Tutorial #1 - oTree Objects

Juan Cabral - jbc.develop@gmail.com

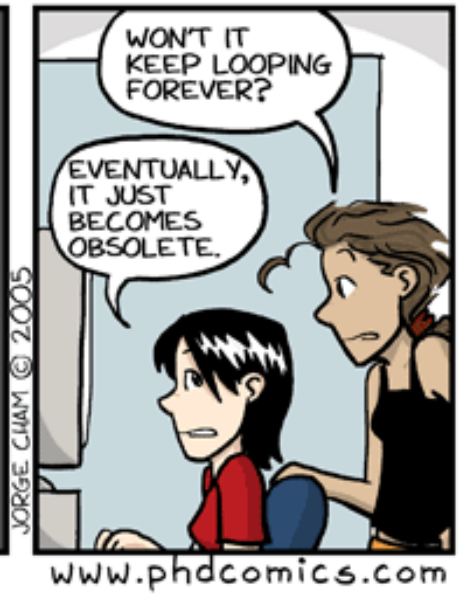
Jan, 2018

Tutorial #1: Public goods game.

```
% phd.m
%
% author: Cecilia
% date: 09/08/05

load THESIS_TOPIC

while (funding==true)
    data = run_experiment(THESIS_TOPIC);
    GOOD_ENOUGH = query(advisor);
    if (data > GOOD_ENOUGH)
        graduate();
        break
    else
        THESIS_TOPIC = new();
        years_in_gradschool += 1;
    end
end
```



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

Tutorial #1: Public goods game.

This is a three player game where each player is initially endowed with 100 points. Each player individually makes a decision about how many of their points they want to contribute to the group. The combined contributions are multiplied by 2, and then divided evenly three ways and redistributed back to the players.

Tutorial #1: Public goods game.

`models.Constants`

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.

- 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).
- Each contribution is multiplied by 2. So let's define multiplier and set it to 2.

Tutorial #1: Public goods game.

`models.Player`

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

Tutorial #1: Public goods game.

`models.Group`

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.
Finally let's define our payoff function. The argument to the function should be a group whose payoffs should be calculated.

Tutorial #1: Public goods game.

`views.py` and Templates

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

1. 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**.
2. The template contains a brief explanation of the game, and a form field where the player can enter their contribution.

Tutorial #1: Public goods game.

`views.py` and Templates

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

Tutorial #1: Public goods game.

`views.py` and Templates

Consideration

5. 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.

Tutorial #1: Public goods game.

Finally

- Edit the `views.page_sequence`
- Define the session in **sessions.py**
- Reset the database and run

Understanding oTree

oTree is a **Framework**

- A Framwork is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software
- Frameworks have key distinguishing features that separate them from normal libraries:
  - The overall program's flow of control is not dictated by the caller, but by the framework.
  - A user can extend the framework - usually by selective overriding
  - Users can extend the framework, but should not modify its code.

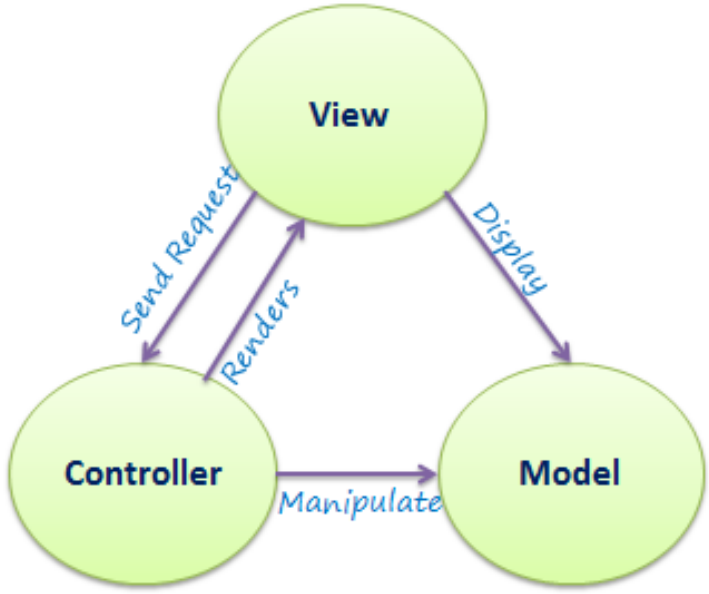
Understanding oTree

oTree is a **Model-View-Controller** (MVC) Framework

- The **model** is the central component of the pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface.[6] It directly manages the data, logic and rules of the application.
- A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- The **controller**, accepts input and converts it to commands for the model or view

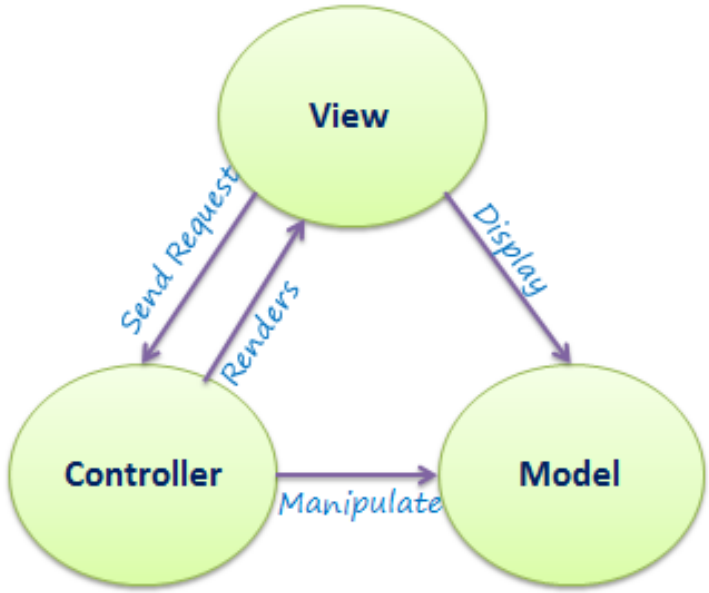
Understanding oTree

oTree is a **Model-View-Controller** (MVC) Framework



Understanding oTree

oTree is a **Model-View-Controller** (MVC) Framework



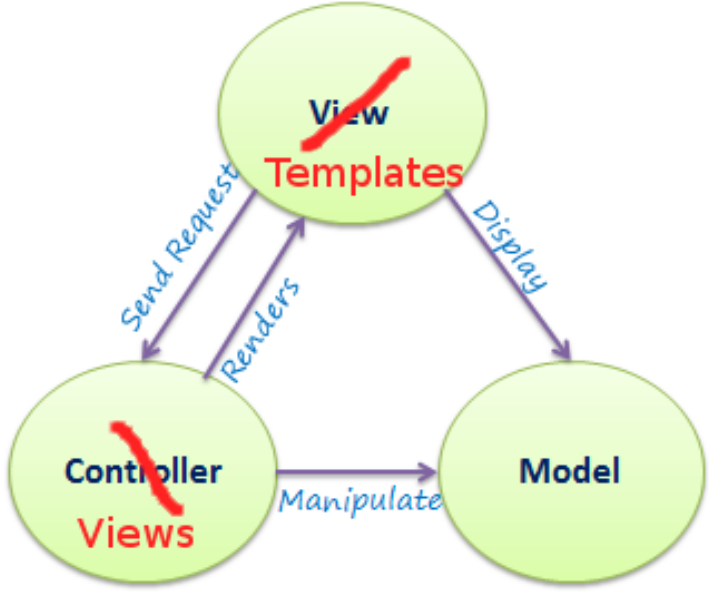
Wait!

Understanding oTree

~~oTree is a **Model-View-Controller** (MVC) Framework~~

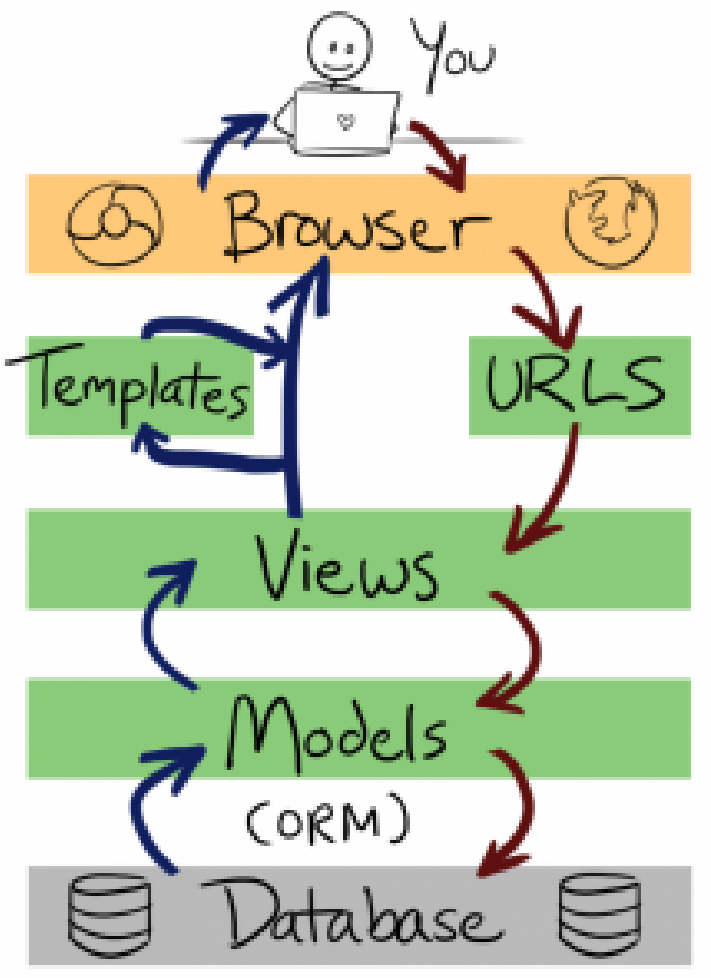
oTree is a **Model-View-Template** (MVT) Framework

- Model
- View => **controller**
- Template => **template**



Understanding oTree

~~Django~~ oTree workflow



oTree is a **Full-Stack Web Framework** based on **Django**

Django is a free and open-source web framework, written in Python, which follows the MVT architectural pattern. It is maintained by the Django Software Foundation (DSF), an independent organization.

Full stack is:

- Database
- Web Templates
- User Management
- URL Mapping

oTree vs Django

oTreee IS Django with some logic already defined

- The domain object model (DOM) is already defined.
- The URL mapping is automatic generted from `settings.SESSION_CONFIG` and `views.page_sequence`.
- The Page and WaitPages.
- The test bots (this is for tomorrow)

oTree Models

- Defined in `models.py`
- Is where you define your app's data models:
  - Subsession
  - Group
  - Player
- **Remember:** A player is part of a group, which is part of a subsession.

oTree Models

Model-Fields

- The main purpose of `models.py` is to define the columns of your database tables. Let's say you want your experiment to generate data that looks like this:

name	age	is_student
John	30	False
Alice	22	True
Bob	35	False
...		

- Here is how to define the above table structure:

```
class Player(BasePlayer):
    name = models.CharField()
    age = models.IntegerField()
    is_student = models.BooleanField()
```

oTree Models

Model-Fields Considerations

- When you run otree resetdb, it will scan your models.py and create your database tables accordingly. (Therefore, you need to run resetdb if you have added, removed, or changed a field in models.py.)

oTree Models

Model-Fields List

- The full list of available fields is in the **Django documentation**.
- The most commonly used ones are:
  - **CharField/TextField** (for text)
  - **FloatField** (for real numbers)
  - **BooleanField** (for true/false values)
  - **IntegerField**, and **PositiveIntegerField**.
- Additionally, oTree has **CurrencyField**

oTree Models

Model-Fields Configuration

- Any field you define will have the initial value of `None`.
- If you want to give it an initial value, you can use `initial=`:

```
class Player(BasePlayer):
    some_number = models.IntegerField(initial=0)
```

- Any numeric field support a minimun and maximun limits

```
offer = models.IntegerField(min=12, max=24)
```

- Also any field support a selection from a set of values

```
level = models.IntegerField(choices=[1, 2, 3])
```

oTree Models

Constant class

- The Constants class is the recommended place to put your app's parameters and constants that do not vary from player to player.
- Here are the required constants:
  - **`name_in_url`**: the name used to identify your app in the participant's URL.
    For example, if you set it to public_goods, a participant's URL might look like this:
    `http://host.com/p/zuzepona/public_goods/Introduction/1/`
  - **`players_per_group`**: described in Groups.
  - **`num_rounds`**: described in Rounds.

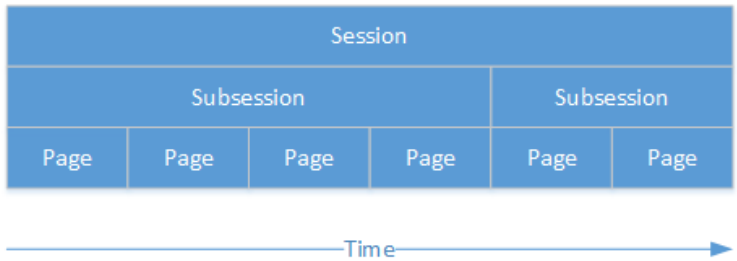
oTree Models - Subsession class

A session is a series of subsessions; subsessions are the "sections" or "modules" that constitute a session. For example:

> *if a session consists of a public goods game followed by a questionnaire:*
>
> - `the public goods game would be subsession 1`
> - `and the questionnaire would be subsession 2.`

In turn, each subsession is a sequence of pages the user must navigate through. For example:

> *if you had a 4-page public goods game followed by a 2-page questionnaire:*



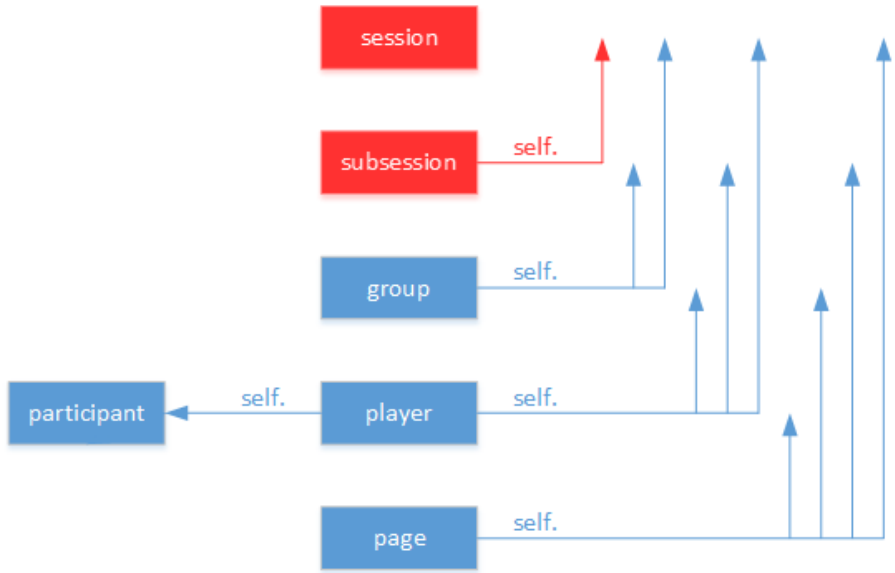
If a game is repeated for multiple rounds, **each round is a subsession**.

oTree Models

Subsession class

Here is a list of attributes and methods for subsession objects.

- **`session`** The session this subsession belongs to



- **`round_number`**: Gives the current round number. Only relevant if the app has multiple rounds (set in Constants.num_rounds).

oTree Models

Subsession class

- **`creating_session()` Method**: This method is executed when the admin clicks "create session"
  - allows you to initialize the round, by setting initial values on fields players, groups, participants, or the subsession. For example:

```
class Subsession(BaseSubsession):

    def creating_session(self):
        for p in self.get_players():
            p.some_field = some_value
```

- **`get_groups()`**: Returns a list of all the groups in the subsession.
- **`get_players()`**: Returns a list of all the players in the subsession.

oTree Models

Group class

Each subsession can be further divided into groups of players; for example:

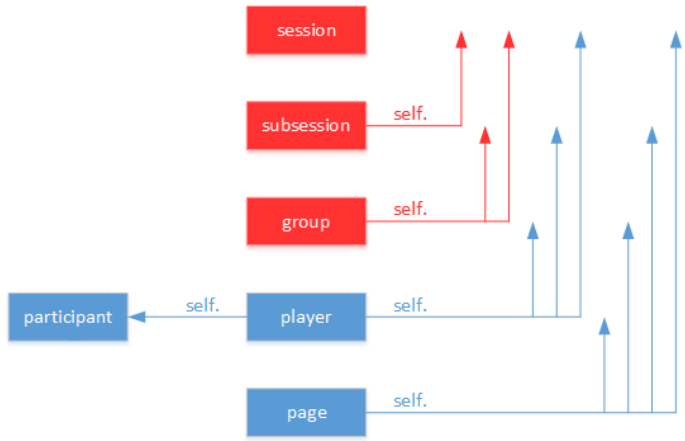
> *you could have a subsession with 30 players, divided into 15 groups of 2 players each. (Note: groups can be shuffled between subsessions.)*

oTree Models

Group class

Here is a list of attributes and methods for group objects.

- **`session`** and **`subsession`**




```
class Group(BaseGroup):
    def set_payoff(self):
        self.subsession.round_number
```

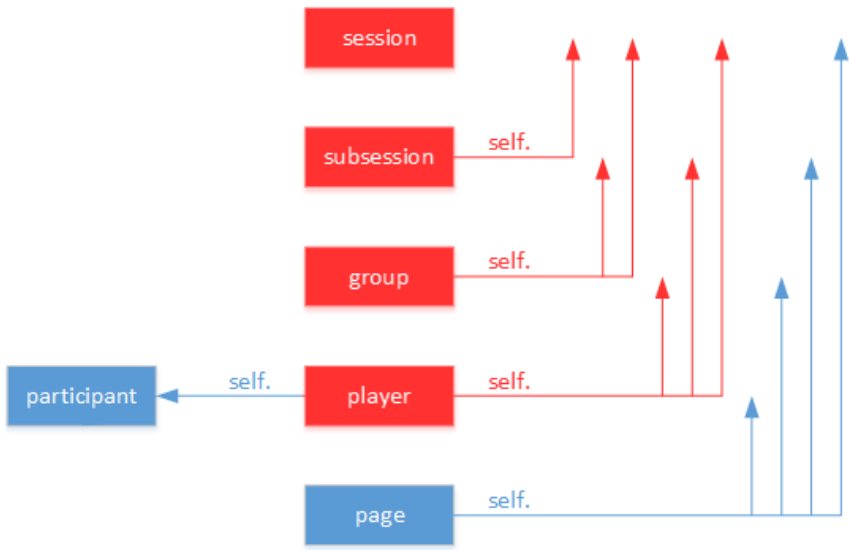
- **`get_players()`**: Returns a list of all the players in the subsession

oTree Models

Player class

Here is a list of attributes and methods for player objects.

- `group`, `session` and `subsession`



- `id_in_group` Integer starting from 1. In multiplayer games, indicates whether this is player 1, player 2, etc.
- `payoff` The player's payoff in this round.
- `get_others_in_group()`/`get_others_in_subsession()` list of another players in this group/subsession.

oTree Models

Player class

- `role()` You can define this method to return a string label of the player's role, usually depending on the player's `id_in_group`.
  For example:

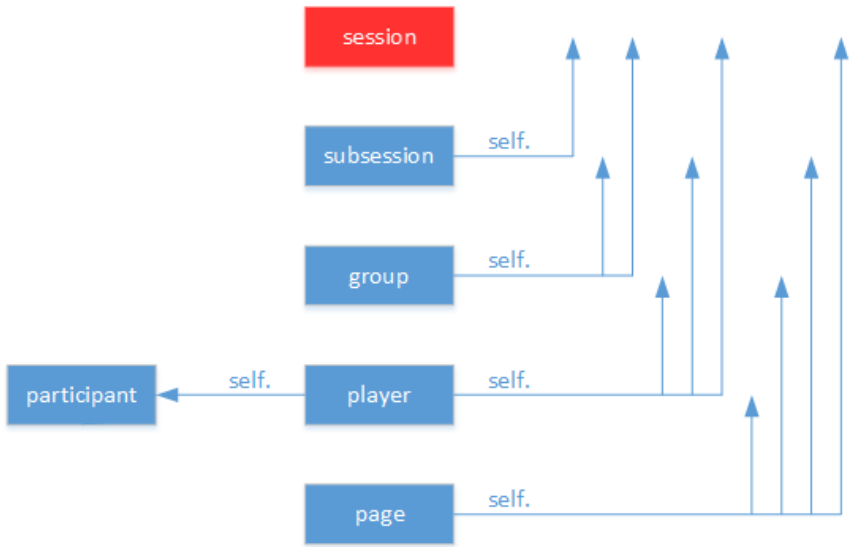
```
class Player(BasePlayer):
    def role(self):
        if self.id_in_group == 1:
            return 'buyer'
        if self.id_in_group == 2:
            return 'seller'
```

- Then you can use `group.get_player_by_role('seller')` to get player 2.
- Also, the player's role will be displayed in the oTree admin interface, in the "results" tab.

oTree Models - **Internals**

Session class

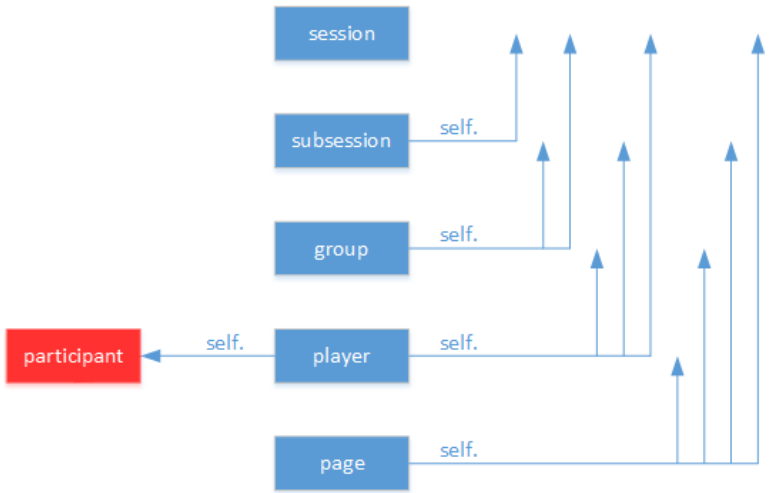
- `num_participants`: The number of participants in the session.
- `config`: Dict-like from `settings.SESSION_CONFIGS`
- `vars`: Dict-like to store global variables that are the same for all participants in the session



oTree Models - **Internals**

Participant class

- **`vars`**: Dict-like to store global variables that are the same for the participant in the session
- **`label`**: It will be used to identify that participant in the oTree admin interface and the payments page, etc.
- **`id_in_session`**: The participant's ID in the session.
- **`payoff`**: automatically stores the sum of payoffs from all subsessions (sum of all `player.payoff`)
- **`payoff_plus_participation_fee()`**: participant's total profit



Interlude - How oTree executes your code

- Any code that is not inside a method is basically global and will only be executed once – when the server starts.

```
class Constants(BaseConstants):

    heads_probability = random.random() # wrong


class Player(BasePlayer):

    heads_probability = models.FloatField(
        initial=random.random())  # wrong
```

Interlude - How oTree executes your code

- The solution is to generate the random variables inside a method, such as creating_session.

```
class Subsession(BaseSubsession):

    def creating_session(self):
        for p in self.get_players():
            p.heads_probability = random.random()
```

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