
The logo for oTree, featuring the word "oTree" in a white, sans-serif font on a dark blue background.

A modern open platform
for social science experiments



oTree Documentation

Release

oTree Team

October 23, 2015

1	Homepage	3
2	About	5
3	Repositories	7
4	Contact	9
5	Contributors	11
6	Mailing list	13
7	Contents:	15
7.1	Tutorial	15
7.2	Download & Setup	30
7.3	Conceptual overview	32
7.4	Applications	33
7.5	Models	33
7.6	Views	35
7.7	Templates	37
7.8	Forms	40
7.9	<i>Self</i> and object model	43
7.10	Groups and multiplayer games	46
7.11	Money and Points	49
7.12	Treatments	51
7.13	Rounds	52
7.14	Localization	53
7.15	Manual testing	55
7.16	Automated testing (bots)	56
7.17	Admin	57
7.18	Server deployment	61
7.19	Troubleshooting	64
7.20	Tips and tricks	65
7.21	Amazon Mechanical Turk	65
7.22	oTree programming For Django Devs	67
7.23	oTree glossary for z-Tree programmers	68
7.24	Indices and tables	73

<http://demo.otree.org/>


Homepage

<http://www.otree.org/>

About

oTree is a Django-based framework for implementing multiplayer decision strategy games. Many of the details of writing a web application are abstracted away, meaning that the code is focused on the logic of the game. oTree programming is accessible to programmers without advanced experience in web app development.

Repositories

-  Games
-  Core Libraries
-  Launcher (*OS independent*)
-  Documentation

Contact

chris@otree.org (you can also add me on Skype by searching this email address; please mention oTree in your contact request)

Please contact me if any part of oTree does not work for you (or is unclear).

Contributors

- Gregor Muellegger (<http://gremu.net/>,  <https://github.com/gregmuellegger>)
- Juan B. Cabral (<http://jbcabral.org>,  <https://github.com/leliel12>)
- Alexander Schepanovski ( <https://github.com/Suor>)
- Alexander Sandukovskiy
- Som Datye

Mailing list

Sign up to be notified about updates to oTree [here](#)

Contents:

7.1 Tutorial

This tutorial will cover the creation of 3 games:

- Public goods game
- Trust game
- Matching pennies

Before proceeding through this tutorial, install oTree according to the instructions in [Download & Setup](#).

7.1.1 Part 1: Public goods game

We will now create a simple [public goods game](#). The completed app is [here](#).

Create the app

If you are running the oTree launcher, click the “terminal” button which will open your command window. Otherwise, open the oTree folder you downloaded, the one that contains `requirements_base.txt`.

In this directory, create the public goods app with this shell command:

```
$ otree startapp my_public_goods
```

Then go to the folder `my_public_goods` that was created.

Define `models.py`

Let's define our data model in `models.py`.

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, let's 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 the docs for more info).
- Each contribution is multiplied by 1.8. So let's define `efficiency_factor` and set it to 1.8:

Now we have:

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

    endowment = c(100)
    efficiency_factor = 1.8
```

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

What data points are we interested in recording about each player? The main thing is how much they contributed. So, we define a field `contribution`:

```
class Player(otree.models.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(otree.models.BaseGroup):

    # ...

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

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(otree.models.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 / Constants.players_per_group
        for p in self.get_players():
            p.payoff = Constants.endowment - p.contribution + self.individual_share
```

Define the template

This game will have 2 pages.

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

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

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/Base.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 %}

```

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

```

{% extends "global/Base.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>

{% endblock %}

```

Define views.py

Now we define our views, which decide 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 (Contribute and Results).

First let's define `Contribute`. We need to define `form_model` and `form_fields` to specify that this page contains a form letting you set `Player.contribution`:

```

class Contribute(Page):

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

```

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

```

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.

When all players have completed the `Contribute` page, the players' payoffs can be calculated. You can trigger this calculation inside 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.

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  
]
```

Define the session config in settings.py

Now we go to `settings.py` and add an entry to `SESSION_CONFIGS`.

Note: Prior to oTree-core 0.3.11, “session config” was known as “session type”. After you upgrade, you can rename `SESSION_TYPES` to `SESSION_CONFIGS`, and `SESSION_TYPE_DEFAULTS` to `SESSION_CONFIG_DEFAULTS`.

In lab experiments, it's typical for users to fill out an exit survey, and then see how much money they made. So let's do this by adding the existing “exit survey” and “payment info” apps to `app_sequence`.

```
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'],  
    },  
    # ...
```

However, we must also remember to add a `{% next_button %}` element to the `Results.html` (somewhere inside the `{% content %}` block, so the user can click a button taking them to the next app in the sequence.

Reset the database and run

Before you run the server, you need to reset the database. In the launcher, click the button “reset database”. Or, on the command line, run `otree resetdb`. (You need to run `resetdb` every time you create a new app, or when you add/change/remove a field in `models.py`. This is because you have new fields in `models.py`, and the SQL database needs to be re-generated to create these tables and columns.)

Then, run the server and open your browser to <http://127.0.0.1:8000> to play the game.

7.1.2 Part 2: Trust game

Now let's create a Trust game, and learn some more features of oTree.

This is a trust game with 2 players. To start, Player 1 receives 10 points; Player 2 receives nothing. Player 1 can send some or all of his points to Player 2. Before P2 receives these points they will be tripled. Once P2 receives the tripled points he can decide to send some or all of his points to P1.

The completed app is [here](#).

Create the app

```
$ otree startapp my_trust
```

Define models.py

First we define our app’s constants. The endowment is 10 points and the donation gets tripled.

```
class Constants:
    name_in_url = 'my_trust'
    players_per_group = 2
    num_rounds = 1

    endowment = c(10)
    multiplication_factor = 3
```

Then we think about how to define fields on the data model. There are 2 critical data points to capture: the “sent” amount from P1, and the “sent back” amount from P2.

Your first instinct may be to define the fields on the Player like this:

```
class Player(otree.models.BasePlayer):

    # <built-in>
    ...
    # </built-in>

    sent_amount = models.CurrencyField()
    sent_back_amount = models.CurrencyField()
```

The problem with this model is that `sent_amount` only applies to P1, and `sent_back_amount` only applies to P2. It does not make sense that P1 should have a field called `sent_back_amount`. How can we make our data model more accurate?

We can do it by defining those fields at the Group level. This makes sense because each group has exactly 1 `sent_amount` and exactly 1 `sent_back_amount`:

```
class Group(otree.models.BaseGroup):

    # <built-in>
    ...
    # </built-in>

    sent_amount = models.CurrencyField()
    sent_back_amount = models.CurrencyField()
```

Even though it may not seem that important at this point, modeling our data correctly will make the rest of our work easier.

Let’s let P1 choose from a dropdown menu how much to donate, rather than entering free text. To do this, we use the `choices=` argument, as well as the `currency_range` function:

```
sent_amount = models.CurrencyField(
    choices=currency_range(0, Constants.endowment, c(1)),
)
```

We’d also like P2 to use a dropdown menu to choose how much to send back, but we can’t specify a fixed list of `choices`, because P2’s available choices depend on how much P1 donated. I’ll show a bit later how we can make this list dynamic.

Also, let’s define the payoff function on the group:

```
def set_payoffs(self):
    p1 = self.get_player_by_id(1)
    p2 = self.get_player_by_id(2)
    p1.payoff = Constants.endowment - self.sent_amount + self.sent_back_amount
    p2.payoff = self.sent_amount * Constants.multiplication_factor - self.sent_back_amount
```

Define the templates and views

We need 3 pages:

- P1's "Send" page
- P2's "Send back" page
- "Results" page that both users see.

It would also be good if game instructions appeared on each page so that players are clear how the game works. We can define a file `Instructions.html` that gets included on each page.

Instructions.html

This template uses Django's template inheritance with the `{% extends %}` command. The file it inherits from is located at `_templates/global/Instructions.html`.

For basic apps you don't need to know the details of how template inheritance works.

```
{% extends "global/Instructions.html" %}

{% block instructions %}
<p>
    This is a trust game with 2 players.
</p>
<p>
    To start, participant A receives {{ Constants.endowment }};
    participant B receives nothing.
    Participant A can send some or all of his {{ Constants.endowment }}
    to participant B. Before B receives these points they will be tripled.
    Once B receives the tripled points he can decide to send some or all
    of his points to A.
</p>
{% endblock %}
```

Send.html

This page looks like the templates we have seen so far. Note the use of `{% include %}` to automatically insert another template.

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

{% block title %}
    Trust Game: Your Choice
{% endblock %}

{% block content %}

    {% include 'my_trust/Instructions.html' %}

    <p>
        You are Participant A. Now you have {{ Constants.endowment }}.
    </p>
```



```

</p>

{% formfield group.sent_amount with label="How much do you want to send to participant B?" %}

{% next_button %}

{% endblock %}

```

We also define the view in views.py:

```

class Send(Page):

    form_model = models.Group
    form_fields = ['sent_amount']

    def is_displayed(self):
        return self.player.id_in_group == 1

```

The `{% formfield %}` in the template must match the `form_model` and `form_fields` in the view.

Also, we use `is_displayed` to only show this to P1; P2 skips the page.

SendBack.html

This is the page that P2 sees to send money back. Here is the template:

```

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

{% block title %}
    Trust Game: Your Choice
{% endblock %}

{% block content %}

    {% include 'my_trust/Instructions.html' %}

    <p>
        You are Participant B. Participant A sent you {{group.sent_amount}}
        and you received {{tripled_amount}}.
    </p>

    {% formfield group.sent_back_amount with label="How much do you want to send back?" %}

    {% next_button %}

{% endblock %}

```

Here is the code from views.py. Notes:

- We use `vars_for_template` to pass the variable `tripled_amount` to the template. Django does not let you do calculations directly in a template, so this number needs to be calculated in Python code and passed to the template.
- We define a method `sent_back_amount_choices` to populate the dropdown menu dynamically. This is the feature called `{field_name}_choices`, which is explained in the reference documentation.

```

class SendBack(Page):

    form_model = models.Group
    form_fields = ['sent_back_amount']

    def is_displayed(self):

```

```
    return self.player.id_in_group == 2

    def vars_for_template(self):
        return {
            'tripled_amount': self.group.sent_amount * Constants.multiplication_factor
        }

    def sent_back_amount_choices(self):
        return currency_range(
            c(0),
            self.group.sent_amount * Constants.multiplication_factor,
            c(1)
        )
```

Results

The results page needs to look slightly different for P1 vs. P2. So, we use the `{% if %}` statement (part of Django's template language) to condition on the current player's `id_in_group`.

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

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

{% block content %}

{% if player.id_in_group == 1 %}
    <p>
        You sent Participant B {{ group.sent_amount }}.
        Participant B returned {{group.sent_back_amount}}.
    </p>
{% else %}
    <p>
        Participant A sent you {{ group.sent_amount }}.
        You returned {{group.sent_back_amount}}.
    </p>
{% endif %}

    <p>
        Therefore, your total payoff is {{player.payoff}}.
    </p>

    {% include 'my_trust/Instructions.html' %}

{% endblock %}
```

Here is the Python code for this page in `views.py`:

```
class Results(Page):

    def vars_for_template(self):
        return {
            'tripled_amount': self.group.sent_amount * Constants.multiplication_factor
        }
```

Wait pages and page sequence

This game has 2 wait pages:

- P2 needs to wait while P1 decides how much to send
- P1 needs to wait while P2 decides how much to send back

After the second wait page, we should calculate the payoffs. So, we use `after_all_players_arrive`.

So, we define these pages:

```
class WaitForP1(WaitPage):
    pass

class ResultsWaitPage(WaitPage):

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

Then we define the page sequence:

```
page_sequence = [
    Send,
    WaitForP1,
    SendBack,
    ResultsWaitPage,
    Results,
]
```

Add an entry to `SESSION_CONFIGS` in `settings.py`

```
{
    'name': 'my_trust',
    'display_name': "My Trust Game (simple version from tutorial)",
    'num_demo_participants': 2,
    'app_sequence': ['my_trust'],
},
```

Reset the database and run

If you are using the launcher, click the “Reset DB” and “Run server” buttons.

If you are on the command line, enter:

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

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

7.1.3 Part 3: Matching pennies

We will now create a “Matching pennies” game with the following features:

- 4 rounds
- The roles of the players will be reversed halfway through
- In each round, a “history box” will display the results of previous rounds
- A random round will be chosen for payment

The completed app is [here](#).

Update otree-core

For this game to work, you need otree-core \geq 0.3.3, which contains important bugfixes.

If you installed oTree prior to 2015-05-25, you need to update your `requirements_base.txt` to the latest version [here](#) and then do:

```
$ pip install -r requirements_base.txt
```

Create the app

```
$ otree startapp my_matching_pennies
```

Define models.py

We define our constants as we have previously. Matching pennies is a 2-person game, and the payoff for winning a paying round is 100 points.

```
class Constants:
    name_in_url = 'my_matching_pennies'
    players_per_group = 2
    num_rounds = 4
    stakes = c(100)
```

Now let's define our `Player` class:

- In each round, each player decides “Heads” or “Tails”, so we define a field `penny_side`, which will be displayed as a radio button.
- We also have a boolean field `is_winner` that records if this player won this round.
- We define the `role` method to define which player is the “Matcher” and which is the “Mismatcher”.

So we have:

```
class Player(otree.models.BasePlayer):

    # <built-in>
    # ...
    # </built-in>

    penny_side = models.CharField(
        choices=['Heads', 'Tails'],
        widget=widgets.RadioSelect()
    )

    is_winner = models.BooleanField()

    def role(self):
        if self.id_in_group == 1:
            return 'Mismatcher'
        if self.id_in_group == 2:
            return 'Matcher'
```

Now let's define the code to randomly choose a round for payment. Let's define the code in `Subsession.before_session_starts`, which is the place to put global code that initializes the state of the game, before gameplay starts.

The value of the chosen round is “global” rather than different for each participant, so the logical place to store it is as a “global” variable in `self.session.vars`.

So, we start by writing something like this, which chooses a random integer between 1 and 4, and then assigns it into `session.vars`:

```
class Subsession(otree.models.BaseSubsession):

    def before_session_starts(self):
        paying_round = random.randint(1, Constants.num_rounds)
        self.session.vars['paying_round'] = paying_round
```

There is a slight mistake, however. Because there are 4 rounds (i.e. subsessions), this code will get executed 4 times, each time overwriting the previous value of `session.vars['paying_round']`, which is superfluous. We can fix this with an `if` statement that makes it only run once (on the first round):

```
class Subsession(otree.models.BaseSubsession):

    def before_session_starts(self):
        if self.round_number == 1:
            paying_round = random.randint(1, Constants.num_rounds)
            self.session.vars['paying_round'] = paying_round
```

Now, let's also define the code to swap roles halfway through. This kind of group-shuffling code should also go in `before_session_starts`. We put it after our existing code.

In oTree, groups are assigned automatically in the first round, and in each round, the groups are kept the same as the previous round, unless you shuffle them in `before_session_starts`. So, at the beginning of round 3, we should do the shuffle. (So that the groups will be in opposite order during rounds 3 and 4.)

We use `group.get_players()` to get the ordered list of players in each group, and then reverse it (e.g. the list `[P1, P2]` becomes `[P2, P1]`). Then we use `group.set_players()` to set this as the new group order:

```
class Subsession(otree.models.BaseSubsession):

    def before_session_starts(self):
        if self.round_number == 1:
            ...
        if self.round_number == 3:
            # reverse the roles
            for group in self.get_groups():
                players = group.get_players()
                players.reverse()
                group.set_players(players)
```

Now we define our `Group` class. We define the payoff method. We use `get_player_by_role` to fetch each of the 2 players in the group. We could also use `get_player_by_id`, but I find it easier to identify the players by their roles as `matcher/mismatcher`. Then, depending on whether the penny sides match, we either make P1 or P2 the winner.

So, we start with this:

```
class Group(otree.models.BaseGroup):

    # <built-in>
    ...
    # </built-in>

    def set_payoffs(self):
        matcher = self.get_player_by_role('Matcher')
        mismatcher = self.get_player_by_role('Mismatcher')

        if matcher.penny_side == mismatcher.penny_side:
            matcher.is_winner = True
            mismatcher.is_winner = False
        else:
            matcher.is_winner = False
            mismatcher.is_winner = True
```

We should expand this code by setting the actual `payoff` field. However, the player should only receive a payoff if the current round is the randomly chosen paying round. Otherwise, the payoff should be 0 points. So, we check the current round number and compare it against the value we previously stored in `session.vars`. We loop through both players (`[P1, P2]`, or `[mismatcher, matcher]`) and do the same check for both of them.

```
class Group(otree.models.BaseGroup):

    # <built-in>
    subsession = models.ForeignKey(Subsession)
    # </built-in>

    def set_payoffs(self):
        matcher = self.get_player_by_role('Matcher')
        mismatcher = self.get_player_by_role('Mismatcher')

        if matcher.penny_side == mismatcher.penny_side:
            matcher.is_winner = True
            mismatcher.is_winner = False
        else:
            matcher.is_winner = False
            mismatcher.is_winner = True
        for player in [mismatcher, matcher]:
            if (self.subsession.round_number ==
                self.session.vars['paying_round'] and player.is_winner):
                player.payoff = Constants.stakes
            else:
                player.payoff = c(0)
```

Define the templates and views

This game has 2 main pages:

- A Choice page that gets repeated for each round. The user is asked to choose heads/tails, and they are

also shown a “history box” showing the results of previous rounds. - A ResultsSummary page that only gets displayed once at the end, and tells the user their final payoff.

Choice

In `views.py`, we define the Choice page. This page should contain a form field that sets `player.penny_side`, so we set `form_model` and `form_fields`.

Also, on this page we would like to display a “history box” table that shows the result of all previous rounds. So, we can use `player.in_previous_rounds()`, which returns a list referring to the same participant in rounds 1, 2, 3, etc. (For more on the distinction between “player” and “participant”, see the reference docs.)

```
class Choice(Page):

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

    def vars_for_template(self):
        return {
            'player_in_previous_rounds': self.player.in_previous_rounds(),
        }
```

We then create a template `Choice.html` below. This is similar to the templates we have previously created, but note the `{% for %}` loop that creates all rows in the history table. `{% for %}` is part of the Django template language.

```

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

{% block title %}
    Round {{ subsession.round_number }} of {{ Constants.num_rounds }}
{% endblock %}

{% block content %}

    <h4>Instructions</h4>
    <p>
        This is a matching pennies game.
        Player 1 is the 'Mismatcher' and wins if the choices mismatch;
        Player 2 is the 'Matcher' and wins if they match.

    </p>

    <p>
        At the end, a random round will be chosen for payment.
    </p>

    <h4>Round history</h4>
    <table class="table">
        <tr>
            <th>Round</th>
            <th>Player and outcome</th>
        </tr>
        {% for p in player_in_previous_rounds %}
            <tr>
                <td>{{ p.subsession.round_number }}</td>
                <td>
                    You were the {{ p.role }} and {% if p.is_winner %}
                    won {% else %} lost {% endif %}
                </td>
            </tr>
        {% endfor %}
    </table>

    <p>
        In this round, you are the {{ player.role }}.
    </p>

    {% formfield player.penny_side with label="I choose:" %}

    {% next_button %}

{% endblock %}

```

ResultsWaitPage

Before a player proceeds to the next round's Choice page, they need to wait for the other player to complete the Choice page as well. So, as usual, we use a WaitPage. Also, once both players have arrived at the wait page, we call the `set_payoffs` method we defined earlier.

```

class ResultsWaitPage(WaitPage):

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

```

ResultsSummary

Let's create `ResultsSummary.html`:

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

{% block title %}
    Final results
{% endblock %}

{% block content %}

    <table class="table">
        <tr>
            <th>Round</th>
            <th>Player and outcome</th>
        </tr>
        {% for p in player_in_all_rounds %}
            <tr>
                <td>{{ p.subsession.round_number }}</td>
                <td>
                    You were the {{ p.role }} and {% if p.is_winner %} won
                    {% else %} lost {% endif %}
                </td>
            </tr>
        {% endfor %}
    </table>

    <p>
        The paying round was {{ paying_round }}.
        Your total payoff is therefore {{ total_payoff }}.
    </p>

{% endblock %}
```

Now we define the corresponding class in `views.py`.

- It only gets shown in the last round, so we set `is_displayed` accordingly.
- We retrieve the value of `paying_round` from `session.vars`
- We get the user's total payoff by summing up how much they made in each round.
- We pass the round history to the template with `player.in_all_rounds()`

In the `Choice` page we used `in_previous_rounds`, but here we use `in_all_rounds`. This is because we also want to include the result of the current round.

```
class ResultsSummary(Page):

    def is_displayed(self):
        return self.subsession.round_number == Constants.num_rounds

    def vars_for_template(self):

        return {
            'total_payoff': sum([p.payoff
                                for p in self.player.in_all_rounds()]),
            'paying_round': self.session.vars['paying_round'],
            'player_in_all_rounds': self.player.in_all_rounds(),
        }
```

The payoff is calculated in a Python “list comprehension”. These are frequently used in the oTree sample games,

so if you are curious you can read online about how list comprehensions work. The same code could be written as:

```
total_payoff = 0
for p in self.player.in_all_rounds():
    total_payoff += p.payoff

return {
    'total_payoff': total_payoff,
    ...
```

Page sequence

Now we define the `page_sequence`:

```
page_sequence = [
    Choice,
    ResultsWaitPage,
    ResultsSummary
]
```

This page sequence will loop for each round. However, `ResultsSummary` is skipped in every round except the last, because of how we set `is_displayed`, resulting in this sequence of pages:

- Choice [Round 1]
- ResultsWaitPage [Round 1]
- Choice [Round 2]
- ResultsWaitPage [Round 2]
- Choice [Round 3]
- ResultsWaitPage [Round 3]
- Choice [Round 4]
- ResultsWaitPage [Round 4]
- ResultsSummary [Round 4]

Add an entry to `SESSION_CONFIGS` in `settings.py`

When we run a real experiment in the lab, we will want multiple groups, but to test the demo we just set `num_demo_participants` to 2, meaning there will be 1 group.

```
{
    'name': 'my_matching_pennies',
    'display_name': "My Matching Pennies (tutorial version)",
    'num_demo_participants': 2,
    'app_sequence': [
        'my_matching_pennies',
    ],
},
```

Reset the database and run

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

7.2 Download & Setup

There are two ways to install oTree that you can choose from:

- The “launcher install” provides a graphical interface for running the oTree server. It also provides a command line interface.
- The “plain install” only lets you run oTree from the command line.

If you encounter any error during installation, please email chris@otree.org with the error message.

7.2.1 Prerequisite: Python 2.7 (not 3.x)

- On Windows: download and install [Python 2.7](#). (oTree does not work with Python 3.) Then add Python to your Path environment variable:
 - Open the Windows Start menu
 - Search for “Edit the system environment variables”, and then click it.
 - Click Environment Variables
 - Select Path in the System variables section
 - Click Edit
 - Add `;C:\Python27;C:\Python27\Scripts` to the end of the list (the paths are separated by semicolons). For example:
`C:\Windows;C:\Windows\System32;C:\Python27;C:\Python27\Scripts`
 - (This assumes that Python was installed to `C:\Python27`.)
- On Mac/Unix, it is very likely that Python is already installed. You can check by opening the Terminal and writing `python` and hit Enter.

If you get something like `-bash: python: command not found` you will have to install it yourself.

Note: Windows/Mac: Verify that it worked by opening your command prompt and entering `python`. You should see the “>>>” prompt.

7.2.2 Launcher install

- Download the launcher from this link:
[oTree_launcher-stable.zip](#)
- Unzip it to an easy-to-access location, like your “Documents” folder.
- Run `otree.py`.
- Initial setup may take 5-10 minutes.
- When the app window launches, click the buttons to create a new deploy and choose a location to store your project files.
- Click the “run server” button
- Note: the oTree launcher is not installed as an app in your Windows start menu or Mac Applications. To reopen the launcher, simply double click `otree.py` again.

Note: For `virtualenv` users: `otree.py` cannot be executed inside a `virtualenv`. You should execute it with a regular non-`virtualenv` python. When it is first executed, it will create a new `virtualenv` and install all its dependencies there.

7.2.3 Plain install

This is an alternative to the “launcher install”.

- From your command line, run the command `pip` to check if Pip is installed. If not, you can download it [here](#).
- Download oTree and unzip it to a convenient location (such as your “Documents” folder). (Or better yet, use Git to clone [this repo](#).)
- In your command line, go to the root directory of the unzipped folder where `requirements_base.txt` is

Note: if you cannot find `requirements_base.txt` make sure you have downloaded `oTree-master.zip`, not `otree-launcher-master.zip`, which is a different download.

- Run these commands (you may need administrator permissions):

```
$ pip install -r requirements_base.txt
```

On Mac, you may need to use `sudo`:

```
$ sudo pip install -r requirements_base.txt
```

(Or you can use a `virtualenv` if you are familiar with that.)

Then run:

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

7.2.4 Explanation: oTree & Django

oTree is built on top of Django.

The oTree folder is a Django project, as explained [here](#).

It comes pre-configured with all the files, settings and dependencies so that it works right away. You should create your apps inside this folder.

If you want, you can delete all the existing example games (like `asset_market`, `bargaining`, etc). Just delete the folders and the corresponding entries in `SESSION_CONFIGS`. Just keep the directories `_static` and `_templates`.

When you install oTree (either using the launcher or running `pip install -r requirements_base.txt`), `otree-core` gets automatically installed as a dependency.

7.2.5 Upgrading/reinstalling oTree

There are several alternatives for upgrading or reinstalling oTree.

(TODO: when to use which)

Upgrade oTree core libraries

In the launcher, click “Version select” and select the most recent version in the menu.

If you are using the “plain install”, change the `otree-core` version number in `requirements_base.txt` and then run:

```
$ pip install -r requirements_base.txt
```

From-scratch reinstallation

- On Windows: Browse to %APPDATA% and delete the folder otree-launcher
- On Mac/Linux: Delete the folder ~/.config/.otree-launcher
- Re-download the launcher

7.3 Conceptual overview

7.3.1 Sessions

In oTree, a session is an event during which participants take part in oTree experiments. An example of a session would be:

“A number of participants will come to the lab and will play trust games (in groups of 2), followed by 2 rounds of ultimatum games, followed by a questionnaire. Participants get paid EUR 10.00 for showing up, plus their earnings from the games.”

To configure a session like this, you would go to `settings.py` and define a “session config”, which is a reusable configuration. This lets you create multiple sessions, all with the same properties.

Add an entry to `SESSION_CONFIGS` like this (assuming you have created apps named `my_app_1` and `my_app_2`):

```
{
    'name': 'my_session_config',
    'display_name': 'My Session Config',
    'participation_fee': 10.00,
    'app_sequence': ['my_app_1', 'my_app_2'],
},
```

Note: Prior to oTree-core 0.3.11, “session config” was known as “session type”. After you upgrade, you can rename `SESSION_TYPES` to `SESSION_CONFIGS`, and `SESSION_TYPE_DEFAULTS` to `SESSION_CONFIG_DEFAULTS`.

This session config is composed of 3 apps:

- Trust game
- Ultimatum game
- Questionnaire

Note that you can reuse apps (such as the `questionnaire` app) in multiple session configs.

Once you have defined a session config, you can run the server, open your browser to the admin interface, and create a new session. You would select “My Session Config” as the configuration to use, and then enter “30” for the number of participants.

An instance of a session would be created, and you would get the start links to distribute to your participants.

In this example, the session would contain 4 “subsessions”:

- Trust game
- Ultimatum game [Round 1]
- Ultimatum game [Round 2]
- Questionnaire

Each subsession can be further divided into groups of players; for example, the trust game subsession would have 15 groups of 2 players each. (Note: groups can be shuffled between subsessions.)

7.3.2 Participants and players

In oTree, the terms “player” and “participant” have distinct meanings. The distinction between a participant and a player is the same as the distinction between a session and a subsession.

A participant is a person who takes part in a session. The participant object contains properties such as the participant’s name, how much they made in the session, and what their progress is in the session.

A player is an instance of a participant in one particular subsession. A participant can be player 1 in the first subsession, player 5 in the next subsession, and so on.

7.4 Applications

In oTree (and Django), an app is a folder containing Python and HTML code. When you create your oTree project, it comes pre-loaded with various apps such as `public_goods` and `dictator`. A session is basically a sequence of apps that are played one after the other.

7.4.1 Creating an app

From the oTree launcher, click the “Terminal” button. (If the button is disabled, make sure you have stopped the server.) When the console window appears, type this:

```
$ otree startapp your_app_name
```

This will create a new app folder based on a oTree template, with most of the structure already set up for you.

The key files are `models.py`, `views.py`, and the HTML files under the `templates/` directory.

Think of this as a skeleton to which you can add as much as you want. You can add your own classes, functions, methods, and attributes, or import any 3rd-party modules.

Then go to `settings.py` and create an entry for your app in `SESSION_CONFIGS` that looks like the other entries.

7.5 Models

This is where you store your data models.

7.5.1 Model hierarchy

Every oTree app needs the following 3 models:

- Subsession
- Group
- Player

A player is part of a group, which is part of a subsession.

7.5.2 Models and database tables

For example, let’s say you are programming an ultimatum game, where in each two-person group, one player makes a monetary offer (say, 0-100 cents), and another player either rejects or accepts the offer. When you analyze your data, you will want your “Group” table to look something like this:

Group ID	Amount offered	Offer accepted
1	50	TRUE
2	25	FALSE
3	50	TRUE
4	0	FALSE
5	60	TRUE

You need to define a Python class that defines the structure of this database table. You define what fields (columns) are in the table, what their data types are, and so on. When you run your experiment, the SQL tables will get automatically generated, and each time users visit, new rows will get added to the tables.

Here is how to define the above table structure:

```
class Group(otree.models.BaseGroup):
    ...
    amount_offered = models.CurrencyField()
    offer_accepted = models.BooleanField()
```

Every time you add, remove, or change a field in `models.py`, you need to run `otree resetdb`, or, in the launcher, click “Reset DB”. (However, you don’t need to run `resetdb` if you only make a change that doesn’t affect your database schema, like modifying `views.py` or an HTML template, etc.)

The full list of available fields is in the Django documentation [here](#).

Additionally, oTree has `CurrencyField`; see [Money and Points](#).

7.5.3 Constants

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` specifies 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://otree-demo.herokuapp.com/p/zuzepona/public_goods/Introduction/1/`

- `players_per_group` (described in [Groups and multiplayer games](#))
- `num_rounds` (described in [Rounds](#))

You should only use `Constants` to store actual constants – things that never change. If you want a “global” variable, you should set a field on the subsession, or use [Global variables](#).

7.5.4 Subsession

The `before_session_starts` Method

You can define this method like this:

```
class Subsession(BaseSubsession):
    ...
    def before_session_starts(self):
        ...
```

This method is executed at the moment when the session is created, meaning it finishes running before the session begins (Hence the name). It is executed once per subsession (i.e. once per round). For example, if your app has 10 rounds, this method will get called 10 times, once for each `Subsession` instance.

It has many uses, such as initializing fields, assigning players to treatments, or shuffling groups. .. [github_icon](#) `image:: /_static/assets/github-icon.png`

7.6 Views

Each page that your players see is defined by a `Page` class in `views.py`. (You can think of “views” as a synonym for “pages”.)

For example, if 1 round of your game involves showing the player a sequence of 5 pages, your `views.py` should contain 5 page classes.

At the bottom of your `views.py`, you must have a `page_sequence` variable that specifies the order in which players are routed through your pages. For example:

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

7.6.1 Pages

Each `Page` class has these methods and attributes:

`def vars_for_template(self)`

A dictionary of variable names and their values, which is passed to the template.

Note: oTree automatically passes `group`, `player`, `subsession`, and `Constants` objects to the template, which you can access in the template, e.g.: `{{ Constants.payoff_if_rejected }}`.

`def is_displayed(self)`

Should 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, if you only want a page to be shown to P2 in each group:

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

`template_name`

The name of the HTML template to display. This can be omitted if the template has the same name as the `Page` class.

Example:

```
# This will look inside:
# 'app_name/templates/app_name/MyView.html'
# (Note that app_name is repeated)
template_name = 'app_name/MyView.html'
```

`timeout_seconds (Remaining time)`

The number of seconds the user has to complete the page. After the time runs out, the page auto-submits.

Example: `timeout_seconds = 20`

When there are 60 seconds left, the page displays a timer warning the participant.

Note: If you are running the production server (`runprodserver`), the page will always submit, even if the user closes their browser window. However, this does not occur if you are running the test server (`runserver`).

`timeout_submission`

Note: Prior to May 26, 2015, this was called `auto_submit_values`.

A dictionary where the keys are the elements of `form_fields`, with the values to be submitted in case of a timeout, or if the experimenter moves the participant forward.

If omitted, then oTree will default to 0 for numeric fields, `False` for boolean fields, and the empty string for text/character fields.

Example: `timeout_submission = {'accept': True}`

If the values submitted `timeout_submission` need to be computed dynamically, you can check [*timeout_happened*](#) and set the values in `before_next_page`.

`timeout_happened`

This boolean attribute is `True` if the page was submitted by timeout. It can be accessed in `before_next_page`:

```
def before_next_page(self):
    if self.timeout_happened:
        self.player.my_random_variable = random.random()
```

This variable is undefined in other methods like `vars_for_template`, because the timeout countdown only starts after the page is rendered.

New in version 0.3.18.

`def before_next_page(self)`

Here you define any code that should be executed after form validation, before the player proceeds to the next page.

`def vars_for_all_templates(self)`

This is not a method on the `Page` class, but rather a top-level function in `views.py`. It is useful when you need certain variables to be passed to multiple pages in your app. Instead of repeating the same values in each `vars_for_template`, you can define it in this function.

7.6.2 Wait pages

Wait pages are necessary when one player needs to wait for others to take some action before they can proceed. For example, in an ultimatum game, player 2 cannot accept or reject before they have seen player 1's offer.

If you have a `WaitPage` in your sequence of pages, then oTree waits until all players in the group have arrived at that point in the sequence, and then all players are allowed to proceed.

If your subsession has multiple groups playing simultaneously, and you would like a wait page that waits for all groups (i.e. all players in the subsession), you can set the attribute `wait_for_all_groups = True` on the wait page.

For more information on groups, see [*Groups and multiplayer games*](#).

Wait pages can define the following methods:

- `def after_all_players_arrive(self)`

This code will be executed once all players have arrived at the wait page. For example, this method can determine the winner of an auction and set each player's payoff.

- `def title_text(self)`

The text in the title of the wait page.

- `def body_text(self)`

The text in the body of the wait page

- `def is_displayed(self)`

If this returns `False` then the player skips the wait page.

If all players in the group skip the wait page, then `after_all_players_arrive()` will not be run.

Note: `is_displayed` on wait pages was added in otree-core 0.3.7

7.7 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](#).

7.7.1 Template blocks

Instead of writing the full HTML of your page, for example:

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

You define 2 blocks:

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

{% block content %}
  Body HTML goes here.

  {% formfield player.contribution with label="What is your contribution?" %}

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

You may want to customize the appearance or functionality of all pages in your app (e.g. by adding custom CSS or JavaScript). To do this, edit the file `_templates/global/Base.html`.

7.7.2 JavaScript and CSS

If you have JavaScript and/or CSS in your page, you should put them in blocks called `scripts` and `styles`, respectively. They should be located outside the `content` block, like this:

```
{% block content %}
  <p>This is some HTML.</p>
{% endblock %}

{% block styles %}

  <!-- define a style -->
  <style type="text/css">
    .bid-slider {
      margin: 1.5em auto;
      width: 70%;
```

```
    }
</style>

<!-- or reference a static file -->
<link href="{% static "my_app/style.css" %}" rel="stylesheet">

{% endblock %}

{% block scripts %}

    <!-- define a script -->

    <script type="text/javascript">
    jQuery(document).ready(function ($) {
        var PRIVATE_VALUE = {{ player.private_value.to_number|escapejs }};

        var input = $('#id_bid_amount');

        $('.bid-slider').slider({
            min: 0,
            max: 100,
            slide: function (event, ui) {
                input.val(ui.value);
                updateBidValue();
            },
        });
    });
    </script>

    <!-- or reference a static file -->
    <script type="text/javascript" src="{% static "my_app/script.js" %}"></script>
{% endblock %}
```

The reasons for putting scripts and styles in separate blocks are:

- It keeps your code organized
- jQuery is only loaded at the bottom of the page, so if you reference the jQuery \$ variable in the content block, it will be undefined.

7.7.3 Customizing the base template

For all apps

If you want to apply a style or script to all pages in all games, you should modify the template `_templates/global/Base.html`. You should put any scripts in the `global_scripts` block, and any styles in the `global_styles` block.

You can also modify `_static/global/custom.js` and `_static/global/custom.css`, which as you can see are loaded by `_templates/global/Base.html`.

Note: If you downloaded oTree prior to September 7, 2015, you need to update `_templates/global/Base.html` to the latest version [here](#).

Old versions have a bug where `custom.js` was not being loaded. See [here](#) for more info.

For one app

If you want to apply a style or script to all pages in one app, you should create a base template for all templates in your app, and put blocks called `app_styles` or `app_scripts` in this base template.

For example, if your app's name is `public_goods`, then you would create a file called `public_goods/templates/public_goods/Base.html`, and put this inside it:

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

{% block app_styles %}

    <style type="text/css">
        /* custom styles go here */
    </style>

{% endblock %}
```

Then each `public_goods` template would inherit from this template:

```
{% extends "public_goods/Base.html" %}
{% load staticfiles otree_tags %}
...
```

7.7.4 Static content (images, videos, CSS, JavaScript)

To include images, CSS, or JavaScript in your pages, make sure your template has loaded `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.

Put your files in that subfolder. You can then reference them in a template like this:

```

```

7.7.5 Plugins

oTree comes pre-loaded with the following plugins and libraries.

Bootstrap

oTree comes with [Bootstrap](#), a popular library for customizing a website's user interface.

You can use it if you want a [custom style](#), or a [specific component](#) like a table, alert, progress bar, label, etc. You can even make your page dynamic with elements like [popovers](#), [modals](#), and [collapsible text](#).

To use Bootstrap, usually you add a `class=` attributes to your HTML element.

For example, the following HTML will create a "Success" alert:

```
<div class="alert alert-success">Great job!</div>
```

Graphs and charts with HighCharts

oTree comes pre-loaded with [HighCharts](#), which you can use to draw pie charts, line graphs, bar charts, time series, and other types of plots.

You can find examples in the library of how to use it.

To pass data like a list of values from Python to HighCharts, you should first pass it through the `otree.common.safe_json()` function. This converts to the correct JSON syntax and also uses `mark_safe` for the template.

Example:

```
>>> a = [0, 1, 2, 3, 4, None]
>>> safe_json(a)
'[0, 1, 2, 3, 4, null]'
```

LaTeX

oTree comes pre-loaded with [KaTeX](#); you can insert LaTeX equations like this:

```
<span class="latex">
  1 + i = (1 + r) (1 + \pi)
</span>
```

7.7.6 oTree on mobile devices

Since oTree uses Bootstrap for its user interface, your oTree app should work on all major browsers (Chrome/Internet Explorer/Firefox/Safari). When participants visit on a smartphone or tablet (e.g. iOS/Android/etc.), they should see an appropriately scaled down “mobile friendly” version of the site. This will generally not require much effort on your part since Bootstrap does it automatically, but if you plan to deploy your app to participants on mobile devices, you should test it out on a mobile device during development, since some HTML code doesn’t look good on mobile devices.

7.8 Forms

Each page in oTree can contain a form, which the player should fill out and submit by clicking the “Next” button. To create a form, first you should go to `models.py` and define fields on your `Player` or `Group`. Then, in your `Page` class, you can define `form_models` to specify the model that this form modifies (either `models.Player` or `models.Group`), and `form_fields`, which is a list of the fields from that model.

When the user submits the form, the submitted data is automatically saved back to the field in your model.

7.8.1 Forms in templates

You should include form fields by using a `{% formfield %}` element. You generally do not need to write raw HTML for forms (e.g. `<input type="text" id="...">`).

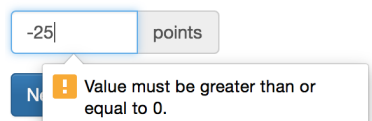
7.8.2 User Input Validation

The player must submit a valid form before they get routed 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.

Example 1:

You are Participant A. Now you have 100 points. How many points will you send to participant B?

Please enter a number from 0 to 100:

A screenshot of an oTree form. At the top, it says "You are Participant A. Now you have 100 points. How many points will you send to participant B?". Below this is a label "Please enter a number from 0 to 100:". There is a text input field containing "-25" and a button labeled "points". Below the input field, there is a blue "Next" button and a yellow error message box that says "Value must be greater than or equal to 0."

Example 2:

Please fix the errors in the form.

This first screen is to give a template for understanding questions and to show you a numeric entry field with restricted values. Just for fun the field accepts any answer that is an odd negative integer or a non-negative integer. Thus it will reject an invalid entry of, say -2, and ask you to try again after you click next, but if you enter a valid, but wrong answer, say 1 it will accept it but give you feedback.

How many understanding questions are there?

Please enter an odd negative integer, or a non-negative integer:

Your entry is invalid.

Next

oTree automatically validates all input submitted by the user. For example, if you have a form containing a `PositiveIntegerField`, oTree will not let the user submit values that are not positive integers, like -1, 1.5, or hello.

You can specify additional validation. For example, here is how you would require an integer to be between 12 and 24:

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

You can constrain the user to a predefined list of choices by using `choices`:

```
year_in_school = models.CharField(
    choices=['Freshman', 'Sophomore', 'Junior', 'Senior'])
```

The user will then be presented a dropdown menu instead of free text input.

If you would like a specially formatted value displayed to the user that is different from the values stored internally, `choices` can be a list consisting itself of tuples of two items. The first element in each tuple is the value and the second element is the human-readable label.

For example:

```
year_in_school = models.CharField(
    choices=[
        ('FR', 'Freshman'),
        ('SO', 'Sophomore'),
        ('JR', 'Junior'),
        ('SR', 'Senior'),
    ]
)
```

After the field has been set, you can access the human-readable name using `get_FOO_display`, like this:

```
self.get_year_in_school_display() # returns e.g. 'Sophomore'
```

If a field is optional, you can do:

```
offer = models.PositiveIntegerField(blank=True)
```

Dynamic validation

If you need a form's choices or validation logic to depend on some dynamic calculation, then you can instead define one of the below methods in your `Page` class in `views.py`.

- `def {field_name}_choices(self)`

Example:

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

- `def {field_name}_min(self)`

The dynamic alternative to `min`.

- `def {field_name}_max(self)`

The dynamic alternative to `max`.

- `def {field_name}_error_message(self, value)`

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:

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

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 would define the `error_message` method in your Page class:

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

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. But if you do this, you must make sure your template also contains conditional logic so that the right `formfield` elements are included.

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 %}
```

`form` is a special variable. It is a Django form object, which is an iterable whose elements are Django form field objects. `formfield` can take as an argument a Django field object, or it can be an expression like `{% formfield player.foo %}` and `{% formfield group.foo %}`, but `player.foo` must be written as a literal rather than assigning `somevar = player.foo` and then doing `{% formfield somevar %}`.

If you use this technique and want a custom label on each field, you can add a `verbose_name` to the model field, as described in the Django documentation, e.g.:

```
contribution = models.CurrencyField(
    verbose_name="How much will you contribute?")
```

Forms with a dynamic vector of fields

Let's say you want a form with a vector of `n` fields that are identical, except for some numerical index, e.g.:

Furthermore, suppose `n` is variable (can range from 1 to `N`).

Currently in oTree, you can only define a fixed number of fields in a model. So, you should define in `models.py` `N` fields (`contribution_1...contribution_N...`), and then use `get_form_fields` as described above to dynamically return a list with the desired subset of these fields.

7.8.3 Widgets

The full list of form input widgets offered by Django is [here](#).

oTree additionally offers

- `RadioSelectHorizontal` (same as `RadioSelect` but with a horizontal layout, as you would see with a Likert scale)
- `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)`

7.8.4 Custom widgets and hidden fields

It's not mandatory to use the `{% formfield %}` element; you can write the raw HTML for any form input if you wish to customize its behavior or appearance. Just include an `<input>` element with the same name attribute as the field. For example, if you want a hidden input, you can do this:

```
# models.py
my_hidden_input = models.PositiveIntegerField()

# views.py
form_fields = ['my_hidden_input', 'some_other_field']

# HTML template
<input type="hidden" name="my_hidden_input"
      value="5" id="id_my_hidden_input"/>
```

Then you can use JavaScript to set the value of that input, by selecting the element by id `id_my_hidden_input`.

For simple widgets you can use jQuery; for more complex or custom form interfaces, you can use a front-end framework with databinding, like React or Polymer.

If you want your custom widget's style to look like the rest of the oTree widgets, you should look at the generated HTML from the `{% formfield %}` tag. You can copy and paste the markup into the template and use that as a starting point for modifications.

7.9 Self and object model

In oTree code, you will see the variable `self` in many places. In Python, `self` refers to the object whose class you are currently in.

For example, in this example, `self` refers to a `Player` object:

```
class Player(object):

    def my_method(self):
        return self.my_field
```

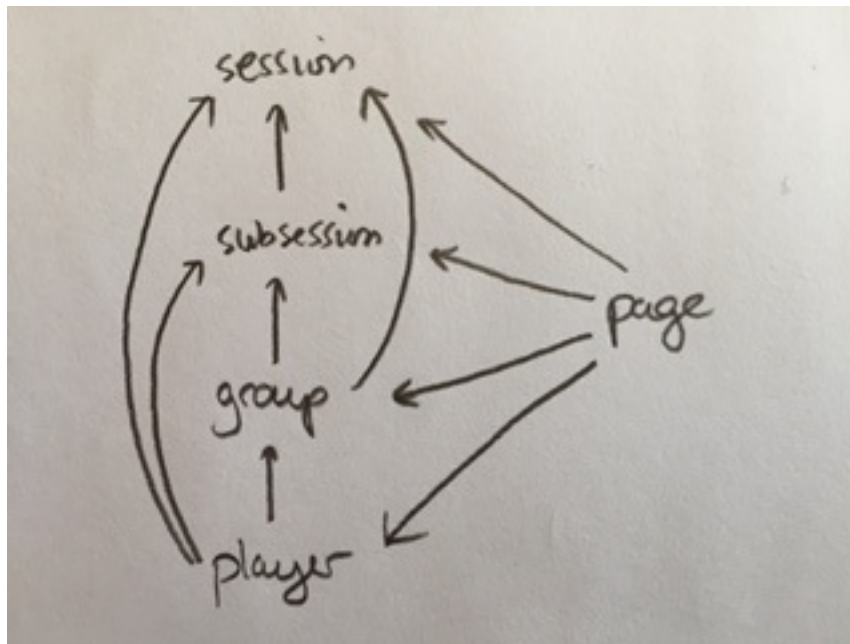
In the next example, however, `self` refers to a `Group` object:

```
class Group(object):

    def my_method(self):
        return self.my_field
```

`self` is conceptually similar to the word “me”. You refer to yourself as “me”, but others refer to you by your name. And when your friend says the word “me”, it has a different meaning from when you say the word “me”.

oTree's different objects are all connected, as illustrated by this diagram.



Player, group, subsession, and session are in a simple hierarchy, 'session' being at the top and 'player' being at the bottom. Then, the 'page' has a pointer to all 4 of these objects.

For example, if you are in a method on the `Player` class, you can access the player's payoff with `self.payoff` (because `self` is the player). But if you are inside a `Page` class in `views.py`, the equivalent expression is `self.player.payoff`, which traverses the pointer from 'page' to 'player'.

Here are some code examples to illustrate:

```

class Session(...) # this class is defined in oTree-core
    def example(self):

        # current session object
        self

        # parent objects
        self.config

        # child objects
        self.get_subsessions()
        self.get_participants()

class Participant(...) # this class is defined in oTree-core
    def example(self):

        # current participant object
        self

        # parent objects
        self.session

        # child objects
        self.get_players()
  
```

in your `models.py`

```

class Subsession(otree.models.Subsession):
    def example(self):
  
```



```

    # current subsession object
    self

    # parent objects
    self.session

    # child objects
    self.get_groups()
    self.get_players()

    # accessing previous Subsession objects
    self.in_previous_rounds()
    self.in_all_rounds()

class Group(otree.models.Group):
    def example(self):

        # current group object
        self

        # parent objects
        self.session
        self.subsession

        # child objects
        self.get_players()

class Player(otree.models.Player):

    def example(self):

        # current player object
        self

        # method you defined on the current object
        self.my_custom_method()

        # parent objects
        self.session
        self.subsession
        self.group
        self.participant

        self.session.config

        # accessing previous player objects
        self.in_previous_rounds()

        # equivalent to self.in_previous_rounds() + [self]
        self.in_all_rounds()

```

in your `views.py`

```

class MyPage(Page):
    def example(self):

        # current page object
        self

        # parent objects
        self.session
        self.subsession
        self.group

```

```
self.player

# example of chaining lookups
self.player.participant
self.session.config
```

You can follow pointers in a transitive manner. For example, if you are in the `Page` class, you can access the participant as `self.player.participant`. If you are in the `Player` class, you can access the session config as `self.session.config`.

Note: Prior to oTree-core 0.3.11, `self.session.config` was known as `self.session.session_type`.

7.10 Groups and multiplayer games

To create a multiplayer game, go to your app's `models.py` and set `Constants.players_per_group`. For example, in a 2-player game like an ultimatum game or prisoner's dilemma, you would set this to 2. If your app does not involve dividing the players into multiple groups, then set it to `None`. e.g. it is a single-player game or a game where everybody in the subsession interacts together as 1 group. In this case, `self.group.get_players()` will return everybody in the subsession. If you need your groups to have uneven sizes (for example, 2 vs 3), you can do this: `players_per_group=[2, 3]`; in this case, if you have a session with 15 players, the group sizes would be `[2, 3, 2, 3, 2, 3]`.

Each player has an attribute `id_in_group`, which is an integer, which will tell you if it is player 1, player 2, etc.

Group objects have the following methods:

- `get_players()`: returns a list of the players in the group.
- `get_player_by_id(n)`: Retrieves the player in the group with a specific `id_in_group`.
- `get_player_by_role(r)`. The argument to this method is a string that looks up the player by their role value. (If you use this method, you must define the `role` method on the player model, which should return a string that depends on `id_in_group`.)

Player objects have methods `get_others_in_group()` and `get_others_in_subsession()` that return a list of the other players in the group and subsession. For example, with 2-player groups you can get the partner of a player, with this method on the `Player`:

```
def get_partner(self):
    return self.get_others_in_group()[0]
```

7.10.1 Group re-matching between rounds

For the first round, the players are split into groups of `Constants.players_per_group`. In subsequent rounds, by default, the groups chosen are kept the same. If you would like to change this, you can define the grouping logic in `Subsession.before_session_starts` (for more info see [The before_session_starts Method](#)).

A group has a method `set_players` that takes as an argument a list of the players to assign to that group, in order. Alternatively, a subsession has a method `set_groups` that takes as an argument a list of lists, with each sublist representing a group.

For example, if you want players to be reassigned to the same groups but to have roles randomly shuffled around within their groups (e.g. so player 1 will either become player 2 or remain player 1), you would do this:

```
def before_session_starts(self):
    for group in self.get_groups():
        players = group.get_players()
```

```
players.reverse()
group.set_players(players)
```

7.10.2 Re-matching based on results of previous rounds

Your experimental design may involve re-matching players based on the results of a previous subsession. For example, you may want the highest-ranked players in round 1 to play against each other in round 2.

You cannot accomplish this using `before_session_starts`, because this method is run when the session is created, before players begin playing.

Instead, you should make a `WaitPage` with `wait_for_all_groups=True` and put the shuffling code in `after_all_players_arrive`. For example:

```
class ShuffleWaitPage(WaitPage):
    wait_for_all_groups = True

    def after_all_players_arrive(self):

        group_matrix = [g.get_players() for g in self.subsession.get_groups()]

        # ... some code to permute this matrix

        self.subsession.set_groups(group_matrix)
```

After this wait page, the players will be reassigned to their new groups.

Example: re-matching by rank

For example, let's say that in each round of an app, players get a numeric score for some task. In the first round, players are matched randomly, but in the subsequent rounds, you want players to be matched with players who got a similar score in the previous round.

First of all, at the end of each round, you should assign each player's score to `participant.vars` so that it can be easily accessed in other rounds, e.g. `self.player.participant.vars['score'] = 10`.

Then, you would define the following page and put it at the beginning of `page_sequence`:

```
class ShuffleWaitPage(WaitPage):
    wait_for_all_groups = True

    # we can't shuffle at the beginning of round 1,
    # because the score has not been determined yet
    def is_displayed(self):
        return self.subsession.round_number > 1

    def after_all_players_arrive(self):

        # sort players by 'score'
        # see python docs on sorted() function
        sorted_players = sorted(
            self.subsession.get_players(),
            key=lambda player: player.participant.vars['score']
        )

        # chunk players into groups
        group_matrix = []
        ppg = Constants.players_per_group
        for i in range(0, len(sorted_players), ppg):
            group_matrix.append(sorted_players[i:i+ppg])
```

```
# set new groups
self.subsession.set_groups(group_matrix)
```

7.10.3 More complex grouping logic

If you need something more flexible or complex than what is allowed by `players_per_group`, you can specify the grouping logic yourself in `before_session_starts`, using the `get_players()` and `set_groups()` methods described above.

Fixed number of groups with a divisible number of players

For example, let's say you always want 8 groups, regardless of the number of players in the session. So, if there are 16 players, you will have 2 players per group, and if there are 32 players, you will have 4 players per group.

You can accomplish this as follows:

```
class Constants:
    players_per_group = None
    groups = 8
    ... # etc

class Subsession(otree.models.BaseSubsession):

    def before_session_starts(self):
        if self.round_number == 1:

            # create the base for number of groups
            num_players = len(self.get_players())
            num_groups = len(Constants.groups)
            players_per_group = [int(num_players/num_groups)] * num_groups

            # verify if all players are assigned
            idxg = 0
            while sum(players_per_group) < num_players:
                players_per_group[idxg] += 1
                idxg += 1

            # reassignment of groups
            list_of_lists = []
            players = self.get_players()
            for g_idx, g_size in enumerate(players_per_group):
                offset = 0 if g_idx == 0 else sum(players_per_group[:g_idx])
                limit = offset + g_size
                group_players = players[offset:limit]
                list_of_lists.append(group_players)
            self.set_groups(list_of_lists)
```

Fixed number of groups with a no divisible number of players

Lets make a more complex example based on the previous one. Let's say we need to divide 20 players into 8 groups randomly. The problem is that $20/8 = 2.5$.

So the more easy solution is to make the first 4 groups with 3 players, and the last 4 groups with only 2 players.

```
class Constants:
    players_per_group = None
    groups = 8
    ... # etc

class Subsession(otree.models.BaseSubsession):

    def before_session_starts(self):
```

```

# if you want to change the
if self.round_number == 1:

    # extract and mix the players
    players = self.get_players()
    random.shuffle(players)

    # create the base for number of groups
    num_players = len(players)
    num_groups = len(Constants.groups)

    # create a list of how many players must be in every group
    # the result of this will be [2, 2, 2, 2, 2, 2, 2, 2]
    # obviously 2 * 8 = 16
    players_per_group = [int(num_players/num_groups)] * num_groups

    # add one player in order per group until the sum of size of
    # every group is equal to total of players
    idxg = 0
    while sum(players_per_group) < num_players:
        players_per_group[idxg] += 1
        idxg += 1
        if idxg >= len(players_per_group):
            idxg = 0

    # reassignment of groups
    list_of_lists = []
    for g_idx, g_size in enumerate(players_per_group):
        # it is the first group the offset is 0 otherwise we start
        # after all the players already exhausted
        offset = 0 if g_idx == 0 else sum(players_per_group[:g_idx])

        # the asignment of this group end when we assign the total
        # size of the group
        limit = offset + g_size

        # we select the player to add
        group_players = players[offset:limit]
        list_of_lists.append(group_players)
    self.set_groups(list_of_lists)

```

7.11 Money and Points

In many experiments, participants play for currency: either virtual points, or real money. oTree supports both scenarios; you can switch from points to real money by setting `USE_POINTS = False` in `settings.py`.

You can specify the payment currency in `settings.py`, by setting `REAL_WORLD_CURRENCY_CODE` to “USD”, “EUR”, “GBP”, and so on. This means that all currency amounts the participants see will be automatically formatted in that currency, and at the end of the session when you print out the payments page, amounts will be displayed in that currency. The session’s `participation_fee` is also displayed in this currency code.

In oTree apps, currency values have their own data type. You can define a currency value with the `c()` function, e.g. `c(10)` or `c(0)`. Correspondingly, there is a special model field for currency values: `CurrencyField`.

Each player has a `payoff` field, which is a `CurrencyField`. Its initial value is `None`. If you want to initialize it to 0, you should do so in `before_session_starts`, e.g.:

```

def before_session_starts(self):
    for p in self.get_players():
        p.payoff = 0

```

Currency values work just like numbers (you can do mathematical operations like addition, multiplication, etc), but when you pass them to an HTML template, they are automatically formatted as currency. For example, if you set `player.payoff = c(1.20)`, and then pass it to a template, it will be formatted as \$1.20 or 1,20 €, etc., depending on your `REAL_WORLD_CURRENCY_CODE` and `LANGUAGE_CODE` settings.

Money amounts are expressed with 2 decimal places by default; you can change this with the setting `REAL_WORLD_CURRENCY_DECIMAL_PLACES`.

Note: instead of using Python's built-in `range` function, you should use oTree's `currency_range` with currency values. It takes 3 arguments (start, stop, step), just like `range`. However, note that it is an inclusive range. For example, `currency_range(c(0), c(0.10), c(0.02))` returns something like:

```
[Money($0.00), Money($0.02), Money($0.04),  
 Money($0.06), Money($0.08), Money($0.10)]
```

In templates, instead of using the `c()` function, you should use the `|c` filter. For example, `{{ 20|c }}` displays as 20 points.

7.11.1 Assigning payoffs

Each player has a `payoff` field, which is a `CurrencyField`. If your player makes money, you should store it in this field. `player.participant.payoff` is the sum of the payoffs a participant made in each subsession (either in points or real money). At the end of the experiment, a participant's total profit can be accessed by `participant.money_to_pay()`; it is calculated by converting `participant.payoff` to real-world currency (if `USE_POINTS` is `True`), and then adding `self.session.config['participation_fee']`.

7.11.2 Points (i.e. “experimental currency”)

Sometimes it is preferable for players to play games for points or “experimental currency units”, which are converted to real money at the end of the session. You can set `USE_POINTS = True` in `settings.py`, and then in-game currency amounts will be expressed in points rather than dollars or euros, etc.

For example, `c(10)` is displayed as 10 points. You can specify the conversion rate to real money in `settings.py` by providing a `real_world_currency_per_point` key in the session config dictionary. For example, if you pay the user 2 cents per point, you would set `real_world_currency_per_point = 0.02`.

Points are integers by default. You can change this by setting `POINTS_DECIMAL_PLACES` in `settings.py`. (e.g. set it to 2 if you want 2 decimal places, so you can get amounts like 3.14 points).

You can change the name “points” to something else like “tokens” or “credits”, by setting `POINTS_CUSTOM_NAME`. (However, if you switch your language setting to one of oTree's supported languages, the name “points” is automatically translated, e.g. “puntos” in Spanish.)

New in version 0.3.30.

Converting points to real world currency

You can convert a point amount to money using the `to_real_world_currency()` method. In the above example, that would be:

```
>>> c(10).to_real_world_currency(self.session)  
$0.20
```

This method requires that `self.session` be passed as an argument, because different sessions can have different conversion rates).

7.12 Treatments

If you want to assign participants to different treatment groups, you can put the code in the subsession’s `before_session_starts` method (for more info see [The `before_session_starts` Method](#)). For example, if you want some participants to have a blue background to their screen and some to have a red background, you would first define a `color` field on the `Player` model:

```
class Player(BasePlayer):
    # ...

    color = models.CharField()
```

Then you can assign to this field randomly:

```
class Subsession(BaseSubsession):

    def before_session_starts(self):
        # randomize to treatments
        for player in self.get_players():
            player.color = random.choice(['blue', 'red'])
```

(To actually set the screen color you would need to pass `player.color` to some CSS code in the template, but that part is omitted here.)

You can also assign treatments at the group level (put the `CharField` in the `Group` class and change the above code to use `get_groups()` and `group.color`).

If your game has multiple rounds, note that the above code gets executed for each round. So if you want to ensure that participants are assigned to the same treatment group each round, you should set the property at the participant level, which persists across subsessions, and only set it in the first round:

```
class Subsession(BaseSubsession):

    def before_session_starts(self):
        if self.round_number == 1:
            for p in self.get_players():
                p.participant.vars['color'] = random.choice(['blue', 'red'])
```

Then elsewhere in your code, you can access the participant’s color with `self.player.participant.vars['color']`.

There is no direct equivalent for `participant.vars` for groups, because groups can be re-shuffled across rounds. You should instead store the variable on one of the participants in the group:

```
def before_session_starts(self):
    if self.round_number == 1:
        for g in self.get_groups():
            p1 = g.get_player_by_id(1)
            p1.participant.vars['color'] = random.choice(['blue', 'red'])
```

Then, when you need to access a group’s color, you would look it up like this:

```
p1 = self.group.get_player_by_id(1)
color = p1.participant.vars['color']
```

For more on vars, see [Accessing data from previous rounds: technique 2](#).

The above code makes a random drawing independently for each player, so you may end up with an imbalance between “blue” and “red”. To solve this, you can alternate treatments, using `itertools.cycle`:

```
import itertools

class Subsession(otree.models.BaseSubsession):
```

```
def before_session_starts(self):
    treatments = itertools.cycle([True, False])
    for g in self.get_groups():
        g.treatment = treatments.next()
```

7.12.1 Choosing which treatment to play

In the above example, players got randomized to treatments. This is useful in a live experiment, but when you are testing your game, it is often useful to choose explicitly which treatment to play. Let’s say you are developing the game from the above example and want to show your colleagues both treatments (red and blue). You can create 2 session configs in `settings.py` that have the same keys to session config dictionary, except the `treatment` key:

```
SESSION_CONFIGS = [
    {
        'name': 'my_game_blue',
        # other arguments...

        'treatment': 'blue',
    },
    {
        'name': 'my_game_red',
        # other arguments...
        'treatment': 'red',
    },
]
```

Then in the `before_session_starts` method, you can check which of the 2 session configs it is:

```
def before_session_starts(self):
    for p in self.get_players():
        if 'treatment' in self.session.config:
            # demo mode
            p.color = self.session.config['treatment']
        else:
            # live experiment mode
            p.color = random.choice(['blue', 'red'])
```

Then, when someone visits your demo page, they will see the “red” and “blue” treatment, and choose to play one or the other. If the demo argument is not passed, the color is randomized.

7.13 Rounds

In oTree, “rounds” and “subsessions” are almost synonymous. The difference is that “rounds” refers to a sequence of subsessions that are in the same app. So, a session that consists of a prisoner’s dilemma iterated 3 times, followed by an exit questionnaire, has 4 subsessions, which consists of 3 rounds of the prisoner’s dilemma, and 1 round of the questionnaire.

7.13.1 Round numbers

You can specify how many rounds a game should be played in `models.py`, in `Constants.num_rounds`.

Subsession objects have an attribute `round_number`, which contains the current round number, starting from 1.

7.13.2 Accessing data from previous rounds: technique 1

Each round has separate Subsession, Group, and Player objects. For example, let's say you set `self.player.my_field = True` in round 1. In round 2, if you try to access `self.player.my_field`, you will find its value is `None` (assuming that is the default value of the field). This is because the `Player` objects in round 1 are separate from `Player` objects in round 2.

To access data from a previous round, you can use the methods `in_previous_rounds()` and `in_all_rounds()` on `player`, `group`, and `subsession` objects.

`player.in_previous_rounds()` and `player.in_all_rounds()` each return a list of players representing the same participant in previous rounds of the same app. The difference is that `in_all_rounds()` includes the current round's player.

For example, if you wanted to calculate a participant's payoff for all previous rounds of a game, plus the current one:

```
cumulative_payoff = sum([p.payoff for p in self.player.in_all_rounds()])
```

Similarly, subsession objects have methods `in_previous_rounds()` and `in_all_rounds()` that work the same way.

Group objects also have `in_previous_rounds()` and `in_all_rounds()` methods, but note that if you re-shuffle groups between rounds, then these methods may not return anything meaningful (their behavior in this situation is unspecified).

Note: `Group.in_all_rounds()` and `Group.in_previous_rounds()` were added in otree-core 0.3.8

7.13.3 Accessing data from previous rounds: technique 2

`in_all_rounds()` only is useful when you need to access data from a previous round of the same app. If you want to pass data between subsessions of different app types (e.g. the participant is in the questionnaire and needs to see data from their ultimatum game results), you should store this data in the participant object, which persists across subsessions. Each participant has a field called `vars`, which is a dictionary that can store any data about the player. For example, if you ask the participant's name in one subsession and you need to access it later, you would store it like this:

```
self.player.participant.vars['first name'] = 'John'
```

Then in a future subsession, you would retrieve this value like this:

```
self.player.participant.vars['first name'] # returns 'John'
```

7.13.4 Global variables

For session-wide globals, you can use `self.session.vars`.

This is a dictionary just like `participant.vars`.

7.13.5 Variable number of rounds

If you want a variable number of rounds, consider setting `num_rounds` to some high number, and then in your app, conditionally hide the `{% next_button %}` element, so that the user cannot proceed to the next page.

7.14 Localization

oTree's participant interface has been translated to the following languages:

- French
- German
- Italian
- Russian
- Spanish
- Hungarian

This means that all built-in text that gets displayed to participants is available in these languages. This includes things like:

- Form validation messages
- Wait page messages
- Dates, times and numbers (e.g. “1.5” vs “1,5”)

So, as long as you write your app’s text in one of these languages, all text that participants will see will be in that language. For more information, see the Django documentation on [translation](#) and [format localization](#).

However, oTree’s admin/experimenter interface is currently only available in English, and the existing sample games have not been translated to any other languages.

7.14.1 Changing the language setting

Go to `settings.py`, change `LANGUAGE_CODE`, and restart the server.

7.14.2 Writing your app in multiple languages

You may want your own app to work in multiple languages. For example, let’s say you want to run the same experiment with English and Chinese participants.

For this, you can use Django’s [translation](#) system.

A quick summary:

- In templates, use `{% blocktrans trimmed %}...{% endblocktrans %}`
- In Python code, use `ugettext`.
- Run `django-admin makemessages` to create the `.po` files
- Edit the `.po` file in [Poedit](#)
- Run `django-admin compilemessages`
- Go to `settings.py`, change `LANGUAGE_CODE`, and restart the server.

7.14.3 Volunteering to localize oTree

You are invited to contribute support for your own language in oTree.

It’s a simple task; you provide translations of about 20 English phrases. Currently we are only translating the participant interface, although we plan to translate the admin interface and launcher later.

[Here](#) is an example of an already completed translation to French.

We are especially looking for volunteers to translate the following files to their respective languages:

- [Chinese](#)
- [Korean](#)
- [Japanese](#)

You can download the file (using “Save As”) and edit it directly in your text editor, or use [Poedit](#).

Please contact chris@otree.org.

7.15 Manual testing

You can launch your app on your local development machine to test it, and then when you are satisfied, you can deploy it to a server.

7.15.1 Testing locally

You will be testing your app frequently during development, so that you can see how the app looks and feels and discover bugs during development. To test your app, run the server in the oTree launcher. You may need to reset the database first.

Click on a session name and you will get a start link for the experimenter, as well as the links for all the participants. You can open all the start links in different tabs and simulate playing as multiple participants simultaneously.

You can send the demo page link to your colleagues or publish it to a public audience.

7.15.2 Debugging

Once you start playing your app, you will sometimes get a yellow Django error page with lots of details. To troubleshoot this, look at the error message and “Exception location” fields. If the exception location is somewhere outside of your app’s code (like if it points to an installed module like Django or oTree), look through the “Trace-back” section to see if it passes through your code. Once you have found a reference to a line of code in your app, go to that line of code and see if the error message can help you pinpoint an error in your code. Googling the error name or message will often take you to pages that explain the meaning of the error and how to fix it.

Debugging with PyCharm

PyCharm has an excellent debugger that you should be using continuously. You can insert a breakpoint into your code by clicking in the left-hand margin on a line of code. You will see a little red dot. Then reload the page and the debugger will pause when it hits your line of code. At this point you can inspect the state of all the local variables, execute print statements in the built-in interpreter, and step through the code line by line.

More on the PyCharm debugger [here](#).

Debugging in the command shell

To test your app from an interactive Python shell, do:

```
$ otree shell
```

Then you can debug your code and inspect objects in your database. For example, if you already ran a “public goods game” session in your browser, you can access the database objects in Python like this:

```
>>> from public_goods.models import Player
>>> players = Player.objects.all()
>>> ...
```

7.16 Automated testing (bots)

Your app's `tests.py` lets you define “bots” that simulate multiple players simultaneously playing your app.

Tests with dozens of bots complete with in seconds, and afterward automated tests can be run to verify correctness of the app (e.g. to ensure that payoffs are being calculated correctly).

This automated test system saves the programmer the effort of having to re-test the application every time something is changed.

7.16.1 Launching tests

oTree tests entire sessions, rather than individual apps in isolation. This is to make sure the entire session runs, just as participants will play it in the lab.

Let's say you want to test the session named `ultimatum` in `settings.py`. To test, click the “Terminal” button in the oTree launcher run the following command from your project's root directory:

```
$ otree test ultimatum_game
```

This command will test the session, with the number of participants specified in `settings.py`. For example, `num_bots` is 30, then when you run the tests, 30 bots will be instantiated and will play concurrently.

To run tests for all sessions in `settings.py`, run:

```
$ otree test
```

7.16.2 Writing tests

Tests are contained in your app's `tests.py`. Fill out the `play_round()` method of your `PlayerBot`. It should simulate each page submission. For example:

```
self.submit(views.Start)
self.submit(views.Offer, {'offer_amount': 50})
```

Here, we first submit the `Start` page, which does not contain a form. The next page is `Offer`, which contains a form whose field is called `offer_amount`, which we set to 50.

If a page contains several submissions, the syntax looks like

```
self.submit(views.Offer, {'first_offer_amount': 50, 'second_offer_amount': 150, 'third_offer_amoun
```

The test system will raise an error if the bot submits invalid input for a page, or if it submits pages in the wrong order.

Rather than programming many separate bots, you program one bot that can play any variation of the game, using conditional logic. For example, here is how you would play if one treatment group sees a “threshold” page but the other treatment group should see an “accept” page:

```
if self.group.threshold:
    self.submit(views.Threshold, {'offer_accept_threshold': 30})
else:
    self.submit(views.Accept, {'offer_accepted': True})
```

To get the maximal benefit, your bot should thoroughly test all parts of your code. Here are some ways you can test your app:

- Ensure that it correctly rejects invalid input. For example, if you ask the user to enter a number that is a multiple of 3, you can verify that entering 4 will be rejected by using the `submit_invalid` method as follows. This line of code will raise an error if the submission is *accepted*:

```
self.submit_invalid(views.EnterNumber, {'multiple_of_3': 4})
```

- You can put assert statements in the bot's `validate_play()` method to check that the correct values are being stored in the database. For example, if a player's bonus is defined to be 100 minus their offer, you can check your program is calculating it correctly as follows:

```
self.submit (views.Offer, {'offer': 30})
assert self.player.bonus == 70
```

- You can use random amounts to test that your program can handle any type of random input:

```
self.submit (views.Offer, {'offer': random.randint(0,100)})
```

Bots can either be programmed to simulate playing the game according to an ordinary strategy, or to test “boundary conditions” (e.g. by entering invalid input to see if the application correctly rejects it). Or yet the bot can enter random input on each page.

7.17 Admin

oTree comes with an admin interface, so that experimenters can manage sessions, monitor the progress of live sessions, and export data after sessions.

Open your browser to the root url of your web application. If you're developing locally, this will be `http://127.0.0.1:8000/`.

7.17.1 Authentication

When you first install oTree, The entire admin interface is accessible without a password. However, when you are ready to launch your oTree app, you should password protect the admin so that visitors and participants cannot access sensitive data.

If you are launching an experiment and want visitors to only be able to play your app if you provided them with a start link, set the environment variable `OTREE_AUTH_LEVEL` to `EXPERIMENT`.

If you would like to put your site online in public demo mode where anybody can play a demo version of your game, set `OTREE_AUTH_LEVEL` to `DEMO`. This will allow people to play in demo mode, but not access the full admin interface.

7.17.2 Start links

There are multiple types of start links you can use. The optimal one depends on how you are distributing the links to your users.

Single-use links

When you create a session, oTree creates 1 start link per participant, each of which contains a unique code for the participant.

Session-wide link

If it is impractical to distribute distinct URLs to each participant, you can provide the same start link to all participants in the session. Note: this may result in the same participant playing twice, unless you use the `participant_label` parameter in the URL (see [Participant labels](#)).

Server-wide (persistent) link

You can create persistent links that will stay constant for new sessions, even if the database is recreated.

This is useful in the following situations:

- You are running multiple lab sessions, and cannot easily distribute new links to the workstations each time you create a session.
- You are running multiple sessions online with the same group of participants, and want each participant to use the same link each time they participate in one of your sessions.

To obtain these persistent links, set a session as the default session, or visit the “Persistent URLs” page in the admin.

7.17.3 Participant labels

You can append a `participant_label` parameter to each participant’s start URL to identify them, e.g. by name, ID number, or computer workstation.

Each time a start URL is accessed, oTree checks for the presence of a `participant_label` parameter and records it for that participant. This label will be displayed in places where participants are listed, like the oTree admin interface or the payments page.

7.17.4 Grouping and randomization

If participants are not using single-use links (see *Single-use links*), oTree will assign the first person who arrives to be P1, the second to be P2, etc. If you would instead like participant selection to be random, you can set `'random_start_order': True`, in the session config dictionary (or `SESSION_CONFIG_DEFAULTS`).

Note that if you use single-use links, then `random_start_order` will have no effect, because each single-use link is tied to a specific participant (the URL contains the participant’s unique code).

7.17.5 Online experiments

Experiments can be launched to participants playing over the internet, in a similar way to how experiments are launched the lab. Login to the admin, create a session, then distribute the links to participants via email or a website.

7.17.6 Kiosk Mode

On your lab’s devices, you can enable “kiosk mode”, a setting available in most web browsers, to prevent participants from doing things like accessing the browser’s address bar, hitting the “back” button, or closing the browser window.

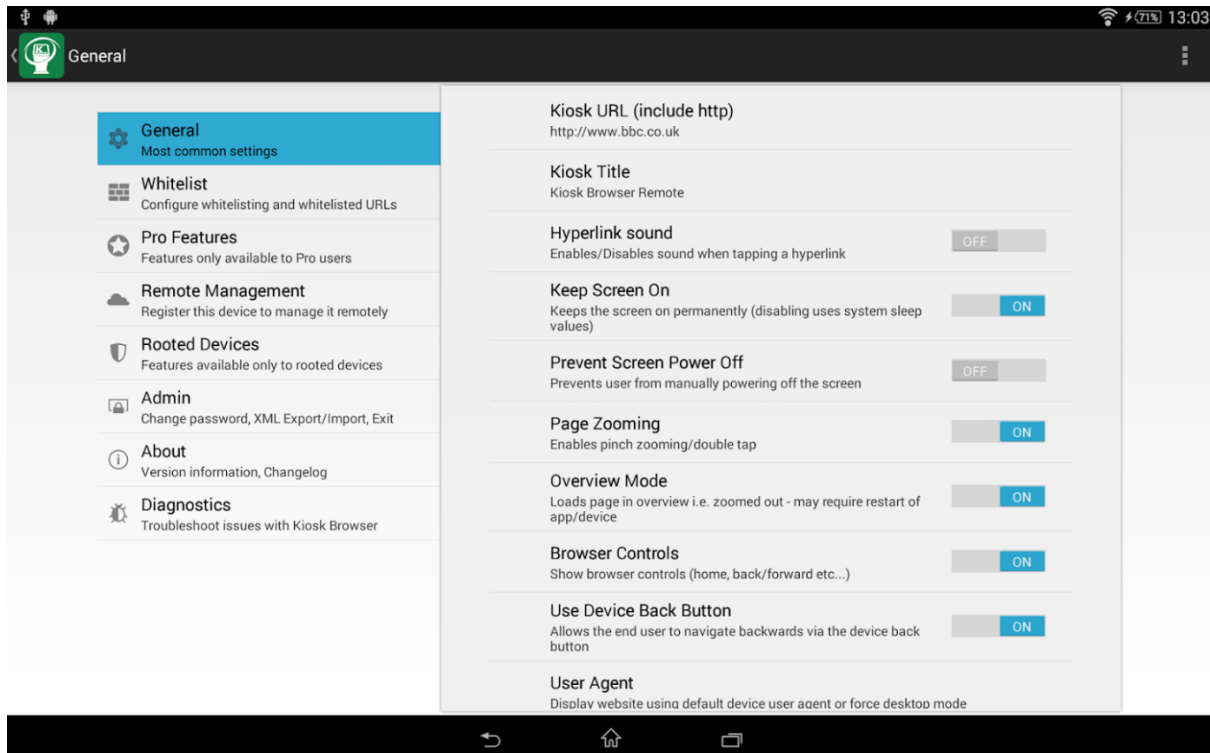
Below are some guidelines on how to enable Kiosk mode.

iOS (iPhone/iPad)

1. Go to Setting – Accessibility – Guided Access
2. Turn on Guided Access and set a passcode for your Kiosk mode
3. Open your web browser and enter your URL
4. Triple-click home button to initiate Kiosk mode
5. Circle areas on the screen to disable (e.g. URL bar) and activate

Android

There are several apps for using Kiosk mode on Android, for instance: [Kiosk Browser Lockdown](#).



oTree comes with an admin interface, so that experimenters can manage sessions, monitor the progress of live sessions, and export data after sessions.

Open your browser to the root url of your web application. If you're developing locally, this will be <http://127.0.0.1:8000/>.

Chrome on PC

1. Go to Setting – Users – Add new user
2. Create a new user with a desktop shortcut
3. Right-click the shortcut and select “Properties”
4. In the “Target” field, add to the end either `--kiosk "http://www.your-otree-server.com"` or `--chrome-frame --kiosk "http://www.your-otree-server.com"`
5. Disable hotkeys (see [here](#))
6. Open the shortcut to activate Kiosk mode

IE on PC

IE on PC See [here](#)

Mac

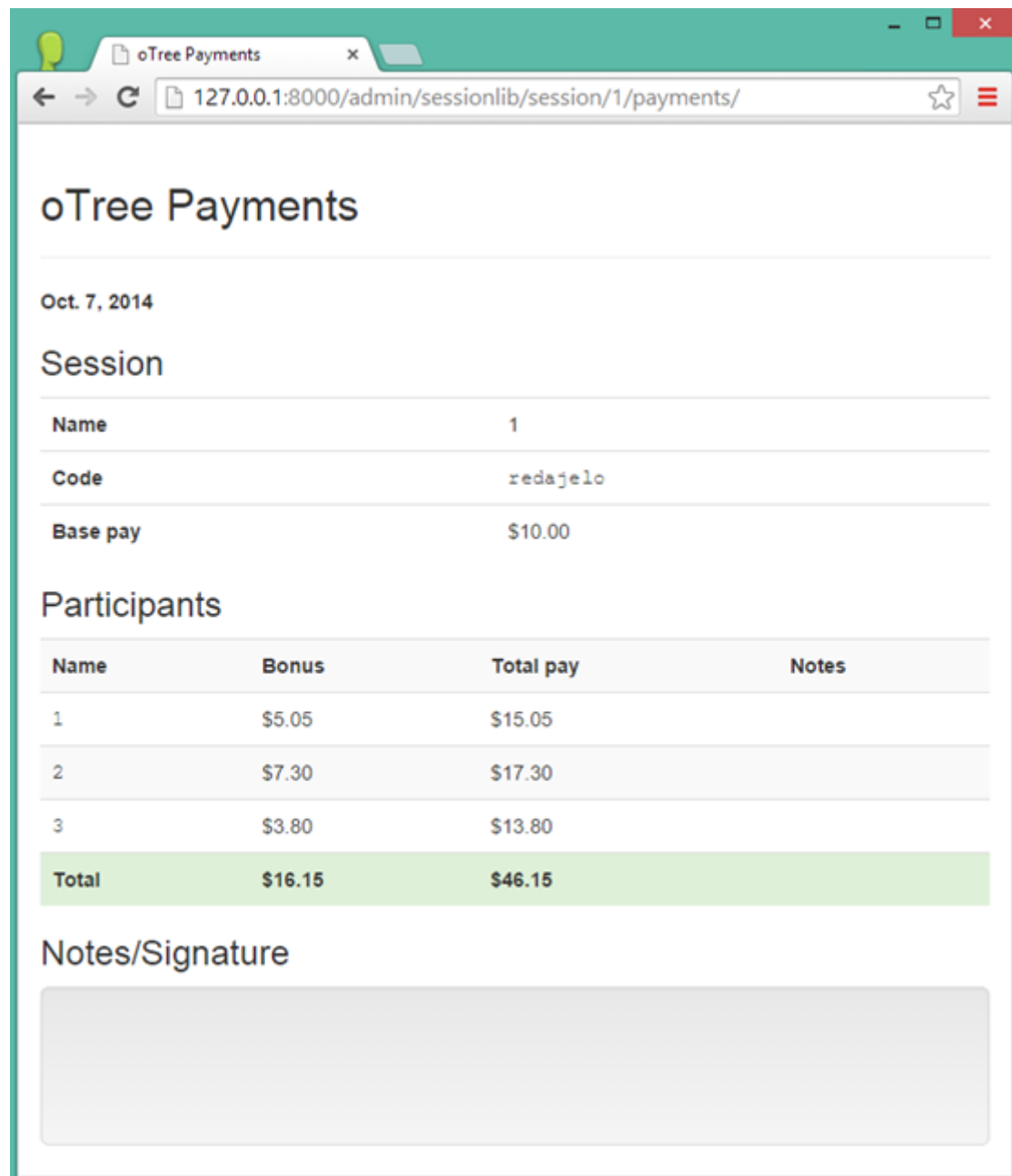
There are several apps for using Kiosk mode on Mac, for instance: [eCrisper](#). Mac keyboard shortcuts should be disabled.

7.17.7 Monitor sessions

While your session is ongoing, you can monitor the live progress in the admin interface. The admin tables update live, highlighting changes as they occur.

7.17.8 Payment PDF

At the end of your session, you can open and print a page that lists all the participants and how much they should be paid.



The screenshot shows a web browser window with the title 'oTree Payments'. The address bar shows the URL '127.0.0.1:8000/admin/sessionlib/session/1/payments/'. The page content includes a header 'oTree Payments', a date 'Oct. 7, 2014', and a section titled 'Session'. Below this is a table with session details: Name (1), Code (redajelo), and Base pay (\$10.00). Another section titled 'Participants' contains a table with columns: Name, Bonus, Total pay, and Notes. The table lists three participants with their respective bonus and total pay amounts. A green row at the bottom shows the 'Total' for all participants. At the bottom of the page is a section titled 'Notes/Signature' with a large text area.

Name	Bonus	Total pay	Notes
1	\$5.05	\$15.05	
2	\$7.30	\$17.30	
3	\$3.80	\$13.80	
Total	\$16.15	\$46.15	

7.17.9 Export Data

You can download your raw data in text format (CSV) so that you can view and analyze it with a program like Excel, Stata, or R.

7.17.10 Autogenerated documentation

Each model field you define can also have a `doc=` argument. Any string you add here will be included in the autogenerated documentation file, which can be downloaded through the data export page in the admin.

7.17.11 Debug Info

When oTree runs in `DEBUG` mode (i.e. when the environment variable `OTREE_PRODUCTION` is not set), debug information is displayed on the bottom of all screens. The debug information consists of the ID in group, the group, the player, the participant label, and the session code. The session code and participant label are two randomly generated alphanumeric codes uniquely identifying the session and participant. The ID in group identifies the role of the player (e.g., in a principal-agent game, principals might have the ID in group 1, while agents have 2).

Debug info	
ID in group	2
Group	6
Player	10
Participant label	DESCIL-W02
Session code	kagageha

7.18 Server deployment

You can develop and test your app locally on your personal computer, using the ordinary `runserver` command.

However, when you want to share your app with an audience, you must deploy to a web server. oTree can be deployed to a cloud service like Heroku, or to your own on-premises server.

7.18.1 Heroku

If you are not experienced with web server administration, Heroku may be the simplest option for you. Instructions on how to deploy oTree to Heroku are [here](#).

Here are the steps for deploying to Heroku.

Create an account

Create a free account on [Heroku](#). You can skip the “Getting Started With Python” guide.

Install the Heroku Toolbelt

Install the [Heroku Toolbelt](#).

This provides you access to the Heroku Command Line utility.

Once installed, you can use the `heroku` command from your command shell.

From the oTree launcher, click the terminal button to access the command shell. Log in using the email address and password you used when creating your Heroku account:

```
$ heroku login
Enter your Heroku credentials.
Email: python@example.com
Password:
Authentication successful.
Authenticating is required to allow both the heroku and git commands to operate.
```

Create the Heroku app

Create an app on Heroku, which prepares Heroku to receive your source code:

```
$ heroku create
Creating lit-bastion-5032 in organization heroku... done, stack is cedar-14
http://lit-bastion-5032.herokuapp.com/ | https://git.heroku.com/lit-bastion-5032.git
Git remote heroku added
When you create an app, a git remote (called heroku) is also created and associated with your local repository.
```

Heroku generates a random name (in this case lit-bastion-5032) for your app, or you can pass a parameter to specify your own app name.

Deploy your code

Open the oTree shell (if using the launcher, click the “Terminal” button).

Make sure you have committed any changes as follows:

```
$ git add .
$ git commit -am '[commit message]'
```

(If you get the message fatal: Not a git repository (or any of the parent directories): .git then you first need to initialize the git repo.)

Then do:

```
$ git push heroku master
$ otree-heroku resetdb [your heroku app name]
```

Go to the [Heroku Dashboard](#), click on your app, click to edit the dynos, and make sure the “worker” dyno is turned on. (This will ensure that the page timeouts defined by `timeout_seconds` still work even if a user closes their browser.)



(If you do not see a “worker” entry, make sure your Procfile looks like [this](#).)

Now visit the app at the URL generated by its app name. As a handy shortcut, you can open the website as follows:

```
$ heroku open
```

Set environment variables

If it's a production website, you should set the environment variables (e.g. `OTREE_PRODUCTION` and `OTREE_AUTH_LEVEL`), like this:

```
$ heroku config:set OTREE_PRODUCTION=1
$ heroku config:set OTREE_AUTH_LEVEL=DEMO
```

To add an existing remote:

```
$ heroku git:remote -a [myherokuapp]
```

Scaling up the server

The Heroku free plan is sufficient for small-scale testing of your app, but once you are ready to go live, we recommend you upgrade your Postgres database to a paid tier (because the row limit of the free version is very low), and scale up your dynos to at least the cheapest paid plan. Note: after you finish your experiment, you can scale your dynos and database back down, so then you don't have to pay the full monthly cost.

Running otree commands on Heroku

(TODO)

7.18.2 Deploying to an on-premises server

Note: If you are just testing your app locally, you can use the `resetdb` and `runserver` commands, which are simpler than the below steps.

Although Heroku deployment may be the easiest option, you may prefer to run oTree on your own server. Reasons may include:

- You do not want your server to be accessed from the internet
- You will be launching your experiment in a setting where internet access is unavailable
- You want full control over how your server is configured

oTree runs on top of Django, so oTree setup is the same as Django setup. Django runs on a wide variety of servers, except getting it to run on a Windows server like IIS may require extra work; you can find info about Django + IIS online. Below, instructions are given for using Unix and Gunicorn.

Database

oTree is most frequently used with PostgreSQL as the production database, although you can also use MySQL, MariaDB, or any other database supported by Django.

You can create your database with a command like this:

```
$ psql -c 'create database django_db;' -U postgres
```

Then, you should set the following environment variable, so that it can be read by `django_db_url`:

```
DATABASE_URL=postgres://postgres@localhost/django_db
```

Then, instead of installing `requirements_base.txt`, install `requirements.txt`. This will install `psycopg2`, which is necessary for using Postgres.

You may get an error when you try installing `psycopg2`, as described [here](#).

The fix is to install the `libpq-dev` and `python-dev` packages. On Ubuntu/Debian, do:

```
sudo apt-get install libpq-dev python-dev
```

The command `otree resetdb` only works on SQLite. On Postgres, you should drop the database and then run `otree migrate`.

Running the server

If you are just testing your app locally, you can use the usual `runserver` command.

However, when you want to use oTree in production, you need to run the production server, which can handle more traffic. You should use a process control system like Supervisor, and have it launch otree with the command `otree runprodserver`.

This will run the `collectstatic` command, and then launch the server as specified in the `Procfile` in your project's root directory. The default `Procfile` launches the Gunicorn server. If you want to use another server like Nginx; you need to modify the `Procfile`. (If you instead want to use Apache, consult the Django docs.)

New in version 0.3.8: `runprodserver`

Warning: Gunicorn doesn't work on Windows, so if you are trying to run oTree on a Windows server or use `runprodserver` locally on your Windows PC, you will need to specify a different server in your `Procfile`.

7.19 Troubleshooting

7.19.1 Common errors

Python not installed

```
'python' is not recognized as an internal or external command, operable program or batch file.
```

Solution: make sure Python 2.7 is installed and add it to your Path.

TemplateEncodingError

If you get this error:

```
TemplateEncodingError: Templates can only be constructed from unicode or UTF-8 strings.
```

This is an old oTree bug; upgrade your version of otree-core (see [Upgrade oTree core libraries](#)).

otree: command not found

If you are using the launcher, click the “Terminal” button. This will ensure your terminal opens with the correct programs loaded. Also, if you are using a version of `otree-core` older than 0.3.20, you need to upgrade (see [Upgrade oTree core libraries](#)).

‘with’ in formfield tag needs at least one keyword argument

```
django.template.base.TemplateSyntaxError: 'with' in formfield tag needs at least one keyword argu
```

This is usually caused by a *formfield* tag with a space after *label*, e.g.:

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

You should remove the space around the = like this:

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

7.20 Tips and tricks

7.20.1 Don't make multiple copies of your app

If possible, you should avoid copying an app's folder to make a slightly different version, because then you have duplicated code that is harder to maintain.

If you need multiple rounds, set `num_rounds`. If you need slightly different versions (e.g. different treatments), then you should use the techniques described in *Treatments*, such as making 2 session configs that have a different `'treatment'` parameter, and then checking for `self.session.config['treatment']` in your app's code.

7.20.2 Don't modify values in Constants

As its name implies, `Constants` is for values that don't change – they are the same for all participants across all sessions. So, you shouldn't do something like this:

```
def my_method(self):
    Constants.my_list.append(1)
```

`Constants` has global scope, so when you do this, your modification will “leak” to all other sessions, until the server is restarted. Instead, if you want a variable that is the same for all players in your session, you should set a field on the subsession, or use *Global variables*.

For the same reason, you shouldn't assign to class attributes on your models. For example, don't do this:

```
class Player(BasePlayer):

    my_list = []

    def foo(self):
        self.my_list.append(1)
```

7.21 Amazon Mechanical Turk

7.21.1 Overview

oTree provides integration with Amazon Mechanical Turk (MTurk).

You can publish your game to MTurk directly from oTree's admin interface. Then, workers on mechanical Turk can accept and play your app as an MTurk HIT and get paid a participation fee as well as bonuses they earned by playing your game.

7.21.2 AWS credentials

To publish to MTurk, you must have an employer account with MTurk, which currently requires a U.S. address and bank account.

oTree requires that you set the following env vars:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`

You can obtain these credentials [here](#):

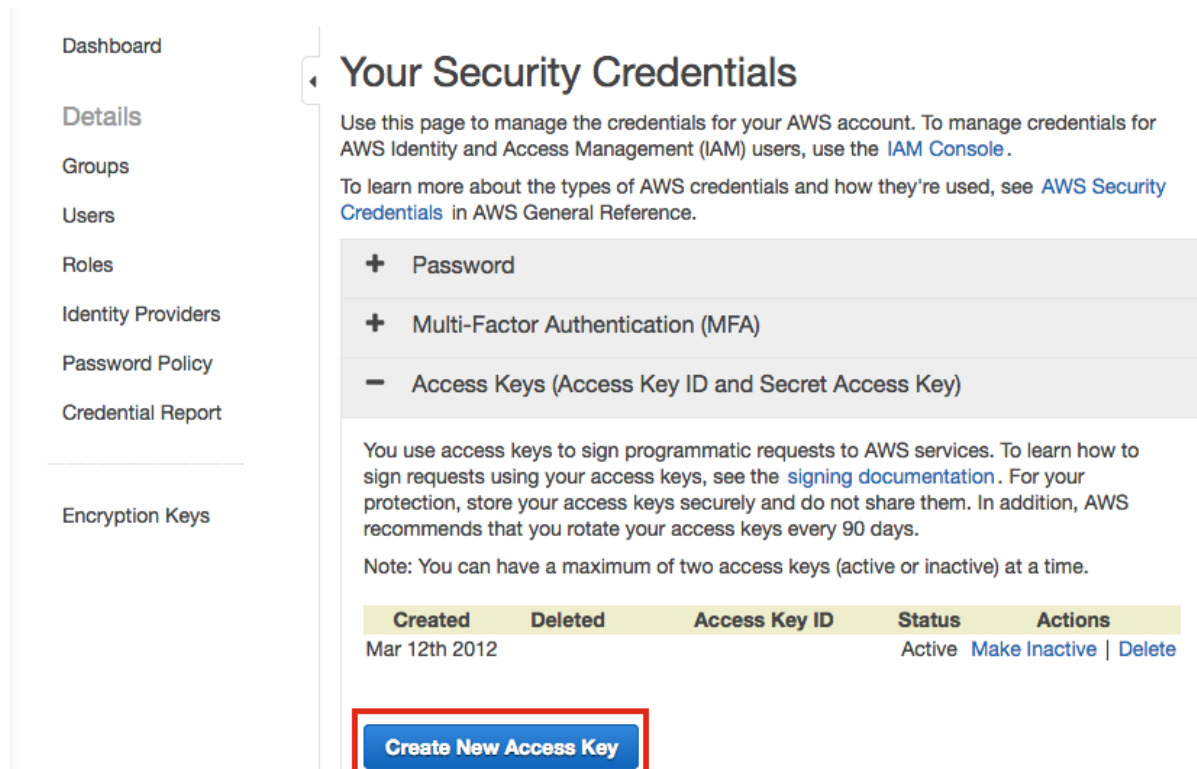


Fig. 7.1: AWS key

On Heroku you would set these env vars like this:

```
$ heroku config:set AWS_ACCESS_KEY_ID=YOUR_AWS_ACCESS_KEY_ID
$ heroku config:set AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_ACCESS_KEY
```

7.21.3 Making your session work on MTurk

You should look in `settings.py` for all settings related to Mechanical Turk (do a search for “mturk”). You can edit the properties of the HIT such as the title and keywords, as well as the qualifications required to participate. The monetary reward paid to workers is `self.session.config['participation_fee']`.

When you publish your HIT to MTurk, it will be visible to workers. When a worker clicks on the link to take part in the HIT, they will see the MTurk interface, with your app loaded inside a frame (as an `ExternalQuestion`). Initially, they will be in preview mode, and will see the `preview_template` you specify in `settings.py`. After they accept the HIT, they will see the first page of your session, and be able to play your session while it is embedded inside a frame in the MTurk worker interface.

The only modification you should make to your app for it to work on AMT is to add a `{% next_button %}` to the final page that your participants see. When the participant clicks this button, they will be directed back to the mechanical Turk website and their work will be submitted.

After workers have completed the session, you can click on the “payments” Tab for your session. Here, you will be able to approve submissions, and also pay the bonuses that workers earned in your game.

7.21.4 Testing your hit in sandbox

The Mechanical Turk Developer Sandbox is a simulated environment that lets you test your app prior to publication in the marketplace. This environment is available for both [worker](#) and [requester](#).

After publishing the HIT you can test it both as a worker and as a requester using the links provided on “MTurk” Tab of your session admin panel.

7.21.5 Preventing retakes (repeat workers)

To prevent a worker from participating in your study twice, you can grant a Qualification to each worker who participates in your study, and then prevent people who already have this qualification from participating in your studies.

This technique is described [here](#).

First, login to your MTurk requester account and create a qualification. Then, go to `settings.py` and paste the qualification’s ID into `grant_qualification_id`. Finally, add an entry to `qualification_requirements`:

```
'grant_qualification_id': 'YOUR_QUALIFICATION_ID_HERE',
'qualification_requirements': [
    qualification.LocaleRequirement("EqualTo", "US"),
    ...
    qualification.Requirement('YOUR_QUALIFICATION_ID_HERE', 'DoesNotExist')
]
```

7.21.6 Multiplayer games

Games that involve synchronous interaction between participants (i.e. wait pages) can be tricky on mechanical Turk.

You should set `timeout_seconds` on each page, so that the page will be auto-submitted if the participant drops out or does not complete the page in time. This way, players will not get stuck waiting for someone who dropped out.

Finally, you can consider a “lock-in” task. In other words, before your multiplayer game, you can have a sub-session that is a single-player app and takes some effort to complete. The idea is that a participant takes the effort to complete this initial task, they are less likely to drop out after that point. Then, the first few participants to finish the lock in task will be assigned to the same group in the next sub-session, which is the multiplayer game.

Warning: If you downloaded oTree prior to July 2, 2015, you need to update your oTree project. You should upgrade your MTurk settings in `settings.py` to the new format [here](#). See the variable `mturk_hit_settings`, which is included in `SESSION_CONFIG_DEFAULTS`. Then upgrade to the latest version of `otree-core`. Also, copy the `Procfile` over the version you have locally.

7.22 oTree programming For Django Devs

7.22.1 Intro to oTree for Django developers

TODO!

Differences between oTree and Django

TODO!

Models

- Field labels should go in the template formfield, rather than the model field's `verbose_name`.
- `null=True` and `default=None` are not necessary in your model field declarations; in oTree fields are null by default.
- On CharFields, `max_length` is not required.

7.23 oTree glossary for z-Tree programmers

For those familiar with z-Tree, here are some notes on the equivalents of various z-Tree concepts in oTree. This document just gives the names of the oTree feature; for full explanations of each concept, see the [reference documentation](#).

This list will expand over time. If you would like to request an item added to this list, or if you have a correction to make, please email chris@otree.org.

7.23.1 z-Tree & z-Leafs

oTree is web-based so it does not have an equivalent of z-Leafs. You run oTree on your server and then visit the site in the browser of the clients.

7.23.2 Treatments

In oTree, these are apps in `app_sequence` in `settings.py`.

7.23.3 Periods

In oTree, these are called “rounds”. You can set `num_rounds`, and get the current round number with `self.subsession.round_number`.

7.23.4 Stages

oTree calls these “pages”, and they are defined in `views.py`.

7.23.5 Waiting screens

In oTree, participants can move through pages and subsessions individually. Participants can be in different apps or rounds (i.e. treatments or periods) at the same time.

If you would like to restrict this independent movement, you can use oTree’s equivalent of “Wait for all...”, which is to insert a `WaitPage` at the desired place in the `page_sequence`.

7.23.6 Subjects

oTree calls these ‘players’ or ‘participants’. See the reference docs for the distinction between players and participants.

7.23.7 Participate=1

Each oTree page has an `is_displayed` method that returns True or False.

7.23.8 Timeout

In oTree, define a `timeout_seconds` on your `Page`. You can also optionally define `timeout_submission`.

7.23.9 Questionnaires

In oTree, questionnaires are not distinct from any other type of app. You program them the same way as a normal oTree app. See the “survey” app for an example.

7.23.10 Program evaluation

In z-Tree, programs are executed for each row in the current table, at the same time.

In oTree, code is executed individually as each participant progresses through the game.

For example, suppose you have this `Page`:

```
class MyPage(Page):

    def vars_for_template(self):
        return {'double_contribution':

    def before_next_page(self):
        self.player.foo = True
```

The code in `vars_for_template` and `before_next_page` is executed independently for a given participant when that participant enters and exits the page, respectively.

If you want code to be executed for all participants at the same time, it should go in `before_session_starts` or `after_all_players_arrive`.

7.23.11 Background programs

The closest equivalent is `before_session_starts`.

7.23.12 Tables

Subjects table

In z-Tree you define variables that go in the subjects table.

In oTree, you define the structure of your table by defining “fields” in `models.py`. Each field defines a column in the table, and has an associated data type (number, text, etc).

You can access all players like this:

```
self.subsession.get_players()
```

This returns a list of all players in the subsession. Each player has the same set of fields, so this structure is conceptually similar to a table.

oTree also has a “Group” object (essentially a “groups” table), where you can store data at the group level, if it is not specific to any one player but rather the same for all players in the group, like the total contribution by the group (e.g. `self.group.total_contribution`).

Globals table

`self.session.vars` can hold global variables.

Table functions

oTree does not have table functions. If you want to carry out calculations over the whole table, you should do so explicitly.

For example, in z-Tree:

```
S = sum(C)
```

In oTree you would do:

```
S = sum([p for p in self.subsession.get_players()])
```

find()

Use `group.get_players()` to get all players in the same group, and `subsession.get_players()` to get all players in the same subsession.

If you want to filter the list of players for all that meet a certain condition, e.g. all players in the subsession whose payoff is zero, you would do:

```
zero_payoff_players = [
    p for p in self.subsession.get_players() if p.payoff == 0]
```

Another way of writing this is:

```
zero_payoff_players = []
for p in self.subsession.get_players():
    if p.payoff == 0:
        zero_payoff_players.append(p)
```

You can also use `group.get_player_by_id()` and `group.get_player_by_role()`.

7.23.13 Groups

Set `players_per_group` to any number you desire. When you create your session, you will be prompted to choose a number of participants. oTree will then automatically divide these players into groups.

Calculations on the group

For example:

z-Tree:

```
sum( same( Group ), Contribution );
```

oTree:

```
sum([p.contribution for p in self.group.get_players()])
```

Player types

In z-Tree you set variables like:

```
PROPOSERTYPE = 1;
RESPONDERTYPE = 2;
```

And then depending on the subject you assign something like:

```
Type = PROPOSERTYPE
```

In oTree you can determine the player’s type based on the automatically assigned field `player.id_in_group`, which is unique within the group (ranges from 1...N in an N-player group).

Additionally, you can define the method `role()` on the player:

```
def role(self):
    if self.id_in_group == 1:
        return 'proposer'
    else:
        return 'responder'
```

7.23.14 Accessing data from previous periods and treatments

See the reference on `in_all_rounds`, `in_previous_rounds` and `participant.vars`.

7.23.15 History box

You can program a history box to your liking using `in_all_rounds`. For example:

```
<table class="table">
  <tr>
    <th>Round</th>
    <th>Player and outcome</th>
    <th>Points</th>
  </tr>
  {% for p in player.in_all_rounds %}
    <tr>
      <td>{{ p.subsession.round_number }}</td>
      <td>
        You were {{ p.role }} and
        {% if p.is_winner %} won {% else %} lost {% endif %}
      </td>
      <td>{{ p.payoff }}</td>
    </tr>
  {% endfor %}
</table>
```

7.23.16 Parameters table

Any parameters that are constant within an app should be defined in `Constants` in `models.py`. Some parameters are defined in `settings.py`.

Define a method in `before_session_starts` that loops through all players in the subsession and sets values for the fields.

7.23.17 Clients table

In the admin interface, when you run a session you can click on “Monitor”. This is similar to the z-Tree Clients table.

There is a button “Advance slowest participant(s)”, which is similar to z-Tree’s “Leave stage” command.

7.23.18 Money and currency

- `ShowUpFee: session.config['participation_fee']`

- Profit: `player.payoff`
- FinalProfit: `participant.payoff`
- MoneyToPay: `participant.money_to_pay()`

Experimental currency units (ECU)

The oTree equivalent of ECU is points, and the exchange rate is defined by `real_world_currency_per_point`.

In oTree you also have the option to not use ECU and to instead play the game in real money.

7.23.19 Layout

Data display and input

In the HTML template, you output the current player's contribution like this:

```
{{ player.contribution }}
```

If you need the player to input their contribution, you do it like this:

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

Layout: !text

In z-Tree you would do this:

```
<>Your income is < Profit | !text: 0="small"; 80 = "large";>.
```

In oTree you can use `vars_for_template`, for example:

```
def vars_for_template(self):
    if self.player.payoff > 40:
        size = 'large'
    else:
        size = 'small'
    return {'size': size}
```

Then in the template do:

```
Your income is {{ size }}.
```

Another way to accomplish this is the `get_FOO_display`, which is described in the reference with the example about `get_year_in_school_display`.

7.23.20 Miscellaneous code examples

Get the other player's choice in a 2-person game

z-Tree:

```
OthersChoice = find( same( Group ) & not( same( Subject ) ), Choice );
```

oTree:

```
others_choice = self.get_others_in_group()[0].choice
```

Check if a list is sorted smallest to largest

z-Tree (source: z-Tree mailing list):

```
iterator(i, 10).sum( iterator(j, 10).count( :i<j & ::values[ :i ] > ::values[ j ] )) == 0
```

oTree:

```
values == sorted(values)
```

Randomly shuffle a list

z-Tree (source: z-Tree mailing list):

```
iterator(i, size_array - 1).do {
  address = roundup( random() * (:size_array + 1 - i), 1);
  if (address != :size_array + 1 - i) {
    temp = :random_sequence[:size_array + 1 - i];
    :random_sequence[:size_array + 1 - i] = :random_sequence[address];
    :random_sequence[address] = temp;
  }
}
```

oTree:

```
random.shuffle(random_sequence)
```

Choose 3 random periods for payment

z-Tree: see [here](#):

oTree:

```
if self.subsession.round_number == Constants.num_rounds:
    random_players = random.sample(self.in_all_rounds(), 3)
    self.payoff = sum([p.potential_payoff for p in random_players])
```

7.24 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)