oTree

An open-source platform for behavioral research

Dimitri DUBOIS
dimitri.dubois@umontpellier.fr

Content

Introduction

- licensed under the MIT open source license
- website: https://www.otree.org/
- documentation: https://otree.readthedocs.io/en/latest/
- citation:

  Chen, D.L., Schonger, M. & Wickens, C., 2016, "oTree - An open-source platform for laboratory, online, and field experiments", Journal of Behavioral and Experimental Finance 9, pp. 88-97

- can run on any device that has a web browser: desktop computer, tablet or smartphone
- need a configured web serveur with python and the django framework

Can be used for lab experiments

for lab-in-the-field experiments (here traders in Morocco)

Villagers in indonesia

for online experiments

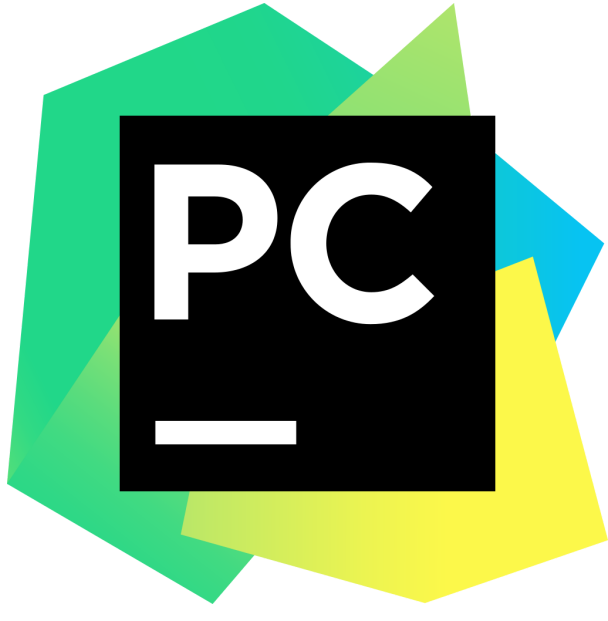
Installation

Python environment : Anaconda

- a Python distribution that includes all the packages needed for data science
- easy to install and manage
- https://www.anaconda.com/products/individual
- steps of the installation process, the installation of otree, and the creation of the otree project here

Pycharm

- Python IDE (Integrated Development Environment)
- https://www.jetbrains.com/pycharm/
- steps of the installation process and configuration for oTree here



To sum up

Anaconda

- installation of anaconda
- open anaconda navigator
- click on "Environments" and then create a new environment (called otree for example)
- once created, click on the green arrow and "Open terminal", and then in the console write `pip install otree`
- once done move to the desired directory with `cd` (for example cd C:\users\me)
- in the console write `otree startproject oTree`
- once done close the console and close anaconda navigator

Pycharm

- installation of pycharm (community version)
- once done, click on "new project", and in the location select the directory you've created with the startproject command
- for the python interpreter, select "previously configured interpreted", then "Conda environment" and get the environment created with anaconda navigator

Terminology

Session

- in experimental economics terminology a session begins when the participants are seated in the laboratory and ends when they leave the laboratory
- in otree terminology a session begins when the experimenter starts the program of the experiment and ends when he ends this program

Subsession

- a session may contains several parts, like for as example a public goods game, an investment game and a socio-demographic questionnaire
- in oTree each part is an application, each application is considered as a subsession

Group

- inside the subsession it is possible to form groups of participants
- the group class manages the participants that are grouped
- groups can be (re)formed at the beginning of each round

Participant

- refers to the subject
- one participant for each connection to the server in a session

Player

- each participant is a player in a subsession / application
- a participant can participate to several applications - in each application he/she is a player
- the player takes decisions and reads informations through the web pages

To sum up

- a session contains several participants and one or several applications / subsession
- each application / subsession contains all participants in the session
- a participant is a player in each subsession
- each subsession can be either one-shot or with several rounds
- each subsession may be composed of one or more groups of players

```
Session
    Subsession
        Group
            Player
```

Creation of an application, declaration of fields and creation of web pages

Practical example: a socio-demographic questionnaire

Creation of an application

- in pycharm's console (terminal): `otree startapp application_name`
- it creates a directory application_name and put inside the skeleton of the application
    - 1 python file : __init__.py
    - 2 html files: MyPage.html et Results.html
- the __init__ file contains one class for the parameters (C), three classes for the fields/attributes (Subsession, Group and Player) and three classes for the web pages (MyPage, ResultsWaitPage, Results)
- a list at the bottom of the file, called `page_sequence` set the order of the page sequence for each round

The __init__.py file

- class C: for constants / parameters that do not change during the session and are the same for all players
    - NAME_IN_URL: can be change
    - PLAYERS_PER_GROUP: None if individual decision, otherwise the size of the groups
    - NUM_ROUNDS: 1 if one-shot, otherwise the number of rounds to play
- class Subsession, Group, and Players allow to create fields/attributes, there will be stored in the database
    - **Subsession**: fields that are shared by all the players
    - **Group**: fields that are shared by the members of the group, i.e. group decisions or group values
    - **Player**: individual fields, to store individual decisions

The Player class

- to create player's fields, i.e. fields/attributes whose values can be different for each player. (*For the demographic questionnaire: gender, age, nationality etc.*
- these attributes are automatically stored in the database

Types of fields

- IntegerField or FloatField for numbers
- StringField or LongStringField for text
- BooleanField for binary values (True or False)
- CurrencyField for numbers converted in currency

Field creation

- all fields are classes of the `models` module in oTree, so it is necessary to write the code as

```
gender = models.IntegerField()
age = models.IntegerField()
```

Optional arguments

- **label**: the text next to the widget
- **min, max**: for float and integer field
- **choices**: to provide a list of choices
- **initial**: the default initial value
- **blank**: whether the field is mandatory (True) or not (False, default)
- **widget**: the type of widget to use to display the field on the screen

Choices

- a list of proposals

```
gender = models.StringField(
  label="Your gender",
  choices=["Woman", "Man"]
)
```

- can also be a list of tuples of size 2, in which the first element is the value stored in the database and the second the value that is displayed. Allows to store a numeric value (code) but to display a string value

```
gender = models.IntegerField(
  label="Your gender",
  choices=[(0, "Woman"), (1, "Man")]
)
```

Widgets

the type of widget to use to display the field on the screen

- graphical object that display the field
- Integer, float, currency, string and LongString: if no choices are provided, the type of widget cannot be changed
- if choices are provided, a dropdown list by default, but can be changed to RadioSelect or RadioSelectHorizontal
- BooleanField: by default 2 vertical RadioButton. Can be changed to RadioSelectionHorizontal or CheckboxInput

```
gender = models.IntegerField(
    label="Your gender",
    choices=[(0, "Woman"), (1, "Man")],
    widget=widgets.RadioSelectHorizontal
)
student = models.BooleanField(
    label="Are you student?",
    widget=widgets.RadioSelectHorizontal
)
```

Practice

- create a new application called demographics
- create the fields of the demographic questionnaire:
    - gender: woman / man
    - age
    - marital status: single, married, divorced, widowed
    - student: yes / no
    - level of study (Bac, Licence, Master, PhD)
    - field of study (Education, Sciences, Engineering, Law and Political Science, Agriculture and Food, Health, Economics and Management, Technical Studies)
    - monthly income (Less than 1000€, 1001€ - 2000€, 2001€ - 3000€, 3001€ - 4000€, 4001€ - 5000€, More than 5000€)

The classes for the html pages

- two types of pages: Page and WaitPage
- each class that inherits from Page must have the same name than a html file
- a Page is used either to collect data and/or to display information
- a WaitPage is to wait for the other players, either the group members or all the players in the subsession

Page that collects data

- after the class declaration, one must specify the form_model (either group or player) and the form_fields (list of fields to display and for which to collect a decision)
- fields put in form_fields must have the same name than the fields created in the Player class (or Group class if form_model is group)

```
class Questionnaire(Page):
    form_model = 'player'
    form_fields = ['gender', 'age', 'student']
```

- if the page only displays information let "pass" after class declaration

The html files

- html tags (p, ul, li, table etc.)
- css and javascript
- class C, Subsession, Group and Player are loaded, so that their attributes can be displayed

to collect the answer to a field

```
{{ formfields }}
```

to display all the fields given in the form_fields attribute of the corresponding page

```
{{ formfield "field_name" }}
```

to display the field field_name. This fields has be in the form_fields list of the corresponding page

to display the value of a field

```
{{ player.field_name }}
```

to display the value of the field field_name of the player class

The settings file

- once the application is created, necessary to add some information in the `SESSION_CONFIGS` list
- this list contains the list of the applications; each item of the list is a dictionnary
- the mandatory keys are: *name, display_name, app_sequence and num_demo_participants*
    - **name** is used internally, it can be whatever you want, but must be unique
    - **display_name** is what is displayed in the admin interface
    - **app_sequence** is a list with the applications that must be launched successively
    - **num_demo_participants** is the number of participants for a demo (must be compatible with the group size)

In this list of session configs, we can define an item with multiple applications, by putting multiple applications in the app_sequence. For example, if your experiment consists of a welcome app, a public good game, a socio-demographic questionnaire, and a final app, the item could be:

```
dict(
    name="public_good_experiment",
    display_name="Public Good Game - Complete",
    app_sequence=["welcome", "public_good", "demographics", "final"],
    num_demo_participants=4
)
```

Practice

- add the dict for the demographics application in the SESSION_CONFIGS list of the settings file
- rename MyPage.html to Questionnaire.html
- rename class MyPage in the __init__.py file, and in the page_sequence
- remove ResultsWaitPage and Results (not necessary in this application)
- write the html code of the file Questionnaire.html (use {{ formfields }})

To test the application

- in the pycharm terminal, write otree devserver
- it starts a local web server
- in your web browser: http://127.0.0.1:8000

Formation of groups, use of existing methods and creation of new methods

Practical example: the public goods game

The public goods game

- Fixed groups of size 4 (randomly formed in the first round, then fixed for the rest of the rounds)
- Endowment of 20 tokens
- Two accounts : private (1 token = 1 ECU), public (1 token = 0.50 ECU for each group member)
- 10 rounds
- Two screens per round : Decision and Results
- Results screen : player's investment in both accounts, total in the public account by the group and round payoff
- A final screen that displays the final payoff (cumulative payoff) in euros

Quick overview of the steps for writing the code of the application

- parameters (endowment, MPCR) must be set in the C class
- group formation at the beginning of the application (round 1), and then group stay unchanged
- individuals choose the amount they place in the public account (individual field)
- after all players in the group have taken their decision, the total in the public account is calculated and stored in the Group class (group field)
- individual payoff are calculated and stored in the Player class

The C class

- values that are identical for all the players and that won't be modified
- attributes in this class won't be stored in the database

```
PLAYERS_PER_GROUP = 4
ENDOWMENT = 20
MPCR = 0.5
```

The Subsession class

- fields that are the same for all the players
- declared fields are stored in the database
- two useful attributes: subsession.session and subsession.round_number (current round)
- several useful methods:
    - subsession.get_groups(): return a list with the groups
    - subsession.get_players(): return a list with all the players
    - subsession.group_randomly(): form random groups
    - subsession.group_like_round(round_number): set the same groups than in the round round_number

For the public goods game, no field in this class

The Group class

- fields that are the same for all the group members
- declared fields are stored in the database
- several useful methods:
    - group.get_players(): return a list with the group members (players)
    - group.in_round(round_number): return the group fields for the corresponding round
    - group.in_all_rounds(): return a list of the group fields for each round starting from the first one to the current one

```
class Group(BaseGroup):
    total_public_account = models.IntegerField()
```

The Player class

- individual fields, stored in the database
- useful methods:
    - get_others_in_group(): return a list with the other players of the group
    - in_round(round_number): return the player's fields for the corresponding round
    - in_previous_rounds(): return a list with the player's field for all the previous rounds
    - in_all_rounds(): return a list with the player's field for all the previous rounds + the current one
- useful attributes: subsession, group, round_number
- already created field: payoff (which is a CurrencyField)

```
class Player(BasePlayer):
    private_account = models.IntegerField()
    public_account = models.IntegerField(
        label="Enter the number of tokens you place in the public account",
        min=0, max=C.ENDOWMENT
    )
```

Additional methods

A "section" to put before the # Pages

A method to form the groups

```
def creating_session(subsession):
    if subsession.round_number == 1:
        subsession.group_randomly()
    else:
        subsession.group_like_round(1)
```

creating_session(subsession) is a method called when the session is started. The method is called for each application of the app_sequence and inside each application for each round

A method to compute the total number of tokens placed by the group members and the individual payoffs

```
def compute_total_public_account(group):
    group.total_public_account = sum(
        [p.public_account for p in group.get_players()]
    )
    for p in group.get_players():
        p.payoff = p.private_account + group.total_public_account * C.MPCR
```

we'll see later how/when to call this method

Practice

- write the code in the C, Group and Player classes
- write the methods to form the groups and to compute the total number of tokens placed in the public account by the group members (and the individual payoffs)

Management of the html pages in the __init__ file

- a class that inherits from Page inherits of the attributes and methods of the mother class
- three useful methods: `is_displayed(), vars_for_template() and before_next_page()`

is_displayed

can be set to define the condition under which the page is displayed. For example the page with the instructions of the game is displayed only in the first round, and the final page is displayed only if this is the last round.

```
class Instructions(Page):
    def is_displayed(player: Player):
        return player.round_number == 1

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

vars_for_template()

- used to send variables to the html page
- must return a dictionnary

```
class Instructions(Page):
    def is_diplayed(player: Player):
        return player.round_number == 1

    def vars_for_template(player: Player):
        return dict(
            nb_others_in_group=C.PLAYERS_PER_GROUP - 1
        )
```

then in the html file

```
You and the {{ nb_others_in_group }} members of your group have to decide ...
```

before_next_page()

- execute the instructions before loading the next_page

```
class Decision(Page):
    form_model = "player"
    form_fields = ["public_account"]

    def before_next_page(player: Player):
        player.private_account = C.ENDOWMENT - player.public_account
```

WaitPage

- by default, a wait page waits for all the players in the subsession. Set `wait_for_all_groups = False` to wait only for the group members.
- if a method must be called once the players arrive at the wait page, override the method `after_all_players_arrive`
    - either subsession or group as the parameter in the signature of the method depending on the value of wait_for_all_groups: if True the method is called only once, for all the players in the subsession, while if False, the method is called for each group.

by personal convention I add WaitForGroup to class name if wait_for_all_groups = False (I wait for the group members), and WaitForAll if wait_for_all_groups = True (I wait for all the players of the subsession)

```
class DecisionWaitForGroup(WaitPage):
    wait_for_all_groups = False

    def after_all_players_arrive(group: Group):
        compute_total_public_account(group)
```

Manual wait page

- if the experimenter wants to read the instructions aloud after subjects have read them silently, i.e. the experiment wants to control the moment at which the next page will be displayed
- create a page without any button, so that subjects are forced to wait
- use "Advance slowest user" on the administration page (server side), in the "Monitor" tab
- by personal convention the name of this kind of page ends with "WaitMonitor".

Pratice

- write the code of the public goods game (description here)
- once the application works, add the demographics questionnaire at the end of the game (app_sequence in the entry keys in the settings file)

Role assignment, asymmetric web page sequencing and use of conditional structure in web pages

Practical example : the investment game

The investment game

- pairs of players
- 2 roles : trustor and trustee, 1 trustor and 1 trustee in each pair
- both roles endowed with 10 Euros (or ECUs)
- trustor chooses how many Euros/Ecus he/she sends to the trustee
- each amount sent by the trustor is tripled by the experimenter, i.e. the trustee receives 3 times the amount sent by the trustor
- the trustee sends back how many Euros/Ecus he/she wants, between zero and the receveid amount
- Trustor's payoff : 10 - amount sent + amount returned by the trustee
- Trustee's payoff : 10 + 3 x amount sent by the trustor - amount sent back to the trustor

Roles

- roles can be defined in the C class, it must end with `_ROLE`
- roles are automatically set in the group, in order they are defined in the C class

```
class C(BaseConstants):
    PLAYERS_PER_GROUP = 2
    TRUSTOR_ROLE = "trustor"
    TRUSTEE_ROLE = "trustee
```

The first player in the group (`id_in_group = 1`) will have the role of trustor and the second one (`id_in_group = 2`) the role of trustee.

Remark
If in the group several players must have the same role, for example 2 buyers and 2 sellers in a group of 4, this doesn't work. In that case it is necessary to create a dedicated field in the player class (market_role for example) and to set the value depending on the id_in_group attribute.

player.role can be used as a condition to display a page

```python
class DecisionTrustor(Page):
    def is_displayed(player: Player):
        return player.role == C.TRUSTOR_ROLE


class DecisionTrustee(Page):
    def is_displayed(player: Player):
        return player.role == C.TRUTEE_ROLE
```

Conditional structure in the html file

```
<p>
    You have the role of
    {{ if player.role == C.TRUSTOR_ROLE }}
        trustor.
    {{ else }}
        trustee.
    {{ endif }}
</p>
```

- exists also `{{ elif }}` if needed

Display a value as a currency

- add `|cu` filter to the variable

```
<p>
    You have an endowment of {{ C.ENDOWMENT|cu }}.
</p>
```

will display "You have an endowment of €10.00".

Set min, max or choices of a field dynamically

- in the investment game, the maximum the trustee can send back to the trustor is the amount she received. But this value is not known before the trustor makes her decision
- possible to set the maximum value of a field after its creation with `def field_name_max()`

```
def amount_sent_back_max(group):
    return group.amount_received
```

Practice

- write the code of the (one-shot) investment game (description here)

Four tips for improving existing applications

Put the fields in a table with a for loop in the html file

By default, if we let `{{ formfields }}` in the web page and there are several fields, the labels and fields are one below the other, as in the socio-demographic application for example. To change that we can put the labels and fields in a table.

```
<table class="table">
    {{ for field in form }}
    <tr>
        <td>{{ field.label }}</td>
        <td>{{ field }}</td>
    </tr>
    {{ endfor }}
</table>
```

Create a history

- in repeated games, like the public good game, it is a good practice to let the players consult the history of the game, i.e. the value of the variables in the past rounds
- as previously shown, the for loop and with a table allows the creation of a history
- two useful methods of the player class: player.in_all_rounds and player.in_previous_rounds can be called in the web page

```
<table class="table">
    <thead>
        <tr>
            <th>Round</th>
            <th>Private account</th>
            <th>Public account</th>
            <th>Total public account</th>
            <th>Payoff</th>
        </tr>
    </thead>
    <tbody>
        {{ for p in player.in_all_rounds }}
        <tr>
            <td>{{ p.round_number }}</td>
            <td>{{ p.private_account }}</td>
            <td>{{ p.public_account }}</td>
            <td>{{ p.group.total_public_account }}</td>
            <td>{{ p.payoff }}</td>
         </tr>
        {{ endfor }}
    </tbody>
</table>
```

Set some parameters in the session configs entry and get them in the application

- useful when you plan to have several treatments
- for example if you want to run the public good game with a different MPCR
- either you change the value in the C class
- or you export this parameter and put it in the session configs. In that case, if you want to change the value of the MPCR, you do it in the session configs, not in the code of the application
- the change in the session configs can be done on the admin interface when creating a session

```
SESSION_CONFIGS = [
    dict(
        name="public_goods",
        display_name="Public Goods Game",
        app_sequence=["public_goods"],
        num_demo_participants=4,
        mpcr=0.5
    ),
]
```

Then, in the creating_session method

```
class SubSession(BaseSubsession):
    mpcr = models.FloatField()

# ADDITIONAL METHODS
def creating_session(subsession):
    subsession.mpcr = subsession.session.config["mpcr"]
```

Whenever you used C.MPCR in the application, replace it with subsession.mpcr.

Include an image in the web page

- create a subfolder in the `_static` directory with the name of your application (or at the root of the static folder if you plan to use the picture in several applications).
- in the html file, inside the img tag, add static in the src attribute

```
<img src="{{static 'application_name/picture_name.png' }}" />
```

- works the same way for video files and audio files, but it is better to put this kind of file elsewhere on the web and to just add the link in the src attribute

Practice

- put the fields of the socio-demographic questionnaire in a table
- create an history in the public goods game application
- export the mpcr parameter in the settings file
- create a welcome application with the picture of your lab and add it in the app_sequence of the public good game

Tests and simulations

- useful to test the code and generate data to check that everything is fine (values of the variables, calculations etc.)
- create a tests.py file
- add the import

```
from otree.api import *
from . import *
import random
```

- create a class PlayerBot(Bot) and add a method play_round()
- right now with the current version of oTree 5, it is necessary to use `self`

```
class PlayerBot(Bot):
  def play_round(self):
      yield Welcome
```

- yield each page of the page_sequence
- in the console, write `otree test welcome`

If the page has some fields, provide a dictionnary with the field name as key and a value

```
class PlayerBot(Bot):
    def play_round(self):
        if self.player.round_number == 1:
            yield Instructions

        yield Decision, dict(public_account=random.randint(0, C.ENDOWMENT))
        yield Results

        if self.player.round_number == C.NUM_ROUNDS:
            yield Final
```

If the page is without a submitt (or next) button, then yield a Submission, with the argument `check_html=False`

```
yield Submission(InstructionsWaitMonitor, check_html=False)
```

- by default, run `num_demo_participants` participants, but possible to set a different value by adding a number at the end of the command
  `otree test welcome 10` runs 10 participants
- add `--export path` to export the simulated data in the *path* folder
  `otree test public_goods 20 --export /tmp` to export the simulated data in the */tmp* folder

Pratice

- write the tests file of the public goods game, the investment game and the socio-demographic questionnaire
- create simulated data of the applications (app_sequence=["public_goods", "investment_game", "demographics"])

Data preprocessing

The csv files generated by otree need to be prepared for data analysis :

- remove unnecessary columns
- rename some columns
- check the series' type
- sometimes create new variables
- merge the tables

Can be done either with R or Python (pandas module in a jupyter-notebook)

```
pgg = pd.read_csv("...")  # load the csv file
print(pgg.shape)
print(pgg.columns.to_list())
pgg = pgg[[columns_kept]]  # keep only the needed variables

# rename the kept variables to facilitate the manipulation
pgg = pgg.rename(columns={"participant.code": "participant", "session.code": "session"})
pgg = pgg.rename(columns={c: c.split(".")[1] for c in df_pgg.columns if "." in c})

# the same with the investment game and the socio-demographics data frame
# ...

# merge the two data frames in a new data frame, with the participant as the key
df = pd.merge(pgg, investment_game, on=["participant"], how="left")
df = df.merge(demographics, on=["participant"], how="left")
```

Practice

- open a jupyter-notebook and prepare the data for analysis
- if you prefer, you can prepare the data analysis with R

The admin side: rooms, payments and admin report

Rooms

- a room is a virtual lab
- each participant has an id (participant_label) and has to enter this id to enter the room
- allows to create individual links, and avoids multiple connections by the same participant
- participant_label is one of the columns displayed in the monitor tab and in the Payments tab in the admin interface

To create a room

- create a folder named `_rooms` at the root of your otree project
- inside this folder create a text file (myroom.txt for example)
- inside this text file, add a participant label per row. It could be whatever you want

```
lab_001
lab_002
lab_003
lab_004
```

In the settings file, add a `ROOMS` list and add a dictionnary
An item for each room (each item being a dictionnary)

```
ROOMS = [
    dict(
        name="my_room",
        display_name="My Room",
        participant_label_file="_rooms/myroom.txt",
    )
]
```

In the admin interface, tab Rooms

- click on the room "My Room"
- connect clients to the room at http://127.0.0.1:8000/rooms/my_room
- clients enter a label (one of those in the text file)

Easy to send to a participant his personal link

- in a lab or on internet, provide directly individual links, such that http://127.0.0.1:8000/rooms/?participant_label=lab_001
- then wait to have the number of participants needed for the experiment, and once done, start the session
- once the session is started, open the Monitor tab to follow the players during the experiment
- once the session is finished, click on the Rooms tab, and close the room

Practice

- create a room
- start a session in this room

Payments

- the tab "Payments" displays the payoff of each participant
- the value displayed is given by the variable **participant.payoff**
- it is equal to the cumulative payoff since the beginning of the session, i.e. the cumulative value of **player.payoff** (for each round of each application)
- the value can be overriden by setting player.participant.payoff = new_value
- it is the case for example if we want to pay only one round or only one application

Pay only one round of the repeated game

- create a method and call this method just before the end of the application
- inside this method, select the paid round and set the value of this round's payoff to participant.payoff

```
# ADDITIONAL METHODS
def set_final_payoff(player):
    paid_round = random.randint(1, C.NUM_ROUNDS)
    player.participant.payoff = player.in_round(paid_round).payoff

# PAGES
class Results(Page):
    def before_next_page(player: Player):
        if player.round_number == C.NUM_ROUNDS:
            set_final_payoff(player)
```

Pay one application among several

- there exists an attribute in the participant class, called vars, which is a dictionnary, and in which you can store whatever you want
- the participant is the same in the different applications of the session, so you can access this "vars" dictionary from any application of the session
- so, in a method called at the end of each application you store the player's payoff for this application in the participant.vars dictionnary
- create a final application that displays the payoff of each application (with an explanation text)
- within this "final" application, you randomly select the application that will actually be paid

In each application of the experiment :

```python
# ADDITIONAL METHODS
def set_final_payoff(player):
    paid_round = random.randint(1, C.NUM_ROUNDS)
    txt_final = f"Round {paid_round} has been randomly selected to be paid"
    player.participant.vars["app_name"] = dict(
        payoff=player.in_round(paid_round).payoff,
        txt_final=txt_final
    )

# PAGES
class Results(Page):
    def before_next_page(player: Player):
        if player.round_number == C.NUM_ROUNDS:
            set_final_payoff(player)
```

In a final application (i.e. a dedicated application):

```
# ADDITIONAL METHODS
def select_paid_app(subsession):
    apps = ["public_goods", "investment_game"]  # the list of potentially paid applications
    for p in subsession.get_player():
        paid_app = random.choice(apps)
        p.participant.payoff = p.participant.vars[paid_app]["payoff"]

# PAGES
class BeforeFinalPage(WaitPage):
    wait_for_all_groups = True

    def after_all_players_arrive(subsession):
        select_paid_app(subsession)

class Final(Page):
    pass
```

Pratice

- use the procedure to pay only one round in the public goods game application

Admin report

- add a customized page to the admin interface, under the tab name "report"
- useful to create a customized payment page, or a page to display informations about the applications, or to display live statistics, either in a table or in a graph (with Highcharts, cf. infra)

Create a method **vars_for_admin_report(subsession)**, that returns a dictionnary

```
def vars_for_admin_report(subsession):
    infos_groups = list()
    for g in subsession.get_groups():
        trustor = g.get_player_by_role(C.TRUSTOR_ROLE)
        trustee = g.get_player_by_role(C.TRUSTEE_ROLE)
        infos_groups.append(
            dict(
                group_id=g.id_in_subsession,
                trustor=trustor.id_in_subsession,
                trustee=trustee.id_in_subsession,
                amount_sent=g.field_maybe_none("amount_sent"),
                amount_sent_back=g.field_maybe_none("amount_sent_back"),
                trustor_payoff=trustor.payoff,
                trustee_payoff=trustee.payoff
            )
        )
    return dict(infos_groups=infos_groups)
```

Create an html file name **admin_report.html**

```
<table class="table">
    <thead>
    <tr>
        <th>Group</th>
        <th>Trustor</th>
        <th>Trustee</th>
        <th>Amount sent</th>
        <th>Amount sent back</th>
        <th>Trustor's payoff</th>
        <th>Trustee's payoff</th>
    </tr>
    </thead>
    <tbody>
    {{ for g in infos_groups }}
    <tr>
        <td>g{{ g.group_id }}</td>
        <td>p{{ g.trustor }}</td>
        <td>p{{ g.trustee }}</td>
        <td>{{ g.amount_sent }}</td>
        <td>{{ g.amount_sent_back }}</td>
        <td>{{ g.trustor_payoff }}</td>
        <td>{{ g.trustee_payoff }}</td>
    </tr>
    {{ endfor }}
    </tbody>
</table>
```

Practice

- create an admin report, for the investment game and another one for the public good game

Internationalisation

Create a lexicon with the words or sentences translated in the different languages. For example, a file **lexicon_en.py**:

```
class Lexicon:
    amount_sent="You have to decide the amount you sent to the trustee."
    amount_sent_back="You have to decide the amount you sent back to the trustor."
```

and a file **lexicon_fr.py**

```
class Lexicon:
    amount_sent="Vous devez décider du montant que vous  envoyez au joueur B."
    amount_sent_back="Vous devez décider du montant que vous renvoyez au joueur A."
```

Import the language code and depending on it import the corresponding lexicon file

In the __init__ file in the import section

```
from settings import LANGUAGE_CODE

if LANGUAGE_CODE == 'en':
    from .lexicon_en import Lexicon
else:
    from .lexicon_fr import Lexicon
```

In the page class you add the lexicon in the vars_for_template method

```
class DecisionTrustor(Page):
    def vars_for_template(player: Player):
        return dict(
            language=LANGUAGE_CODE,
            lexicon=Lexicon
        )
```

and then in the html file

```
{{ lexicon.amount_sent }}
```

For long sentences or whole paragraphs, or if some variables are needed (as `player.payoff` for example), it is easier to use the if statement

```
{{ if language == "fr" }}
    <p>
        Un paragraphe en français
    </p>

{{ elif language == "en" }}
    <p>
        a paragraph in english.
    </p>

{{ endif }}
```

Pratice

- add a language for one of the applications

HTML, CSS and Bootstrap

HTML

- set of tags that are interpreted by the browser

Useful tags

- h1 to h6 are 6 levels of title
- p, br for paragraphs and break return
- table, thead, tbody, tr, th, td for tables
- ul, ol, li for list environment
- img for picture
- span, div, two kinds of blocks; span is an inline-block and div is a block

Introduction to HTML by the Mozilla fundation here

CSS

- Cascading Style Sheet
- set the style of the html tags
- either in the "style" attribute of a tag

```
<p style="font-size: 0.8em; color:
brown;">
  the sentences
</p>
```

or inside a style block

```
<style>
  p {
      font-size: 0.8em;
      color: brown;
  }
</style>
```

id attribute

In the previous example, all the p of the html page are concerned. To apply the style to only one p (or tag) you must define an id to the corresponding p

```
<p id="mon_p">
    my sentence
</p>
```

and then select the id (add a # before the name of the id to let the browser knows you're searching for an id)

```
<style>
    #mon_p {
        font-size: 0.8em;
        color: brown;
    }
</style>
```

class attribute

If it's for several p (tags) but not all of them, you define a class

```
<p class="my_p">
    my sentence
</p>
```

and then

```
<style>
    p.my_p {
        font-size: 0.8em;
        color: brown;
    }
</style>
```

It means `all the paragraph of class my_p`

Main style properties

- font-family, font-size, font-weight, font-style
- color, background-color
- text-align, text-decoration
- height, width, margin, padding

- List of all CSS properties
- Online CSS generator

Bootstrap

- library that defines a set of classes for many tags
- web-responsive
- Boostrap's website
- otree web interfaces are based on bootstrap
- you can add some nice graphical elements easily, mainly by setting value in the class attribute of the html tags (often div)

```
<div class="card bg-light">
    <div class="card-body">
        <p>Here some sentences</p>
    </div>
</div>
```

Main useful classes

- card: `<div class="card">...</div>` : to place one or more paragraphs inside a frame whose background color can be defined (doc)
- `<div class="row">...</div>` : to define a row, in wich you can set a grid with new divs and "col-x" class
- col-x: a row is divided into 12 columns, so you can set the number of cols a div may span, with `<div class="col-6">` for example
- look at boostrap grid to understand how it works
- table: table, table-bordered, table-sm, table-stripped etc. (doc)
- text alignement: text-justify, text-center, text-right
- text-color: text-danger, text-warning, text-info etc.
- margin: mt-1 to mt-5 for margin-top, mb-1 to mb-5 for margin-bottom, mx-1 to mx-5 for magin left and right, m-auto for margin-auto
- padding: pt-1 to pt-5 for padding-top, pb-1 to pb-5 for padding-bottom

Pratice

- improve the web interfaces of the public goods game, by using the card div (instructions and explanations sentences), and by changing the style of your history table

JavaScript

- programming language interpreted and executed on the client side, by the browser
- can access to the tags and change their css properties
- can add and handle events (click for example)

Introduction to JavaScript by the mozilla fundation here

- js can be added with a `<script>...</script>` tag at any place in the document
- also in event attributes of some html tags, like `onclick` for example
- or in a dedicated block at the end of the document

```
{{ block scripts }}
<script>
    js functions and instructions
</script>
{{ endblock }}
```

Main js rules

- ";" at the end of each instruction
- // for one line comments and / ... / for multiple lines comments
- main types: Number, String, Boolean, Array and Dictionnary
- Array is like a python list, can store different types of objects

```
let my_array = [1, false, "welcome"];
```

- Dictionnary is like in python:

```
let my_dict = {"key": "value"};
```

and you can access to the value with my_dict["key"]

Conditional structure

```
if (condition_1) {
    instructions;
} else if (condition_2){
    instructions;
} else {
    instructions;
}
```

For loop

```
for (let i=0; i<10; i++) {
    instructions repetead 10 times;
}

// array
for (let i in myArray)
    console.log(myArray[i]);

// or
for (let i of myArray) {
    console.log(i);  // display the value
}

// dictionnary
for (let [k, v] of Object.entries(my_dict))
    console.log(`${k}: ${v}`);
```

While

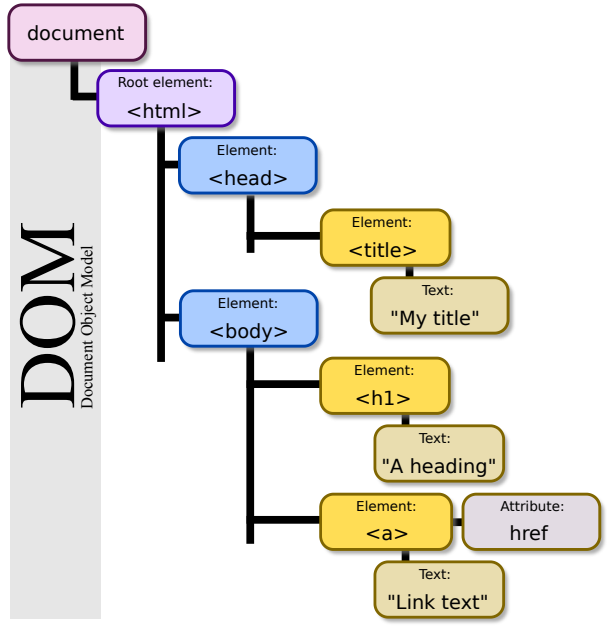
```
while(condition) {
    instructions;
}
```

To create your own function

```
function myFunction(arguments_needed) {
    instructions;
    return something
}
```

The document object model (DOM)

- created by the browser once the page is loaded
- list all the tags, their attributes and their values
- the root is the "document" object
- you can open the developer tools of your browser to see it



- can be called with js
- document.title: displays or changes the title (tab name)
- document.write(text): write the text at the place where the function is called

Selection of html tags with querySelector and querySelectorAll

- querySelector returns the first element that matches the request
- querySelectorAll return an array with all the elements that match the resquest
- the request can be a tag, an id, a class etc.
- you have a list of the existing selector here
- the easiest way is to set an id to the html element, and to select the element by its id, as in the example below. A `#` must be added to let the selector know that an id must be searched

In the html code (a hidden paragraph only displayed if needed)

```
<p id="error_message" class="text-danger" style="visibility: hidden"></p>
```

In the script section

```
<script>
let p_error = document.querySelector("#error_message");
p_error.innerHTML = "Please enter a value";
p_error.style.visibility = "visible";
</script>
```

Select an otree widget

- otree sets an id to each widget, the id is `id_field_name`, so a field "nationality" will have an id `id_nationality`

To get the value of the selected element just add `.value`

```
let nationality = document.querySelector("#id_nationality").value;
```

For radio buttons, to get the checked button

```
document.querySelector("input[name=field_name]:checked").value;
```

Possible to connect a function to on event, for example `onclick` for a button

```
<button type="button" onclick="validate();">Ok</button>
```

```
function validate() {
    let nationality = document.querySelector("#id_nationality").value;
    if (nationality === "") {
        let p_error = document.querySelector("p#error_message");
        p_error.innerHTML = "Please enter a value";
        p_error.style.visibility = "visible";
        return false;
    }
    return true;
}
```

Add an event listener to a widget

- useful when you want to display/hide or enable/disable a widget depending on the value of another

Example with a dropdown menu and the change event

```
<select id="my_select" >
  <option value="female">Female</option>
  <option value="male">Male</option>
</select>
<p id="result"></p>
```

with the js

```
let my_select = document.querySelector("#my_select");
my_select.addEventListener("change", function() {
    document.querySelector("#result").innerText = my_select.selectedOptions[0].innerText;
})
```

Example : Balloon Analogue Risk Task

The participant clicks on a button to inject some air in a balloon. Each injection the balloon has one half probability to explode. Each injections worths 0.5€ but if the balloon explodes the participant losts every accumulated amount. This task is used to measured the risk attitude of the participant.

```
let nb_inject = 0;  // number of injections

// we get the needed elements in the DOM
let btn_inject = document.querySelector("#btn_inject");
let nb_inject_area = document.querySelector("#nb_inject_area");  // a span in which we write the current
number of injections
let img_balloon = document.querySelector("#img_ballon");

btn_inject.addEventListener("click", function() {
    // each injection the balloon has a one half probability to explode
   if (Math.random() <= 0.5) {
       img_balloon.src = "{{ static 'bart/exploded.png' }}";  // we change the image (its path)
       return
   } else {
       img_balloon.style.width += 2;
       img_balloon.style.height += 2;
       nb_inject += 1;
       nb_inject_area.innerHTML = nb_inject;
   }
})
```

Practice

Write the BART application

Live method

- allows a live communication between the web page and the server
- useful for a chat application or a auction market application for example
- javascripts methods in the web page:
    - **liveSend(data)** to send data to the server (usually a dictionnary)
    - **liveRecv(data)** to receive the data from the server
- python method (in the correspond page class): `live_method(player: Player, data)`
- this method must return a dictionnary, with as key the player (id_in_group) to send the data to and as value the data
- to send to every group members the key must be 0

```python
class MyPage(Page):
    def live_method(player: Player, data):
        print(f"Received from player {player.id_in_group}: {data}")
        # ... do stuffs with the data
        return {0: data}  # here data is sent to every group member
```

Practice

- write a simple application where the player sends messages to everyone in her group

Highcharts

- js/css library to make graphics
- website: https://www.highcharts.com/
- installation: just load the library by adding

```
<script src="https://code.highcharts.com/highcharts.js">
</script>`
```