Competition Between Sellers

Let's set up a second app that we can use to showcase some additional oTree features. The public goods game that we set up previously is a symmetric game but many games have asymmetric roles. Let's set up a simple game with 2 sellers who sell a single unit of a good and a single buyer who wants to buy one single unit. First, the sellers have a marginal cost of zero and simultaneously post prices p for their good so their payoff is p if they sell their unit and zero otherwise. Second, the buyer observes the prices and chooses which seller to buy from. The buyer has to choose one of the offers and has a payoff function $$ \pi_B = 10 - p $$. Start a new app called competition.

```
otree startapp competition
```

Setting up init.py

Go to competition's initpy.py and set the constants up for three players per group (initially) and two repetitions.

```
class C(BaseConstants):
    NAME_IN_URL = 'competition'
    PLAYERS_PER_GROUP = 3
    NUM_ROUNDS = 2
```

We have three kinds of variables that we must define: the player's role, the offers and the buy decision. If I had a fixed number of sellers I could define everything as group variables: offer1, offer2, buy decision. But if we have more than two sellers I would need to introduce more group variables. So instead, I opt for player variables. It is not perfect since the offer variable for the buyer will be empty but that is not problematic. Variables can remain empty. Let's create three player variables: an offer, which is a currency amount, and a buy_from, which is an integer number and indicates which player's offer was chosen, and a game_role, which is a string. I'm avoiding role here because it is a pre-defined oTree variable that has a bit of an odd usage in my opinion.

```
class Player(BasePlayer):
    offer = models.CurrencyField(
        label="How much do you want to offer?",
        min = 0,
        max = 10
    )
    buy_from = models.IntegerField(
        label="Who do you want to buy from?",
        widget=widgets.RadioSelect()
        choices=[
            [1, "Player 1"],
            [2, "Player 2"]
        ]
    )
    game_role = models.StringField()
```

Next, we need to define the roles. There is one buyer but there are multiple sellers. The appropriate time to assign the role is when we create the session. Just like in our public goods game, create a new block # FUNCTIONS at the end of the init.py and define the automatically called creating_session function.

The logic for this function is straight-forward. We are getting a list of all groups in the subsession first. Then we loop through all the groups. For each group, we get a list of all the players in the group. Within each group each player has a pre-defined id_in_group variable, which is an integer variable defining the player's ID in the group, starting at 1. Loop through all players in the group and check their ID: If the player's ID is equal to the maximum number of players in each group, let this player be the buyer. Else, give this subject the role of a seller.

```
# FUNCTIONS
def creating_session(subsession: Subsession):
    groups = subsession.get_groups()
    for g in groups:
        players = g.get_players()
        for p in players:
            if p.id_in_group == C.PLAYERS_PER_GROUP:
                p.game_role = 'buyer'
            else:
                p.game_role = 'seller'
```

Next, we set up our pages. We need a page called Offer that only sellers see where they can decide what price to offer. While the sellers decide the buyer waits on a waitpage. Afterwards, there is a page called Decision on which the buyer decides what to buy. While the buyer makes their decision, the sellers wait on another wait page. Finally, we need a simple feedback form.

```
# PAGES
class Offer(Page):
    pass


class DecisionWaitPage(WaitPage):
    pass


class Decision(Page):
    pass


class ResultsWaitPage(WaitPage):
    pass


class Results(Page):
    pass


page_sequence = [Offer, DecisionWaitPage, Decision, ResultsWaitPage,
Results]
```

The Offer page is only displayed to sellers and gives the seller the option to make a choice.

```
# PAGES
class Offer(Page):
    @staticmethod
    def is_displayed(player: Player):
        return player.game_role == 'seller'

    form_model = 'player'
    form_fields = ['offer']
```

The Decision page is similar but we have to send a list of offers and the corresponding seller IDs. We gather all the data and use Python's zip function to put them together. zip takes in two (or more) lists and always takes the first two elements from both lists and puts them in a tuple, then takes the next two elements and so on. So we end up with a list of the structure [(Seller 1 ID, Seller 1 Offer), (Seller 2 ID, Seller 2 Offer)]

```
class Decision(Page):
    @staticmethod
    def is_displayed(player: Player):
        return player.game_role == 'buyer'

    form_model = 'player'
    form_fields = ['buy_from']

    @staticmethod
    def vars_for_template(player: Player):
        sellers = player.get_others_in_group()
        offers = [s.offer for s in sellers]
        seller_ids = [s.id_in_group for s in sellers]
        return dict(
            sellers_offers = list(zip(seller_ids, offers))
        )
```

The ResultsWaitPage can again be used to calculate payoffs. The logic is as follows: As soon as everybody has made their choices, we get the buyer in the group by getting the player with the highest ID in the group. Then we get the ID from the seller that they chose. We can use this ID to identify the seller and get their offer.

```
class ResultsWaitPage(WaitPage):
    @staticmethod
    def after_all_players_arrive(group: Group):
        buyer = group.get_player_by_id(C.PLAYERS_PER_GROUP)
        bought_from = buyer.buy_from
        seller = group.get_player_by_id(bought_from)
        price = seller.offer
```

Using all this information we can calculate the payoff. If the player's role is buyer, they earn 10 minus the price. If the player is a seller and got chosen, they receive the price. Else the payoff is zero.

```
class ResultsWaitPage(WaitPage):
    @staticmethod
    def after_all_players_arrive(group: Group):
        buyer = group.get_player_by_id(C.PLAYERS_PER_GROUP)
        bought_from = buyer.buy_from
        seller = group.get_player_by_id(bought_from)
        price = seller.offer
        players = group.get_players()
        for p in players:
            if p.game_role == 'buyer':
                p.payoff = 10 - price
            elif p.game_role == 'seller' and p.id_in_group == bought_from:
                p.payoff = price
            else:
                p.payoff = 0
```

Finally, on the Results page we want to give subjects feedback on which seller was chosen. We gather the same data as before in vars_for_template, just remember that now we are in a player function, not a group function. So we have to ask the player first, which group they are in. Also, include a dummy variable if the seller successfully sold to the buyer.

```
class Results(Page):
    @staticmethod
    def vars_for_template(player: Player):
        buyer = player.group.get_player_by_id(C.PLAYERS_PER_GROUP)
        bought_from = buyer.buy_from
        seller = player.group.get_player_by_id(bought_from)
        price = seller.offer
        seller_success = bought_from == player.id_in_group
        return dict(
            seller=bought_from,
            seller_success=seller_success,
            price=price,
        )
```

Pages

Set up three simple pages. Offer.html

```
{{ block title }}
    Offer
{{ endblock }}

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

Decision.html

```
{{ block title }}
    Decision
{{ endblock }}

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

and Results.html

```
{{ block title }}
    Results
{{ endblock }}

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

Let's leave Offer.html as it is. In Decision.html let's display the offers in a table. In html we can define a simple table by calling

```
<table class="table"> </table>
```

To load the oTree defaults for how a table should look, load the table class when generating the table. The first row of a table should be the header. We start a table row and add two table headers within it.

```
<tr>
    <th>Seller</th>
    <th>Offer</th>
</tr>
```

Next, we want to dynamically generate as many rows as there are sellers. We already pushed a list of sellers and their offers to the page. oTree can loop through all the elements with a simple for loop.

```
{{ for data in sellers_offers }}
{{ endfor }}
```

For each data element in this list of offers we want to create a row and then create two cells. Cell 1 should be the seller's ID, which is at position 0 of the tuple, and Cell 2 should be the offer, which is at position 1.

```
{{ for data in sellers_offers }}
    <tr>
        <td>{{ data.0 }}</td>
        <td>{{ data.1 }}</td>
    </tr>
{{ endfor }}
```

Putting it all together

```
{{ block title }}
    Decision
{{ endblock }}

{{ block content }}
    <table class="table">
        <tr>
            <th>Seller</th>
            <th>Offer</th>
        </tr>

        {{ for data in sellers_offers }}
            <tr>
                <td>{{ data.0 }}</td>
                <td>{{ data.1 }}</td>
            </tr>
        {{ endfor }}
    </table>

    {{ formfields }}
    {{ next_button }}
{{ endblock }}
```

Finally, we have to do some work on Results.html because this is a page that three types of players see: the buyer, the seller who sold and the seller(s) who did not sell. We can use if conditions to check the player role and remind the player in a short sentence. Then, in a second sentence we give feedback with an if, elif and else condition.

```
<p>You were
{{ if player.game_role == 'buyer' }}
     the buyer.
{{ else }}
    a seller.
{{ endif }}
</p>

<p>
    {{ if player.game_role == 'buyer' }}
    You bought from Seller {{ seller }} at price {{ price }}.
    {{ elif seller_success }}
    The buyer bought from you at price {{ price }}.
    {{ else }}
    The buyer bought from another seller at price {{ price }}.
    {{ endif }}
</p>
```

Finally, we add a new session config in settings.py just for this app.

```
SESSION_CONFIGS = [
    dict(
        name='competition',
        display_name='Competition',
        app_sequence=['competition'],
        num_demo_participants=3,
    ),
]
```

Run otree devserver and test the setup. You can also try and change the PLAYERS_PER_GROUP constant but remember to also change num_demo_participants.