaitPage):
    @staticmethod
    def after_all_players_arrive(group: Group):
        pass
```

Now we get into the first real bit of programming. Within this function we want to

- sum up the contributions of all players for each group
- store the total contributions in the total group account
- and calculate payoffs for all players in each group.

The function already works on the group level. First, we start by getting a list of all players in the group and storing it in a local variable. This variable is deleted after the function is finished. We ask the group that is currently running this function to tell us which players are attached to it. This is a pre-defined oTree function which returns a list of players.

```
players = group.get_players()
```

Each of these players has chosen some contribution. Let's create a list of all the contributions in the group. We could do this cumbersomely by creating a for-loop but Python has a handy workaround. By invoking edged brackets we let Python know to create a list. Then, we tell Python to create this list by running an operation on every item of another list. We tell Python: Please use the list players. For each element, which I will call 'p' in this list, give me the contribution that is associated with this list, i.e. p.contribution. Finally, put everything back into a list.

```
contributions = [p.contribution for p in players]
```

Now we have a list of contributions and can easily sum it up. We can directly store this information in the group variable.

```
group.total = sum(contributions)
```

Given the total group account we can calculate the payout by multiplying it with the multiplier from the constants. Note that this payout in our payoff formula is the same for all players within a group. We simply use another local variable to store this value.

```
payout = group.total * C.MULTIPLIER
```

Now we have all the information to calculate payoffs for each player. We know their budget, their contribution, and their payout from the group account. We construct a simple for loop and calculate the payoff for each player element in the list of players. oTree comes with an automatically pre-defined payoff variable for each player, so we do not have to declare it.

```
for p in players:
    p.payoff = C.BUDGET - p.contribution + payout
```

Putting it all together, our wait page looks as follows:

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

Again, let's test it before we continue.

Round Feedback

Next, we will provide feedback to players about what happened in the current round. Let's start with simple feedback about the player's contribution, total contributions and the resulting payoff. These variables are easily added in Feedback.html. In the content block we add.

```
{{ block content }}
    <p>In total, all players contributed {{ group.total }} into the group account.</p>
    <p>You contributed {{ player.contribution }} and you earned {{ player.payoff }}.</p>
    {{ next_button }}
{{ endblock }}
```

But what if we want to provide more detailed feedback? Let's also provide the player with their partners' contributions. Accessing other players' variables is not possible directly through the html code though. To give players access to this information we have to push it through a function on the page's class definition. So navigate to class Feedback in __init__.py. We can use the pre-defined function vars_for_template to push data to the page. This is a function that operates on the player level.

```
class Feedback(Page):
    @staticmethod
    def vars_for_template(player: Player):
        pass
```

First, we ask the player object: Please give me a list of all players who are currently with you in your group. Then we use the same trick as before to construct a list of contributions and payoffs.

```
others = player.get_others_in_group()
others_contributions = [o.contribution for o in others]
others_payoffs = [o.payoff for o in others]
```

Next, we push this information to the page via a dictionary by using the command return.

```
return dict(
    others_contributions=others_contributions,
    others_payoffs=others_payoffs,
)
```

In the end our code looks as follows.

```
class Feedback(Page):
    @staticmethod
    def vars_for_template(player: Player):
        others = player.get_others_in_group()
        others_contributions = [o.contribution for o in others]
        others_payoffs = [o.payoff for o in others]
        return dict(
            others_contributions=others_contributions,
            others_payoffs=others_payoffs,
        )
```

We can now use these variables on our page. Return to Feedback.html. We can now access the values by calling {{ others_contributions }} but if we do so we just receive an ugly list. You can try it by adding the following line in the content block.

```
<p>Your partners contributed {{ others_contributions }} and earned {{
others_payoffs }}.</p>
```

To access the list you have to add a dot and a number. To get the first element we call {{ others_contributions.0 }}. Note that in many programming languages the first element is zero, not one. Replace the above line with two lines for the two partners.

```
<p>Partner 1 contributed {{ others_contributions.0 }} and earned {{
others_payoffs.0 }}.</p>
<p>Partner 2 contributed {{ others_contributions.1 }} and earned {{
others_payoffs.1 }}.</p>
```

Final Feedback

On the last page of the app we want to inform subjects about their total earnings. Subjects should see this page only after the last round. Navigate to the FinalFeedback class and add the is_displayed function. oTree will only display the page if the function returns the value True. Again, we set it up as a @staticmethod. In the function we simply check if the round that the player is currently in is the same as the maximum number of rounds in the game. In Python a logical check is implemented by two equal signs.

```
class FinalFeedback(Page):
    @staticmethod
    def is_displayed(player: Player):
        return player.round_number == C.NUM_ROUNDS
```

Player payoffs are reset every round to zero, but every payoff that a player earns is automatically added to the participant's payoff variable. Remember, the participant is the object that tracks the subject through the entire experiment. To display the total payoffs, we simply call {{ participant.payoff }} in FinalFeedback.html.

```
{{ block content }}
    <p>In total you earned {{ participant.payoff }}.</p>
    {{ next_button }}
{{ endblock }}
```

And we are done with our first, very basic experiment! Subjects can make choices and earn money. We will use this game as a basis for further modifications.