Contents

What is oTree?

What is oTree?

- platform to program social science experiments
- based on Django, which is a framework to develop web applications (ie applications that run on the browser, such as online sales, internet email)
- open software, Python language
- runs on any device with a browser (computer, tablet, cellphone)
- requires no software installed on the participant's device
- runs on the web (no local network needed) or on a local network (no internet needed)

What is oTree?

- excellent documentation: otree.readthedocs.io
- discussion forum with +5 posts a day
- given Python's and Django's popularity for tasks more general than experiments, one is likely to find solutions to any issues online

What is oTree?

- an experiment in oTree is a web app created in the Django framework
- oTree's value added: it initializes a project with the structure already in place for the app to function, and one only has to add the elements particular to the experiment
  - what participants will see on their screens
  - data participants have to enter
  - what to do with the data (calculations, storage)

The Shell and Python

The Shell and Python

The Shell

- a program that accepts text commands
- it can run other programs and navigate through the computer's file system
- we'll use it to install oTree, initialize an experiment, update a database, and launch the server
- useful commands:
  - pwd: print working directory
  - cd [*directory*]: change location
  - up arrow: previous command
  - copy-paste a directory: write directory address quickly

The Shell and Python

Python

- we'll complete an adaptation of the oTree's oficial documentation's Python tutorial
- we will work on an interactive Jupyter Notebook
- for this we need to install Python 3 and Jupyter

The Shell and Python

- to install Python 3
  - download Python 3 from https://www.python.org/downloads/
- to install Jupyter
  - open the Terminal (Mac) or Command Prompt (Windows)
  - $ *pip3 install –upgrade pip*
  - $ *pip3 install jupyter*

The Shell and Python

To create a Jupyter Notebook on the Desktop from the Shell

- open Terminal (Mac) or Command Prompt (Windows)
- navigate to the Desktop
- $ *jupyter notebook*
- this will launch the browser with the Notebook's server
- New/Python3

The Shell and Python

Alternative: create a Jupyter Notebook on the web without installing software (this requires stable internet connection for everyone):

- go to https://tmpnb.org/
- New/Python3

The Shell and Python

Alternative: create a Jupyter Notebook from PyCharm

- create a new Pure Python project
- go to Settings/Preferences $\rightarrow$ project:[name] $\rightarrow$ Project Interpreter $\rightarrow$ + $\rightarrow$ Jupyter
- File/New... $\rightarrow$ Jupyter Notebook
- type in commands and hit shift+enter
- go to the link shown by PyCharm, likely to be http://127.0.0.1:8888/

Example: simple questionnaire

Let's create a couple of apps to understand the general structure of an oTree project

Example: simple questionnaire

Example: simple questionnaire

What the app will do:

- Page 1: participant enters his name and age
- Page 2: display the participant's name and age

See *Demo Survey*.

Example: simple questionnaire

1. Initialize oTree project (directory with bare bones already in place)

   - navigate on the Shell to the desired location
   - create a project called project_cess
     - *otree startproject project_cess*
     - Include sample games? (y or n): n

Example: simple questionnaire

2. Create app that asks for the participant's name and age
   - sit on directory created
     - *cd project_cess*
   - initialize app called simple_questionnaire
     - *otree startapp simple_questionnaire*

Example: simple questionnaire

3. Open project on PyCharm to edit code
    - open PyCharm
    - Open
    - select folder project_cess

Example: simple questionnaire

To edit the code, we will go step by step through the tutorial on the *official documentation*.

In general there are 4 areas to work on for any oTree app:

- *models.py*: defines the database
- templates: contains each page shown to participants
  - in html, not Python
  - does not accept calculations nor the operations learned in the Python tutorial
- *views.py*: defines the logic and sequence of page presentation
- *settings.py*: defines general session configuration

Example: simple questionnaire

models.py

Define models.py

Open `models.py` and scroll to the line that says `class Player(BasePlayer):`. Here we can define what fields will be stored in the database for each player. Let's add 2 fields:

- `name` (which is a `CharField`, meaning text characters)
- `age` (which is a positive integer field)

```
class Player(BasePlayer):
    name = models.CharField()
    age = models.PositiveIntegerField()
```

Example: simple questionnaire

Templates

Define the template

This survey has 2 pages:

- Page 1: players enter their name and age
- Page 2: players see the data they entered on the previous page

In this section we will define the HTML templates that users see.

So, let's make 2 HTML files under `templates/my_simple_survey/`.

Example: simple questionnaire

Templates

Let's name the first page `MyPage.html`, and put these contents inside:

```
{% extends "global/Page.html" %}
{% load staticfiles otree_tags %}

{% block title %}
    Enter your information
{% endblock %}

{% block content %}

    {% formfield player.name with label="Enter your name" %}

    {% formfield player.age with label="Enter your age" %}

    {% next_button %}

{% endblock %}
```

Example: simple questionnaire

Templates

The second template will be called `Results.html`.

```
{% extends "global/Page.html" %}
{% load staticfiles otree_tags %}

{% block title %}
    Results
{% endblock %}

{% block content %}

    <p>Your name is {{ player.name }} and your age is {{ player.age }}.</p>

    {% next_button %}
{% endblock %}
```

Example: simple questionnaire

views.py

Define views.py

Now we define our views, which contain the logic for how to display the HTML templates.

Since we have 2 templates, we need 2 `Page` classes in `views.py`. The names should match those of the templates (`MyPage` and `Results`).

First let's define `MyPage`. This page contains a form, so we need to define `form_model` and `form_fields`. Specifically, this form should let you set the `name` and `age` fields on the player.

```
class MyPage(Page):
    form_model = models.Player
    form_fields = ['name', 'age']
```

- each page is a separate class (Page o WaitPage parent class)

Example: simple questionnaire

views.py

Now we define `Results`. This page doesn't have a form so our class definition can just say `pass`.

```
class Results(Page):
    pass
```

If `views.py` already has a `WaitPage`, you can delete that, because WaitPages are only necessary for multi-player games and more complex games.

- Waitpages are helpful to control the pace of the session, even in experiments with no subject interactions

Example: simple questionnaire

views.py

Then, set your `page_sequence` to `MyPage` followed by `Results`. So, all in all, `views.py` should contain this:

```
from otree.api import Currency as c, currency_range
from . import models
from ._builtin import Page, WaitPage
from .models import Constants


class MyPage(Page):
    form_model = models.Player
    form_fields = ['name', 'age']


class Results(Page):
    pass


page_sequence = [
    MyPage,
    Results
]
```

Example: simple questionnaire

settings.py

Define the session config in settings.py

Now we go to `settings.py` in the project's root directory and add an entry to `SESSION_CONFIGS`.

```
SESSION_CONFIGS = [
    {
        'name': 'my_simple_survey',
        'display_name': "My Simple Survey",
        'num_demo_participants': 3,
        'app_sequence': ['my_simple_survey'],
    },
    # other session configs go here ...
]
```

'app_sequence': ['simple_questionnaire']

Example: simple questionnaire

Reset the database and run

Enter:

```
otree resetdb
otree runserver
```

Then open your browser to `http://127.0.0.1:8000` to try out the survey.

Example: public goods game

Example: public goods game

The game:

- Groups of 3 members
- Each member receives 100 tokens
- Each member decides to contribute $0 \leq g_i \leq 100$ simultaneously
- Each member ends up with $2G/3 + 100 - g_i$

Example: public goods game

What the app will do:

- Form groups of 3 members
- Page 1: participant decides how much to contribute
- Calculate final earnings based on contributions from all group members
- Page 2: participant receives information on final earnings

Example: public goods game

As before, we'll work in 4 areas:

- *models.py*
- templates
- *views.py*
- *settings.py*
- this time we will add test bots, which will greatly simplify our life when testing the program

Example: public goods game

Let's creat an app inside the project already initialized (*project_cess*)

- sit on the directory created
  - *pwd* to make sure we are seated on project_cess
- initialize app with the name public_goods_game
  - *otree startapp public_goods_game*
- there should now be a folder in PyCharm called *public_goods_game*

We'll edit the code following the steps described in the *official documentation*.

Example: public goods game

models.py

Define models.py

Open `models.py`. This file contains the game's data models (player, group, subsession) and constant parameters.

First, let's modify the `Constants` class to define our constants and parameters – things that are the same for all players in all games. (For more info, see Constants.)

- There are 3 players per group. So, change `players_per_group` to 3. oTree will then automatically divide players into groups of 3.
- The endowment to each player is 100 points. So, let's define `endowment` and set it to `c(100)`. (`c()` means it is a currency amount; see Money and Points).
- Each contribution is multiplied by 2. So let's define `efficiency_factor` and set it to 2:

Example: public goods game

models.py

Now we have:

```
class Constants(BaseConstants):
    name_in_url = 'my_public_goods'
    players_per_group = 3
    num_rounds = 1

    endowment = c(100)
    efficiency_factor = 2
```

Example: public goods game

models.py

Now let's think about the main entities in this game: the Player and the Group.

After the game is played, what data points will we need about each player? It's important to know how much each person contributed. So, we define a field `contribution`, which is a currency (see Money and Points):

```
class Player(BasePlayer):
    contribution = models.CurrencyField(min=0, max=Constants.endowment)
```

What data points are we interested in recording about each group? We might be interested in knowing the total contributions to the group, and the individual share returned to each player. So, we define those 2 fields:

```
class Group(BaseGroup):

    total_contribution = models.CurrencyField()
    individual_share = models.CurrencyField()
```

Example: public goods game

models.py

Now let's define a method that calculates the payoff (and other fields like `total_contribution` and `individual_share`). Let's call it `set_payoffs`:

```
class Group(BaseGroup):

    total_contribution = models.CurrencyField()
    individual_share = models.CurrencyField()

    def set_payoffs(self):
        self.total_contribution = sum([p.contribution for p in self.get_players()])
        self.individual_share = self.total_contribution * Constants.efficiency_factor / Constan
        for p in self.get_players():
            p.payoff = Constants.endowment - p.contribution + self.individual_share
```

self.individual_share = self.total_contribution * Constants.efficiency_factor / Constants.players_per_group

Example: public goods game

Templates

Define the template

This game has 2 pages:

- Page 1: players decide how much to contribute
- Page 2: players are told the results

In this section we will define the HTML templates to display the game.

So, let's make 2 HTML files under `templates/my_public_goods/`.

Example: public goods game

Templates

The first is `Contribute.html`, which contains a brief explanation of the game, and a form field where the player can enter their contribution.

```
{% extends "global/Page.html" %}
{% load staticfiles otree_tags %}

{% block title %} Contribute {% endblock %}

{% block content %}

<p>
    This is a public goods game with
    {{ Constants.players_per_group }} players per group,
    an endowment of {{ Constants.endowment }},
    and an efficiency factor of {{ Constants.efficiency_factor }}.
</p>


{% formfield player.contribution with label="How much will you contribute?" %}

{% next_button %}

{% endblock %}
```

Example: public goods game

Templates

The second template will be called `Results.html`.

```
{% extends "global/Page.html" %}
{% load staticfiles otree_tags %}

{% block title %} Results {% endblock %}

{% block content %}

<p>
    You started with an endowment of {{ Constants.endowment }},
    of which you contributed {{ player.contribution }}.
    Your group contributed {{ group.total_contribution }},
    resulting in an individual share of {{ group.individual_share }}.
    Your profit is therefore {{ player.payoff }}.
</p>

{% next_button %}

{% endblock %}
```

Example: public goods game

views.py

Define views.py

Now we define our views, which contain the logic for how to display the HTML templates. (For more info, see Views.)

Since we have 2 templates, we need 2 `Page` classes in `views.py`. The names should match those of the templates (`Contribute` and `Results`).

Example: public goods game

views.py

First let's define `Contribute`. This page contains a form, so we need to define `form_model` and `form_fields`. Specifically, this form should let you set the `contribution` field on the player. (For more info, see Forms.)

```
class Contribute(Page):

    form_model = models.Player
    form_fields = ['contribution']
```

Example: public goods game

views.py

Now we define `Results`. This page doesn't have a form so our class definition can be empty (with the `pass` keyword).

```
class Results(Page):
    pass
```

Example: public goods game

views.py

We are almost done, but one more page is needed. After a player makes a contribution, they cannot see the results page right away; they first need to wait for the other players to contribute. You therefore need to add a `WaitPage`. When a player arrives at a wait page, they must wait until all other players in the group have arrived. Then everyone can proceed to the next page. (For more info, see Wait pages).

When all players have completed the `Contribute` page, the players' payoffs can be calculated. You can trigger this calculation inside the the `after_all_players_arrive` method on the `WaitPage`, which automatically gets called when all players have arrived at the wait page. Another advantage of putting the code here is that it only gets executed once, rather than being executed separately for each participant, which is redundant.

Example: public goods game

views.py

We write `self.group.set_payoffs()` because earlier we decided to name the payoff calculation method `set_payoffs`, and it's a method under the `Group` class. That's why we prefix it with `self.group`.

```
class ResultsWaitPage(WaitPage):

    def after_all_players_arrive(self):
        self.group.set_payoffs()
```

Now we define `page_sequence` to specify the order in which the pages are shown:

```
page_sequence = [
    Contribute,
    ResultsWaitPage,
    Results
]
```

Example: public goods game

settings.py

Define the session config in settings.py

Now we go to `settings.py` in the project's root directory and add an entry to `SESSION_CONFIGS`.

```
SESSION_CONFIGS = [
    {
        'name': 'my_public_goods',
        'display_name': "My Public Goods (Simple Version)",
        'num_demo_participants': 3,
        'app_sequence': ['my_public_goods', 'survey', 'payment_info'],
    },
    # other session configs ...
]
```

'app_sequence': ['public_goods_game']

Example: public goods game

Reset the database and run

Enter:

```
$ otree resetdb
$ otree runserver
```

Then open your browser to `http://127.0.0.1:8000` to play the game.

Example: public goods game

To combine simple_questionnaire and public_goods_game apps:

- In *settings.py*:
  - 'app_sequence': ['simple_questionnaire', 'public_goods_game']

Test bots

Test bots

- test bots simulate participants
- allow to test the code of an entire session in seconds
    - testing manually is very painful and time consuming
- helpful to check that the program runs to the end without error, and to verify that the data is being generated correctly according to the simulated decisions

Test bots

Steps:

- in *tests.py*, add an action for each field the participant must fill in
- in *settings.py*, add *'use_browser_bots': True*
- *$ otree resetdb*
- *$ otree runserver*

Test bots

tests.py for *simple_questionnaire* app

```
class PlayerBot(Bot):

    def play_round(self):
        yield (views.MyPage, {'name': 'El Bot', 'age': 18})
        yield (views.Results)
```

Test bots

tests.py for *public_goods_game* app

```
class PlayerBot(Bot):

    def play_round(self):

        if self.player.id_in_group == 1:
            yield (views.Contribute, {'contribution': c(100)})

        elif self.player.id_in_group ==2:
            yield (views.Contribute, {'contribution': c(50)})

        else:
            yield (views.Contribute, {'contribution': c(0)})

        yield (views.Results)
```

Test bots

settings.py

```
SESSION_CONFIGS = [
    {
        'name': 'primer_cuestionario',
        'display_name': 'Mi primer cuestionario',
        'num_demo_participants': 3,
        'app_sequence': ['cuestionario', 'bien_publico'],
        'use_browser_bots': True
    }
]
```