Tutorial #1 - oTree Objects

Juan Cabral - jbc.develop@gmail.com

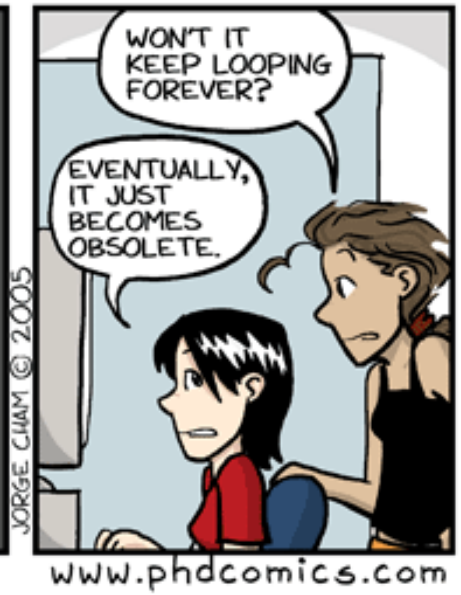
Jan, 2018

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.

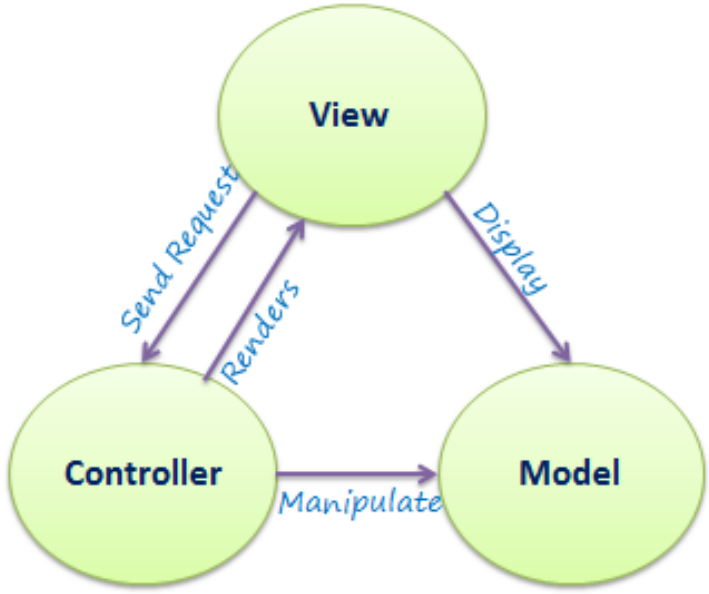
Understanding oTree

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

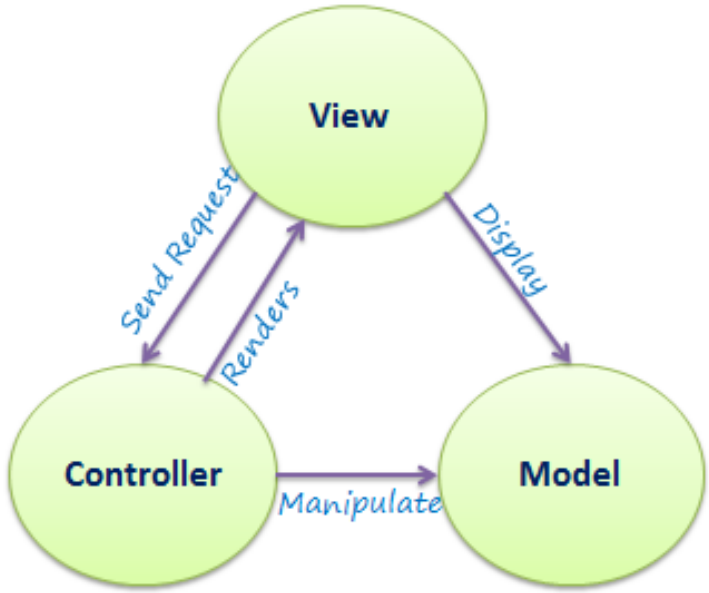
Understanding oTree

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



Understanding oTree

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



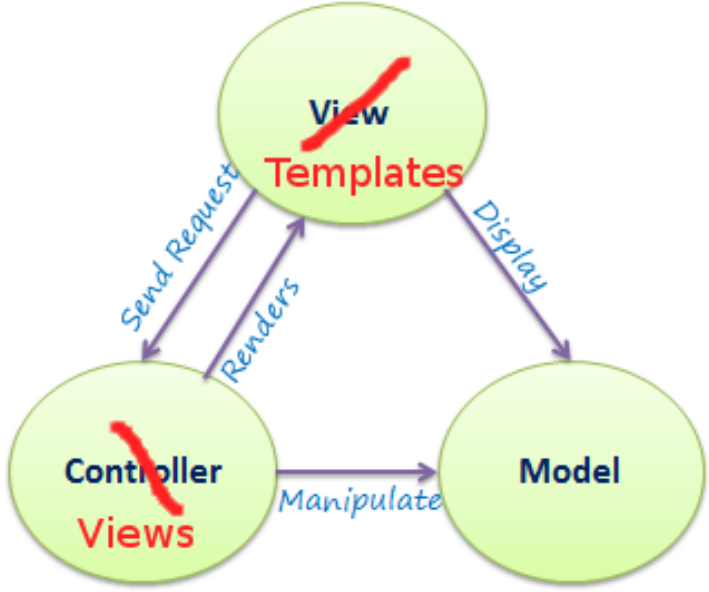
Wait!

Understanding oTree

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

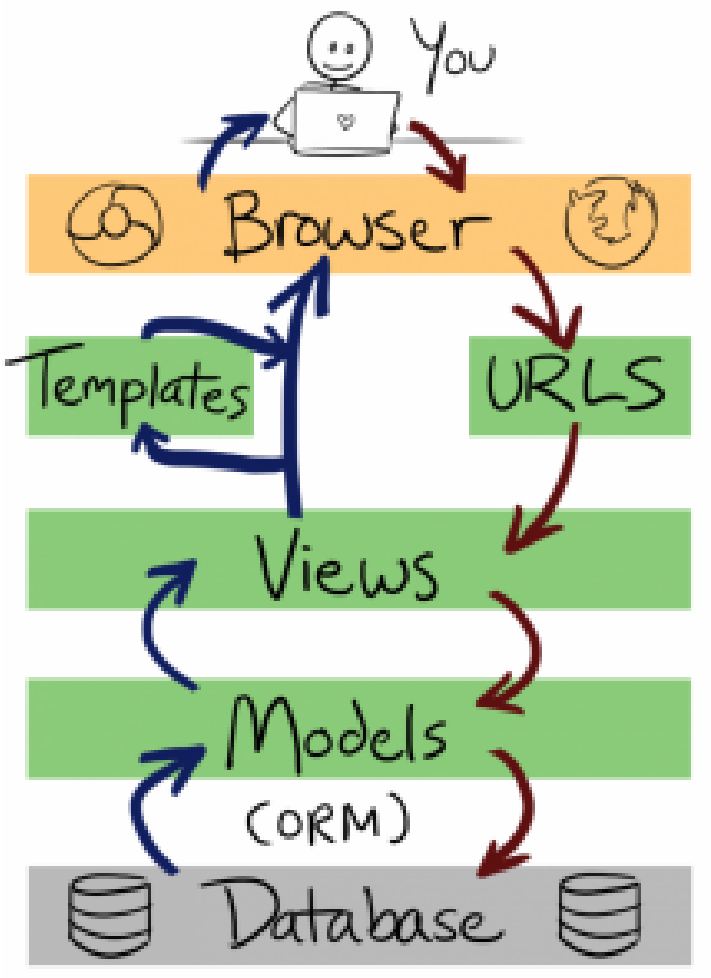
oTree is a **Model-View-Template** (MVT) Framework

- 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.

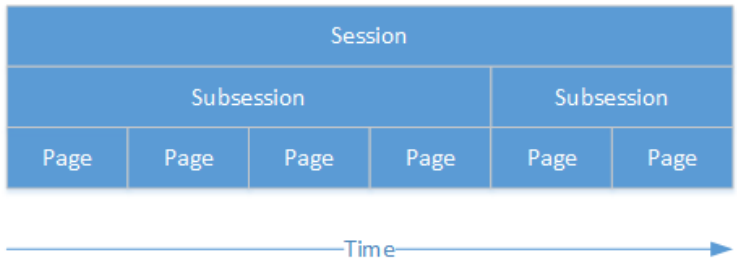
oTree Models - Subsession class

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:*



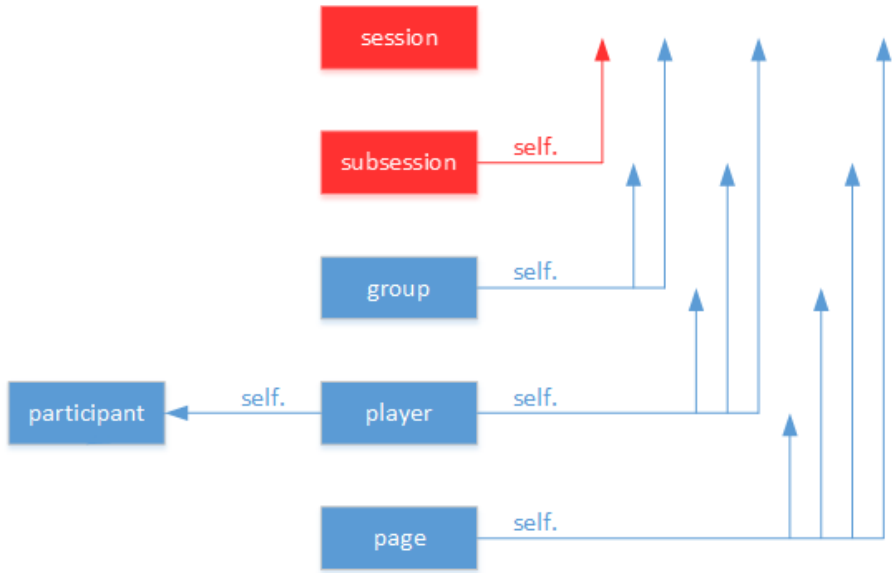
If a game is repeated for multiple rounds, **each round is a subsession**.

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:

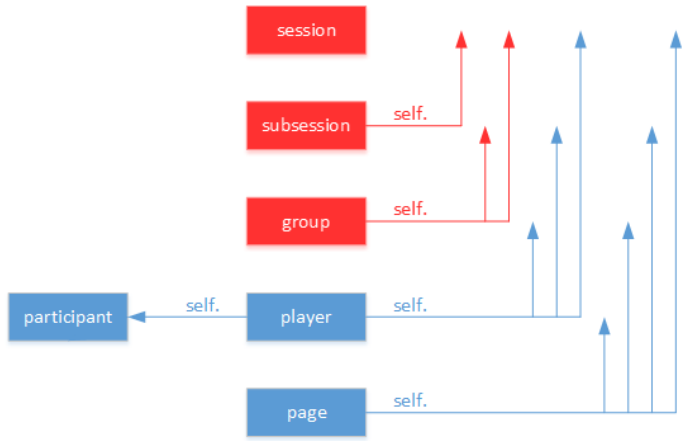
> *you could have a subsession with 30 players, divided into 15 groups of 2 players each. (Note: groups can be shuffled between subsessions.)*

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
```

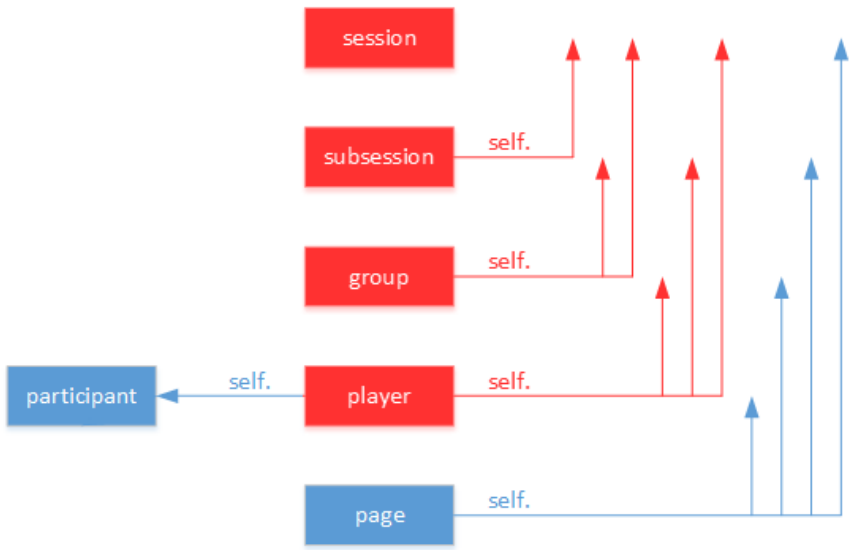
- **`get_players()`**: Returns a list of all the players in the subsession

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:

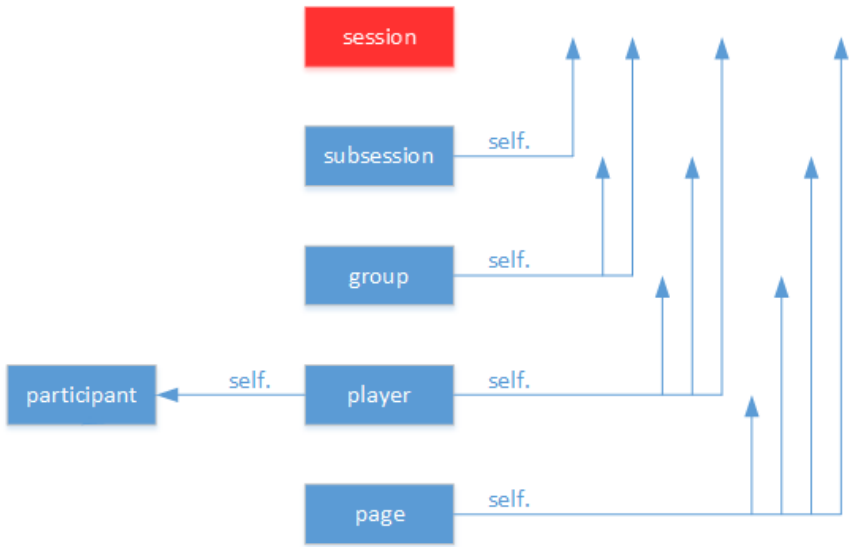
```
class Player(BasePlayer):
    def role(self):
        if self.id_in_group == 1:
            return 'buyer'
        if self.id_in_group == 2:
            return 'seller'
```

- 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

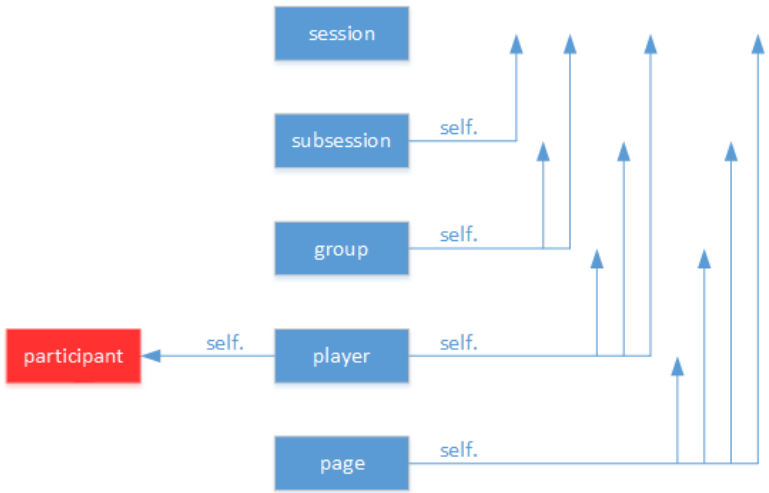
- `num_participants`: The number of participants in the session.
- `config`: Dict-like from `settings.SESSION_CONFIGS`
- `vars`: Dict-like to store global variables that are the same for all participants in the session



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.

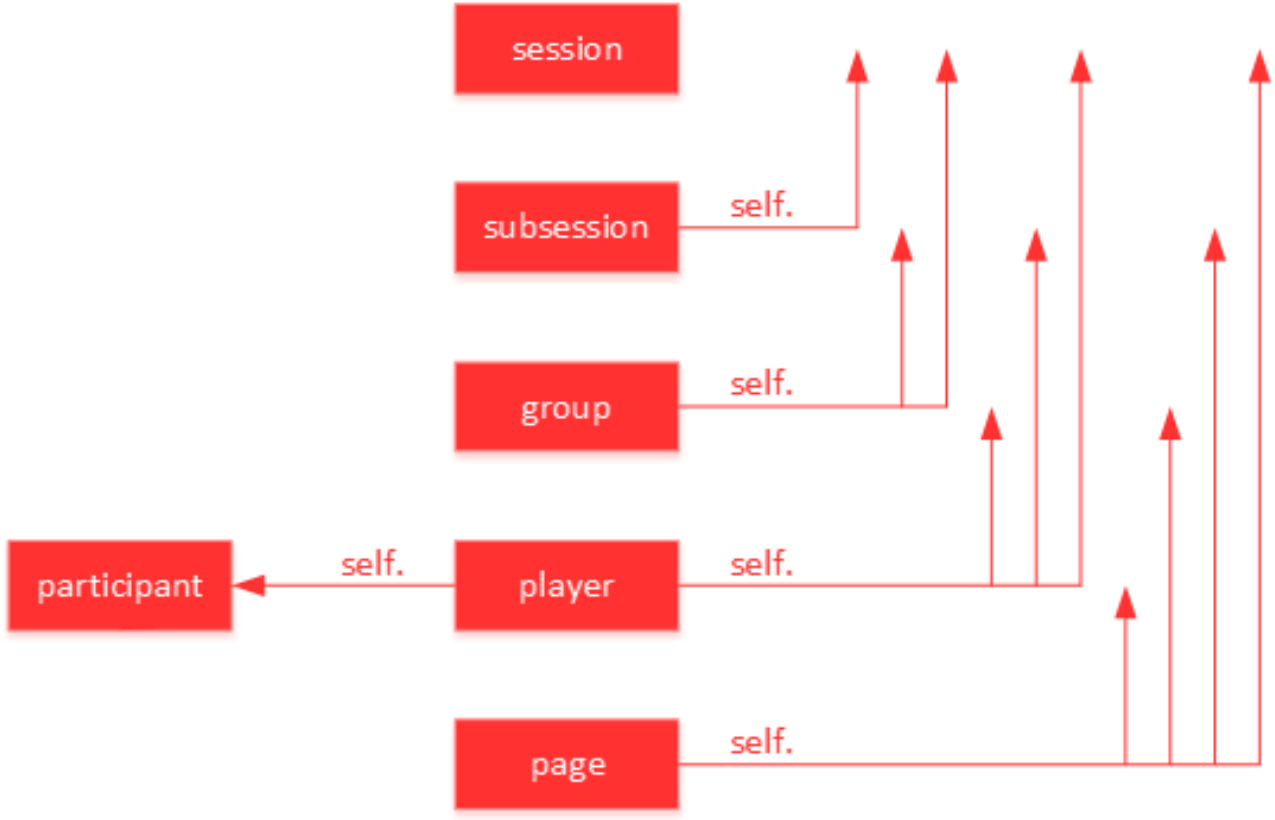
```
class Subsession(BaseSubsession):

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

oTree Views

Page class.

Each page that your players see is defined by a `Page` class in `views.py`



oTree Views

Page class.

- Here is where the logic of your experiment lives.
- In oTree context "views" is basically a synonym for "pages".
- Your **`views.py`** must have a **`page_sequence`** variable that gives the order of the pages. For example:

```
page_sequence = [Start, Offer, Accept, Results]
```

oTree Views

`Page.is_displayed()` method

- You can define this function to return `True` if the page should be shown, and `False` if the page should be skipped. If omitted, the page will be shown.
  For example, to only show the page to P2 in each group:

```
class Page1(Page):
    def is_displayed(self):
        return self.player.id_in_group == 2
```

- Or only show the page in round 1:

```
class Page1(Page):
    def is_displayed(self):
        return self.round_number == 1
```

oTree Views

Page.vars_for_template() method

- You can use this to return a dictionary of variable names and their values, which is passed to the template. Example:

```
class Page1(Page):
    def vars_for_template(self):
        return {'a': 1 + 1,
                'b': self.player.foo * 10}
```

- Then in the template you can access a and b like this:

```
Variables {{ a }} and {{ b }} ...
```

oTree Views

`Page.vars_for_template()` method

- oTree automatically passes the following objects to the template: **`player`**, **`group`**, **`subsession`**, **`participant`**, **`session`**, and **`Constants`**.
- You can access them in the template like this:

```
{{ Constants.blah }} or {{ player.blah }}
```

- **Warning** vars_for_template executes every time the page is reloaded.

oTree Views

`Page.before_next_page()` method

- Here you define any code that should be executed after form validation, before the player proceeds to the next page.
- If the page is skipped with **`is_displayed`**, then **`before_next_page`** will be skipped as well.
- Example:
- oTree automatically passes the following objects to the template: **`player`**, **`group`**, **`subsession`**, **`participant`**, **`session`**, and **`Constants`**.
- You can access them in the template like this:

```
class Page1(Page):
    def before_next_page(self):
        self.group.set_payoff()
```

oTree Views

Page.template_name attribute

- Each Page should have a file in **templates/** with the same name. For example, if your app has this page **in my_app/views.py**:

```
class Page1(Page):
    ...
```

- Then you should create a file **my_app/templates/my_app/Page1.html**, (note that my_app is repeated).
- If the template needs to have a different name from your view class, set `template_name`

```
class Page1(Page):
    template_name = 'app_name/MyView.html'
```

oTree Templates

- Your app's `templates/` directory will contain the templates for the HTML that gets displayed to the player.
- oTree uses Django's template system.
- Django's templates it's designed to feel comfortable to those used to working with HTML.

oTree Templates

HTML vs. Building Blocks

- Right click on any webpage and select “source code”
- Any HTML starts with something like this

```
<!DOCTYPE html>
    <html lang="en">
        <head>
            <!-- and so on... -->
```

oTree Templates

HTML vs. Building Blocks

- Instead of writing the full HTML of your page, You define 2 blocks:

```
{% block title %} Title goes here {% endblock %}

{% block content %}
    Body HTML goes here.
    {% formfield player.contribution
        label="What is your contribution?" %}
    {% next_button %}
{% endblock %}
```

oTree Templates

Disclaimer

HighCharts, CSS and Javascript are not part of this course.

oTree Templates

Static content (images, videos, CSS, JavaScript)

- To include static files (.png, .jpg, .mp4, .css, .js, etc.) in your pages, make sure your template has at the top.

```
{% load staticfiles %}
```

- Then create a **static/** folder in your app (next to **templates/**).
- Like **templates/** it should also have a subfolder with your app's name, e.g. **static/my_app**.
- Put your files in that subfolder. You can then reference them in a template like this:

```
<img src="{% static 'my_app/my_image.png' %}"/>
```

oTree Templates

Static content (images, videos, CSS, JavaScript)

- If the file is used in multiple apps, you can put it in `_static/global/`, then do:

```
<img src="{% static 'global/my_image.png' %}"/>
```

oTree Templates

Static content (images, videos, CSS, JavaScript)

- If the image/video path is variable (like showing a different image each round), you can construct it in `views.py` and pass it to the template, e.g.:

```
class MyPage(Page):
    def vars_for_template(self):
        path = 'my_app/{}.png'.format(self.round_number)
        return {'image_path': image_path},
```

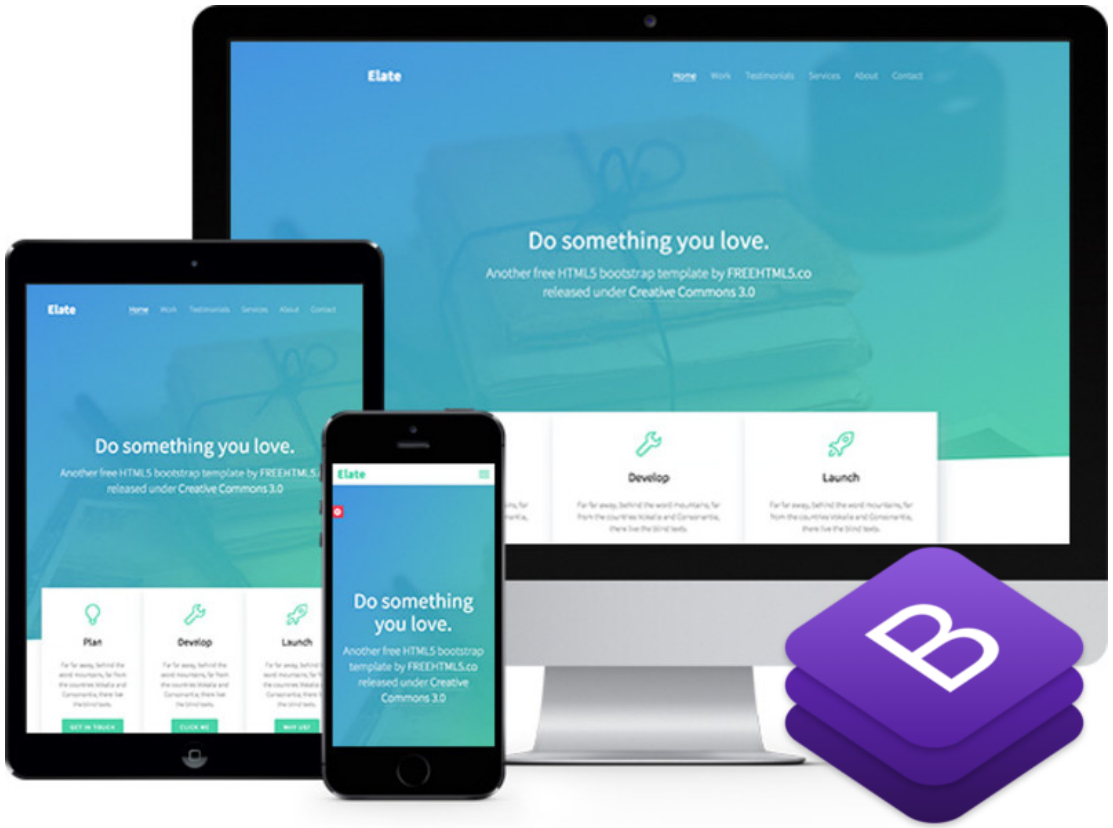
- And then in the template `MyPage.html`:

```
<img src="{% static image_path %}"/>
```

oTree Templates

Mobile devices

- oTree's HTML interface is based on Bootstrap, which works on any modern browser (Chrome/Internet Explorer/Firefox/Safari).
- Bootstrap also tries to shows a "mobile friendly" version when viewed on a smartphone or tablet.



oTree Templates

Template filters

- In addition to the filters available with Django's template language, oTree has the `|c` filter, which is equivalent to the `c()` function. For example, to displays as 20 whatever-is-your-currency.

```
{{ 20|c }}
```

- Also, the `|abs` filter lets you take the absolute value. So, doing `{{ -20|abs }}` would output **20**.

oTree Forms

- Each page in oTree can contain a form.
- To create a form,
  1. Go to `models.py` and define fields on your `Player` or `Group`.
  2. Then, in your *Page* class, you can choose which of these fields to include in the form You do this by:
     - 2.1 setting **`form_model = models.Player`**, or **`form_model = models.Group`**,
     - 2.2 and then set **`form_fields`** to the list of fields you want in your form.
- When the user submits the form, the submitted data is automatically saved to the field in your model.

oTree Forms

For example, here is a models.py:

```
class Group(BaseGroup):
    f1 = models.BooleanField()
    f2 = models.BooleanField()

class Player(BasePlayer):
    f1 = models.BooleanField()
    f2 = models.BooleanField()
```

And a corresponding views.py that defines the form on each page:

```
class Page1(Page):
    form_model = models.Player
    form_fields = ['f1', 'f2'] # player.f1, player.f2

class Page2(Page):
    form_model = models.Group
    form_fields = ['f1', 'f2'] # group.f1, group.f2
```

oTree Forms

Forms in templates

- You should include form fields by using a `{% formfield %}` element:

```
{% formfield player.contribution
    label="How much do you want to contribute?" %}
```

oTree Forms

Forms in templates

- An alternative to using label is to define **verbose_name** on the model field:

```
class Player(BasePlayer):
  contribution = models.CurrencyField(
   verbose_name="How much do you want to contribute?")
```

- Then you can just put this in your template:

```
{% formfield player.contribution %}
```

oTree Forms

Forms in templates

- If you have multiple form fields, you can insert them all at once:

```
{% for field in form %}
    {% formfield field %}
{% endfor %}
```

- also you can simple write

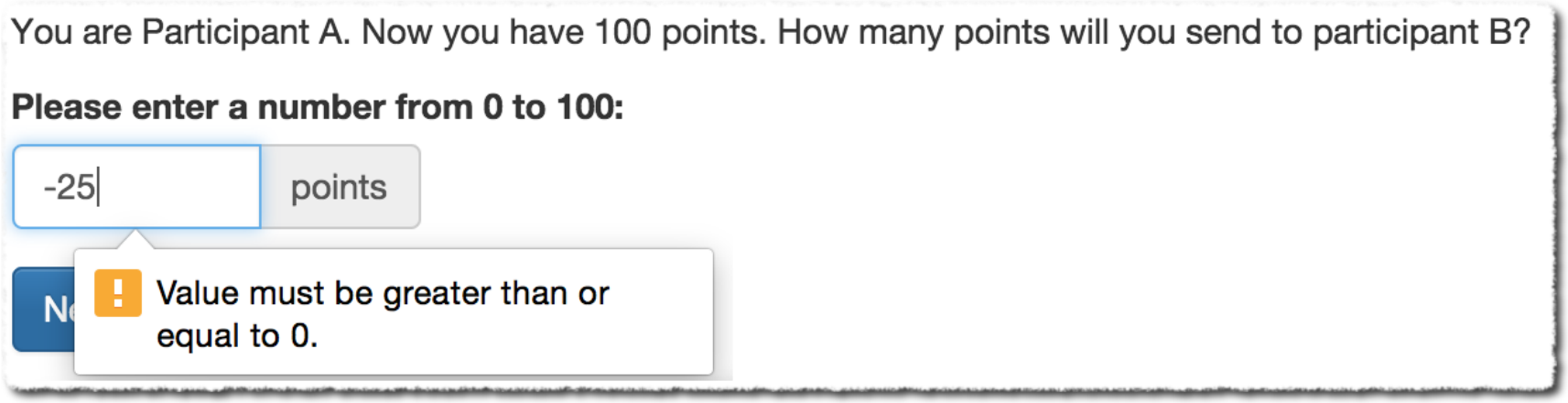
```
{{ form }}  # the Django way
```

oTree Forms

Simple form field validation

- The player must submit a valid form before they go to the next page.
- If the form they submit is invalid (e.g. missing or incorrect values), it will be re-displayed to them along with the list of errors they need to correct.

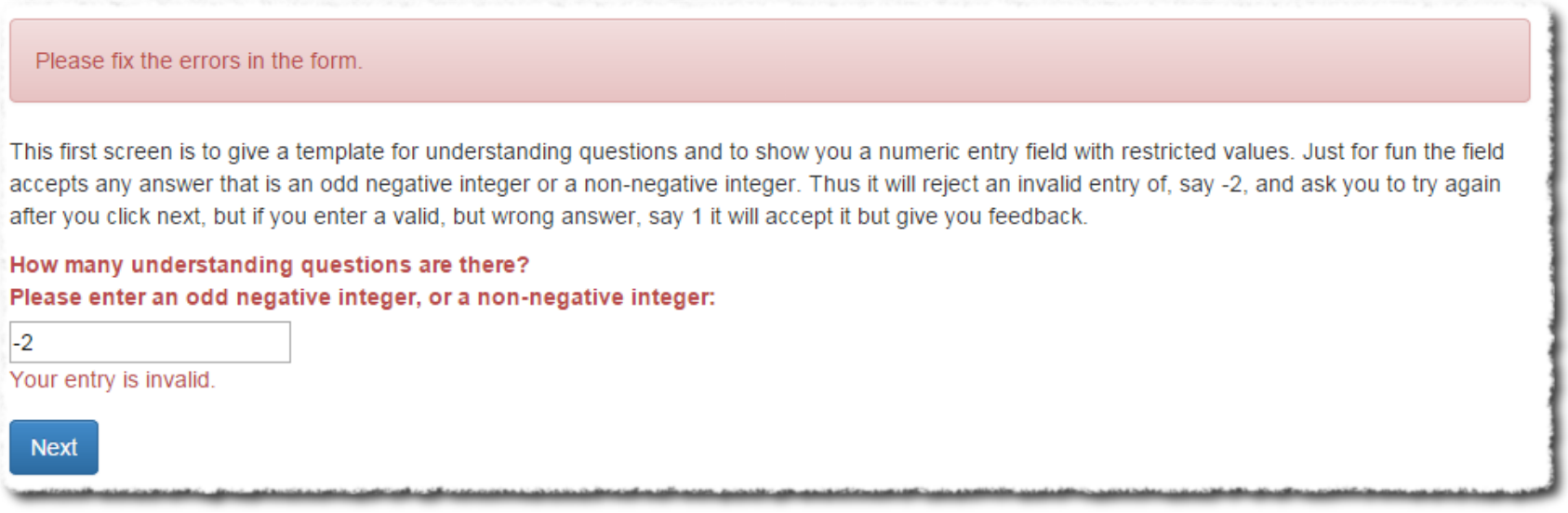
Error 1



oTree Forms

Simple form field validation

Error 2



- oTree automatically validates all input submitted by the user.
- For example, if you have a form containing a **`IntegerField`**, oTree will not let the user submit values that are not positive integers, like **`-1`** , **`1.5`**, or **`hello`**.
- oTree also checks **`min`**, **`max`** and **`choices`** attributes in models.

oTree Forms

Simple form field validation - Choices

- If you want a field to be a dropdown menu with a list of choices, set **`choices=`**:

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

- To use radio buttons instead of a dropdown menu, you should set the widget to **`RadioSelect`** or **`RadioSelectHorizontal`**:

```
# in models.py
level = models.IntegerField(
    choices=[1, 2, 3],
    widget=widgets.RadioSelect)
```

oTree Forms

Simple form field validation - Choices

- You can also set display names for each choice by making a list of **`[value, display]`** pairs:

```
# in models.py
level = models.IntegerField(
choices=[
    [1, 'Low'],
    [2, 'Medium'],
    [3, 'High']])
```

- If you do this, users will just see a menu with "Low", "Medium", "High", but their responses will be recorded as 1, 2, or 3.
- After the field has been set, you can access the human-readable name using get_FOO_display , like this: **`self.get_level_display()`**

oTree Forms

Simple form field validation - Optional fields

- If a field is optional, you can use **`blank=True`**, Then the HTML field will not have the required attribute.

```
# in models.py
offer = models.IntegerField(blank=True)
```

oTree Forms

Dynamic form field validation

- The min, max, and choices described above are only for fixed (constant) values.
- If you want them to be determined dynamically (e.g. different from player to player), then you can instead define one of the next methods in your Page class in views.py.

oTree Forms

Dynamic form field validation - `{field_name}_choices()`

- Like setting **`choices=`** in models.py, this will set the choices for the form field (e.g. the dropdown menu or radio buttons).

Example:

```
class MyPage(Page):

    form_model = models.Player
    form_fields = ['offer']

    def offer_choices(self):
        return currency_range(
            0, self.player.endowment, 1)
```

oTree Forms

Dynamic form field validation - `{field_name}_max()`

The dynamic alternative to setting **`max=`** in `models.py`. For example:

```
class MyPage(Page):

    form_model = models.Player
    form_fields = ['offer']

    def offer_max(self):
        return self.player.endowment
```

oTree Forms

Dynamic form field validation - `{field_name}_min()`

- The dynamic alternative to setting **`min=`** in `models.py`.

```
class MyPage(Page):

    form_model = models.Player
    form_fields = ['offer']

    def offer_max(self):
        return self.player.endowment * .1
```

oTree Forms

Dynamic form field validation - `{field_name}_error_message()`

- This is the most flexible method for validating a field.
- For example, let's say your form has an integer field called **`odd_negative`**, which must be odd and negative. You would enforce this as follows:

```
class MyPage(Page):

    form_model = models.Player
    form_fields = ['odd_negative']

    def odd_negative_error_message(self, value):
        is_odd = (value % 2 == 1)
        is_negative = (value < 0)
        if not (is_odd and is_negative):
            return 'Must be odd and negative'
```

oTree Forms

Dynamic form field validation - Validating multiple fields together

- Let's say you have 3 integer fields in your form whose names are **int1**, **int2**, and **int3**, and the values submitted must **sum to 100**. You can enforce this with the error_message method:

```
class MyPage(Page):

    form_model = models.Player
    form_fields = ['int1', 'int2', 'int3']

    def error_message(self, values):
        total = (
            values["int1"] + values["int2"] +
            values["int3"])
        if total != 100:
            return 'The numbers must add up to 100'
```

oTree Forms

Determining form fields dynamically

- If you need the list of form fields to be dynamic, instead of **`form_fields`** you can define a method **`get_form_fields(self)`** that returns the list. For example:

```
class MyPage(Page):
    form_model = models.Player
    def get_form_fields(self):
        if self.player.num_bids == 3:
            return ['bid_1', 'bid_2', 'bid_3']
        else:
            return ['bid_1', 'bid_2']
```

- **WARNING:** if you do this, you must make sure your template also contains conditional logic so that the right formfield elements are included.

oTree Forms

Determining form fields dynamically

- You can do this by looping through each field in the form. oTree passes a variable form to each template, which you can loop through like this:

```
{% for field in form %}
    {% formfield field %}
{% endfor %}
```

- If you use this technique, you should consider setting verbose_name on your model fields (see Forms in templates).

oTree Forms

Widgets

- The full list of form input widgets offered by Django is here. https://docs.djangoproject.com/en/1.7/ref/forms/widgets/#built-in-widgets
- oTree additionally offers:
  - `RadioSelectHorizontal`: same as RadioSelect but with a horizontal layout
  - `SliderInput`
    - To specify the step size, do: `SliderInput(attrs={'step': '0.01'})`
    - To disable the current value from being displayed, do: `SliderInput(show_value=False)`

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