All information about otree can be found here: https://otree.readthedocs.io/en/latest/index.html (https://otree.readthedocs.io/en/latest/index.html) Otree also have a Studio (a GUI interface to help you quickly set up an experiment with minimal coding requirements), it is suitable for a small to medium-sized project: https://otree.readthedocs.io/en/latest/studio.html (https://otree.readthedocs.io/en/latest/studio.html)

For installation, please follow this tutorial on Otree: https://otree.readthedocs.io/en/latest/install-windows.html#install-windows (https://otree.readthedocs.io/en/latest/install-windows.html#install-windows)

Important Note: Although frequently updating oTree is recommended, you should not do so during the mid of your project. After completing a project (that works), make a backup before updating. The same applies for Python version. It is best to make a comment on top of your project noting down which Python and Otree version was used. This also helps others who look at your file later on.

For this lecture, I am using Python 3.7 and the newest Otree version 3.3.11. I will also use Pycharm instead of Otree Studio to create online experiments.

We'll briefly discuss about Django or other elements of web development where it is necessary. These topics are very huge in itself.

One more thing before we continue:

- *Blue italic lines* mean you have to do something.
- Black normal lines mean explanation (mostly)

Terminology

You can find detailed explanation here: https://otree.readthedocs.io/en/latest/conceptual_overview.html#conceptual-overview (https://otree.readthedocs.io/en/latest/conceptual_overview.html#conceptual-overview). This part is provided for those who are unfamiliar with experimental economics.

Session: In oTree, a session is an event during which multiple participants take part in a series of tasks or games. You can think of it as an appointment on a given day (similar to a doctor's visit)

Subsession: A session is a series of subsessions; subsessions are the “sections” or “modules” that constitute a session. Each subsession can contain multiple pages. If a game repeated for multiple round, each of these rounds is a subsession.

Groups: Each subsession can be further divided into groups of players; for example, you could have a subsession with 30 players, divided into 15 groups of 2 players each. (Note: groups can be shuffled between subsessions.)

To sum up:

A session is a series of subsessions

A subsession contains multiple groups

A group contains multiple players

Each player proceeds through multiple pages

Now let us starts with the first simple project: Creating Questionnaire

Project 1: Simple questionnaire

When starting a new project, you should write down its functionality, its appearance and user interface. This will help your project on track (like a business plan) - this is also called "Project's Spec". Normally, if you are planning your thesis or term paper, this step is already done as you already have a clear picture of what you want to do.

Project 1 spec: We ask the participant for their information (i.e Name, Gender, Major), then on the next page, display this info back to them.

From this spec, we can see that our small app will not have multiple subsessions or groups. The participants just come to the experiment, complete the questionnaire and leave. What about pages? Let assumes we have a long questionnaire to fill, and the participant has to complete multiple pages (2 pages).

Step 1: Starting a project

To create a project make sure you are inside the folder where settings.py is. Then SHIFT + RIGHT CLICK on the empty area inside the folder and select powershell, then type:

```
In [ ]: otree startapp <name_of_project> #I named it proj_simple_survey
```

Inside the newly created folder "proj_simple_survey", open and edit models.py

In Otree, there are 3 datatype models: Player, Group, Subsession. You can think of a model as a table (in which you declare each column to store what information)

This is quite limited, however, it should cover enough for most uses, you can also create your own model if the experiment gets complex. However, to keep everything simple, I will only use these 3 models. If you want a custom model, please check this guide here: https://datascience.blog.wzb.eu/2016/10/31/using-custom-data-models-in-otree/ (https://datascience.blog.wzb.eu/2016/10/31/using-custom-data-models-in-otree/)

Step 2: Declaring models

Open models.py and scroll to the line that says class Player(BasePlayer):. Let's add 3 columns for Name, Gender and Major.

name: is a StringField
gender: is a Boolean field, you can also make it StringField, this is not absolute)
major: is a StringField

There are only a few Field types in Otree:

BooleanField (for true/false and yes/no values)

CurrencyField for currency amounts; (remember our guest lecture's talk? turns out it is already handled by oTree)

IntegerField

FloatField (for real numbers)

StringField (for text strings)

LongStringField (for long text strings; its form widget is a multi-line text area)

More about formfield here: https://otree.readthedocs.io/en/latest/forms.html (https://otree.readthedocs.io/en/latest/forms.html)

In [ ]:
```
class Player(BasePlayer):
    name = models.StringField()
    gender = models.BooleanField(choices=[
        [False, 'Male'],
        [True, 'Female'],
    ])
    major = models.StringField()
```

Next, we need to create pages using HTML (for those of you who have no idea what is HTML please have a 3-min read here: https://simple.wikipedia.org/wiki/HTML (https://simple.wikipedia.org/wiki/HTML)). When it comes to HTML, you really don't have to fear it. It is NOT programming, but just a way to semantically mark sections of a document that will be displayed on the website (somewhat similar to LaTex).

In the same folder as models.py, you should see a folder named "templates", inside this is another folder with your project name: "proj_simple_survey"
You can put HTML templates inside this folder, if you find some interesting webpages that you would like to copy, you can also copy them and put it in here.

Step 3: Making templates

Create a file named "MyPage.html" then use PyCharm to open it. Type in the below code.
Note: You are likely to see the file "MyPage.html" has been automatically created for you when you start your project, you can just use it instead of creating a new file.

In [ ]:
```
{% extends "global/Page.html" %}
{% load otree %}

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

{% block content %}

    Please enter the following information.

    {% formfields %}

    {% next_button %}

{% endblock %}
```

In [ ]:
```
Note that these are HTML file, not Python, thus for comments instead of #, you
need <!-- COMMENT HERE -->
```

Create a file named "Results.html" then use PyCharm to open it. Type in the below code.
The second template will be called Results.html.

```
In [ ]: {% extends "global/Page.html" %}
        {% load otree %}

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

        {% block content %}

            <p>Your name is {{ player.name }} and your major is {{ player.major }}. Yo
        u are
                {% if player.gender %} female
                {% else %} male
                {% endif %}.</p>

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

If you have read about HTML, you will find Django is somewhat similar as you will normally work with tags and templates. You can always refer to their page here for more details about specific templates (i.e. template inheritance, built-in templates, .etc) https://docs.djangoproject.com/en/dev/ (https://docs.djangoproject.com/en/dev/)

Step 4: Putting templates to work

These .html template files on its own will not do anything if you don't hook it up to your backend. Thus, we need to declare some classes in pages.py. Since we have 2 templates, we need 2 classes. The name must be matched with your templates name (MyPage and Results)

What to declare in these classes? Normally, these classes should declare input that you are expecting from the participant (forms that they have to fill, multiple-choice answer, .etc).

Thus, since in MyPage, we are expecting the player to answer their name, age, and gender. We should declare it in our class.

Open pages.py in oTree\proj_simple_survey folder. Type in the following code:

```
In [ ]: class MyPage(Page):
            form_model = 'player' #remember we only have 3 models, in this case it's a
        player (obviously)
            form_fields = ['name', 'major','gender']
```

Since on the Results page, we only display the information, we don't have to declare anything. However, you still need to make a placeholder for the class (it is a good practice to do so)

```
In [ ]: class Results(Page):
            pass #pass means do nothing or nothing to say/declare
```

You probably will see a class regarding to WaitPage in pages.py. Please delete it, we will come back to this when we have multiplayer games. Set your page_sequence to MyPage followed by Results.
The final result of pages.py should look like this:

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


class MyPage(Page):
    form_model = 'player'
    form_fields = ['name', 'major', 'gender']


class Results(Page):
    pass


page_sequence = [
    MyPage,
    Results
]
```

Step 5: Finalizing your session

Then in settings.py in the top folder (normally one level above the project folder), you will find the settings.py.
Open the settings.py file and edit SESSION_CONFIGS as follow:

In [ ]:
```
SESSION_CONFIGS = [
    dict(
        name='proj_simple_survey',
        num_demo_participants=3,
        app_sequence=['proj_simple_survey']
    ),
]
```

The SESSION_CONFIGS, as the name suggested, is used to configure a session. One session information is put as a dictionary with several keys such as
- name (name of the session)
- num_demo_participants (number of participants for testing)
- finally app_sequence (which may comprises of multiple apps). This app_sequence is the most important factor, usually a session comprises of a pre-questionnaire, then the game, and finally a post-questionnaire.

If you didn't delete the old "SESSION_CONFIGS" you can see that you can put multiple session configs here (they are kept separated). We will talk about passing data between subsessions later on, however, passing data between sessions are usually done in the final analysis (during the data analysis process where we match data based on StudentID for example). If you have a session right after another session, usually there is no reason to separate them into 2 different sessions. Thus, Otree does not easily support passing data between sessions.

Overall view of files you have worked with
You can see from this simple project the minimum number of files you have to work with are: *models.py, pages.py, settings.py* and *html templates* (mypage.html, results.html)

The final step is to run the app and test it on your local developement server. *In the folder of settings.py, open powershell and run*

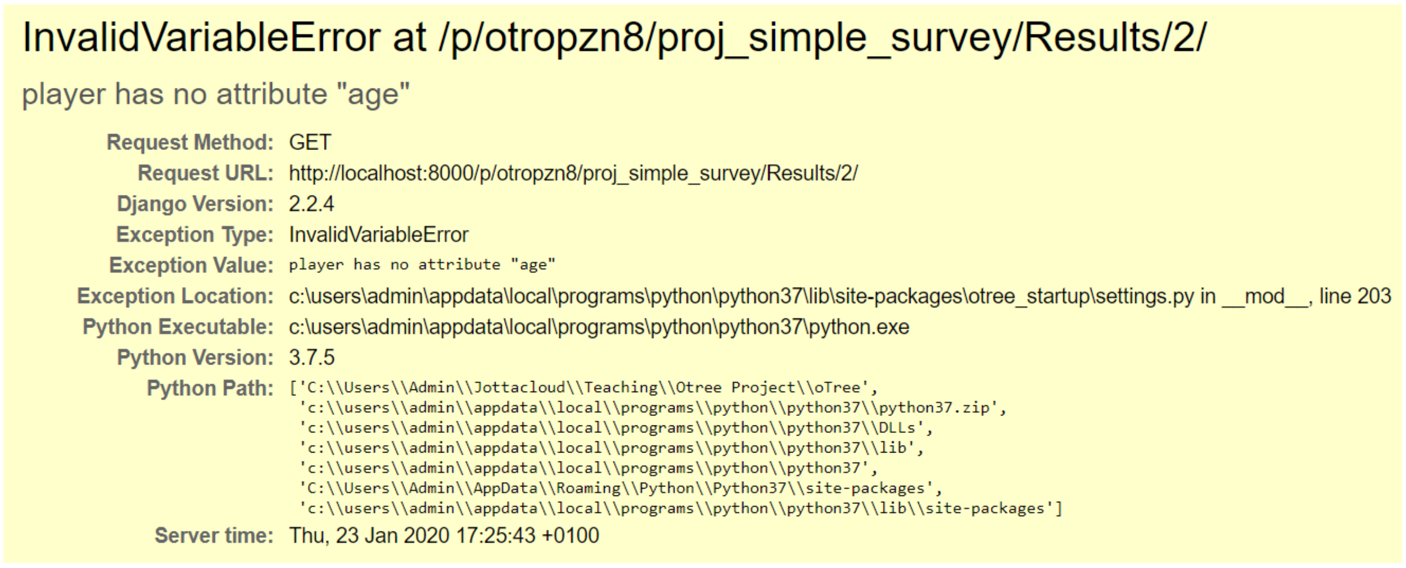
```
In [ ]: otree devserver
```

The result should look like this:
....
INFO Watching for file changes with StatReloader
Open your browser to http://localhost:8000/ (http://localhost:8000/)
To quit the server, press Control+C.

Copy and paste the link on your web browser.

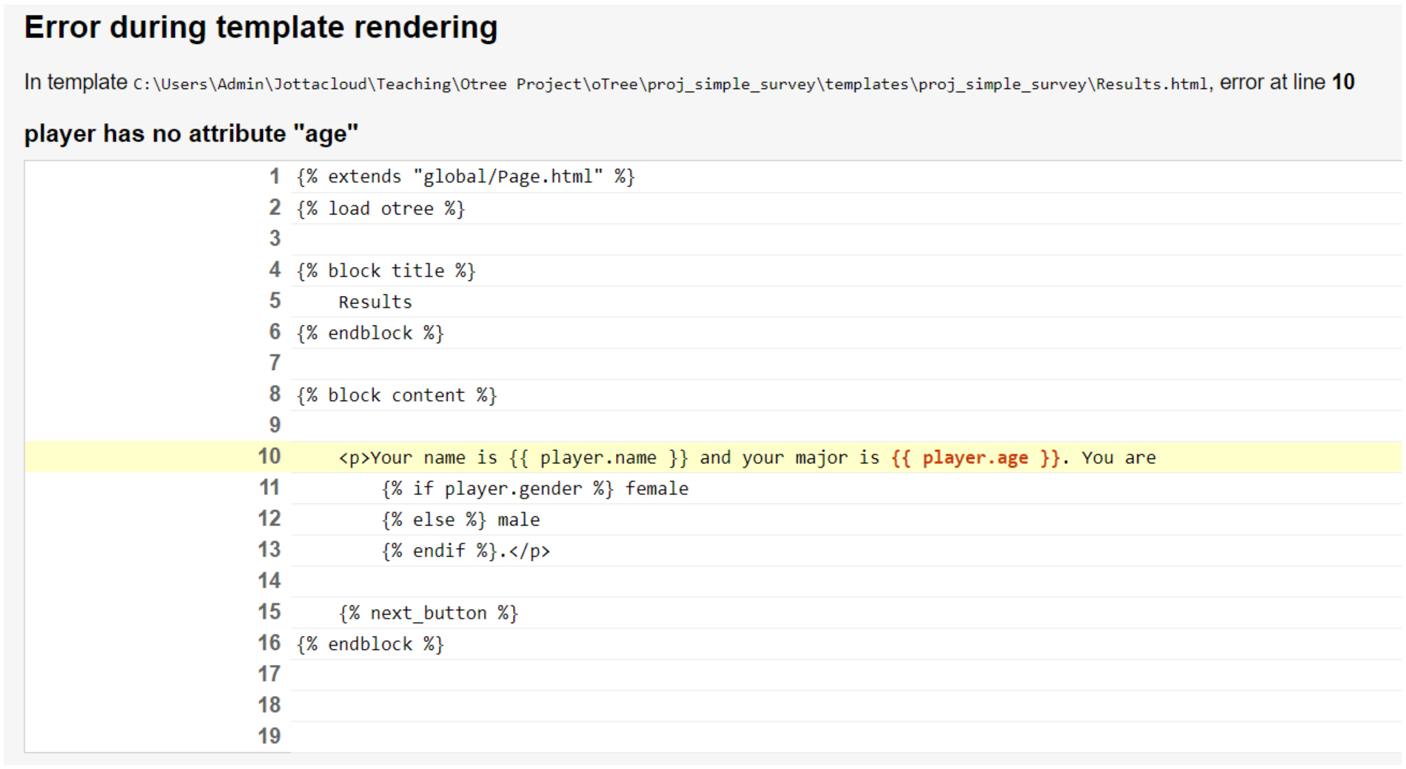
Fixing errors

During your time working with your oTree project, you'll likely encounter errors (it is almost inevitable even for professional developers). Here are some breakdown of the error message, you'll see something like this:



```
InvalidVariableError at /p/otropzn8/proj_simple_survey/Results/2/
player has no attribute "age"

       Request Method: GET
          Request URL: http://localhost:8000/p/otropzn8/proj_simple_survey/Results/2/
       Django Version: 2.2.4
       Exception Type: InvalidVariableError
      Exception Value: player has no attribute "age"
   Exception Location: c:\users\admin\appdata\local\programs\python\python37\lib\site-packages\otree_startup\settings.py in __mod__, line 203
    Python Executable: c:\users\admin\appdata\local\programs\python\python37\python.exe
       Python Version: 3.7.5
          Python Path: ['C:\\Users\\Admin\\Jottacloud\\Teaching\\Otree Project\\oTree',
                        'c:\\users\\admin\\appdata\\local\\programs\\python\\python37\\python37.zip',
                        'c:\\users\\admin\\appdata\\local\\programs\\python\\python37\\DLLs',
                        'c:\\users\\admin\\appdata\\local\\programs\\python\\python37\\lib',
                        'c:\\users\\admin\\appdata\\local\\programs\\python\\python37',
                        'C:\\Users\\Admin\\AppData\\Roaming\\Python\\Python37\\site-packages',
                        'c:\\users\\admin\\appdata\\local\\programs\\python\\python37\\lib\\site-packages']
          Server time: Thu, 23 Jan 2020 17:25:43 +0100
```



```
Error during template rendering

In template C:\Users\Admin\Jottacloud\Teaching\Otree Project\oTree\proj_simple_survey\templates\proj_simple_survey\Results.html, error at line 10

player has no attribute "age"

 1 {% extends "global/Page.html" %}
 2 {% load otree %}
 3
 4 {% block title %}
 5     Results
 6 {% endblock %}
 7
 8 {% block content %}
 9
10     <p>Your name is {{ player.name }} and your major is {{ player.age }}. You are
11         {% if player.gender %} female
12         {% else %} male
13         {% endif %}.</p>
14
15     {% next_button %}
16 {% endblock %}
17
18
19
```

In this case, since the debug mode is on, the error message is quite clear, I am using a variable that is not declared in Player class in models.py.
In some cases, errors are much harder to spot. You have to look at the traceback:

In [ ]:
```
Environment:


Request Method: GET
Request URL: http://localhost:8000/p/otropzn8/proj_simple_survey/Results/2/

Django Version: 2.2.4
Python Version: 3.7.5
Installed Applications:
['otree',
 'django.contrib.auth',
 'django.forms',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'channels',
 'huey.contrib.djhuey',
 'idmap',
 'proj_simple_survey']
Installed Middleware:
['otree.middleware.CheckDBMiddleware',
 'otree.middleware.perf_middleware',
 'whitenoise.middleware.WhiteNoiseMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.common.CommonMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',
 'django.contrib.messages.middleware.MessageMiddleware']


Template error:
In template C:\Users\Admin\Jottacloud\Teaching\Otree Project\oTree\proj_simple
_survey\templates\proj_simple_survey\Results.html, error at line 10
   player has no attribute "age"
   1 : {% extends "global/Page.html" %}
   2 : {% load otree %}
   3 :
   4 : {% block title %}
   5 :     Results
   6 : {% endblock %}
   7 :
   8 : {% block content %}
   9 :
   10 :     <p>Your name is {{ player.name }} and your major is  {{ player.age
}} . You are
   11 :         {% if player.gender %} female
   12 :         {% else %} male
   13 :         {% endif %}.</p>
   14 :
   15 :     {% next_button %}
   16 : {% endblock %}
   17 :
   18 :
   19 :
```

```
Traceback:

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _resolve_lookup
  829.                           current = current[bit]

During handling of the above exception ('Player' object is not subscriptable),
another exception occurred:

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _resolve_lookup
  837.                               current = getattr(current, bit)

During handling of the above exception ('Player' object has no attribute 'age'
), another exception occurred:

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _resolve_lookup
  843.                                   current = current[int(bit)]

During handling of the above exception (invalid literal for int() with base 10
: 'age'), another exception occurred:

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in resolve
  671.                       obj = self.var.resolve(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in resolve
  796.                   value = self._resolve_lookup(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _resolve_lookup
  850.                                                           (bit, current))
# missing attribute

During handling of the above exception (Failed lookup for key [age] in <Player
1>), another exception occurred:

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\views\abstract.py" in dispatch
  296.                           response.render()

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\response.py" in render
  106.                   self.content = self.rendered_content

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\response.py" in rendered_content
  83.            content = template.render(context, self._request)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\backends\django.py" in render
  61.                  return self.template.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
```

```
  171.                     return self._render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _render
  163.         return self.nodelist.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  937.                 bit = node.render_annotated(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in render_annotated
  103.             return self.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\loader_tags.py" in render
  150.             return compiled_parent._render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _render
  163.         return self.nodelist.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  937.                 bit = node.render_annotated(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in render_annotated
  103.             return self.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\loader_tags.py" in render
  150.             return compiled_parent._render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _render
  163.         return self.nodelist.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  937.                 bit = node.render_annotated(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in render_annotated
  103.             return self.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\loader_tags.py" in render
  150.             return compiled_parent._render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in _render
  163.         return self.nodelist.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  937.                 bit = node.render_annotated(context)
```

```
File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in render_annotated
  103.                return self.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\loader_tags.py" in render
  62.                    result = block.nodelist.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  937.                    bit = node.render_annotated(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in render_annotated
  103.                return self.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\loader_tags.py" in render
  62.                    result = block.nodelist.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  937.                    bit = node.render_annotated(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in render_annotated
  103.                return self.render(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in render
  987.                output = self.filter_expression.resolve(context)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree\strict_templates.py" in resolve
  41.                return original_resolve(self, context, ignore_failures=False
)

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\django\template\base.py" in resolve
  679.                                return string_if_invalid % self.var

File "c:\users\admin\appdata\local\programs\python\python37\lib\site-packages
\otree_startup\settings.py" in __mod__
  203.            raise InvalidVariableError(msg) from None

Exception Type: InvalidVariableError at /p/otropzn8/proj_simple_survey/Results
/2/
Exception Value: player has no attribute "age"
```

What to look for in this traceback is your file, if you see any files that you have been working on in this traceback, try to fix it first.
Sometimes you will find some files that are part of an external package (but no traces of any errors caused by your code). You can ask for help by putting this traceback online.
Please don't just write comments, developers need your traceback and snapshot to diagnose.

Traceback only covers syntax error, again logical errors are the hardest to spot, following coding convention will at least mitigate logical errors.

Project 2: Public Goods Game

Before we start here is the description of the classic public goods game, this is also our project's spec:

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

I start the app with:

```
In [ ]: otree startapp proj_public_goods
```

Step 1: Declare some constants

Before we go further, we should declare some constants. Unlike the previous simple project, this game requires a group, which is a piece of information that is unlikely to change, hence, it will be put in Constants.

Constants are the recommended place to put your app’s parameters and constants that do not vary from player to player.

These constants are actually class attributes of the Constants class. In models.py, under class Constants(BaseConstants), type:

```
In [ ]: players_per_group = 3
        num_rounds = 1
        ENDOWMENT = c(1000) # c() means it's a currency
        MULTIPLIER = 2
```

Also in the previous lecture, I mentioned the Python convention that all constant variables should be capitalized. You can (and should) follow this convention. However, note that there are some built-in constants in Otree, such as players_per_group, num_rounds, and name_in_url. These should not be named differently!

Constants can be numbers, strings, booleans, lists, etc. But for more complex data types like dicts, lists of dicts, etc, you should instead define it in a subsession method. For example, instead of defining a Constant called my_dict, do this:

```
In [ ]: class Subsession(BaseSubsession):
            def my_dict(self):
                return dict(a=[1,2], b=[3,4])
```

Step 2: Declare models

As explained previously, you should declare columns in the data table of each player, basically all the information you want to collect from the player. *In models.py under class Player(BasePlayer), type:*

```
In [ ]: class Player(BasePlayer):
            contribution = models.CurrencyField(
                min=0,
                max=Constants.ENDOWMENT,
                label="How much will you contribute?"
            )
```

You might be also thinking of payoff for the participants, however, this is by default already included in BasePlayer class, which Player inherited from

Next, we would need to declare columns for group data. In this case, it is the total amount of contribution of a group (comprises of 3 players).
In models.py under class Group(BaseGroup), type:

```
In [ ]: class Group(BaseGroup):
            total_contribution = models.CurrencyField()
            individual_share = models.CurrencyField()
```

Keep in mind that these variable names will be the column names or more specifically the variable names that you will work with during your data analysis process later on. Thus, mind the naming convention or the naming convention of your workplace (companies, universities). Of course, changing column names in Pandas is possible, it is just more convenient if the names are consistent.

We also want to calculate the group total contribution and the result after multiply it to the MULTIPLIER, and divided by number of players in group.
In models.py under class Group(BaseGroup), type:

```
In [5]: def set_payoffs(self):
            players = self.get_players()
            # get_players is a built-in methods to return a list of players in the ins
        tance of a class
            # (self here is the instance of the class Group)
            contributions = [p.contribution for p in players]  # list comprehension
            self.total_contribution = sum(contributions)
            self.individual_share = self.total_contribution * Constants.MULTIPLIER / C
        onstants.players_per_group
            for p in players:
                p.payoff = Constants.ENDOWMENT - p.contribution + self.individual_shar
        e
```

Step 3: Making templates

First page
In the first page, we would like to ask if the player to make a contribution. *I named the first page as Contribute.html, inside type the following codes:*

```
In [ ]: {% extends "global/Page.html" %}
        {% load otree static %}

        {% block title %}
            Any Title You Want <!-- Need to declare title here usually the same name a
        s the .html file you are working on -->
        {% endblock %}

        {% block content %}
        <p>
            This is a public goods game with
            {{ Constants.players_per_group }} players per group,
            an endowment of {{ Constants.ENDOWMENT }},
            and a multiplier of {{ Constants.MULTIPLIER }}.
        </p>

        {% formfields %}

            {% next_button %}

        {% endblock %}
```

When naming your page, it is recommended to have also a page number (only if your app is long) and a meaningful name, since it is a Python class afterall. We will sometimes use this class alot thus, making it a long name isn't recommended either (something like Contribute_p1 or P1_contribute seems reasonable).

Second page

After filling the form, the participant must wait for other members in his/her group so that we can calculate the result. Thus, a wait page is needed.

Wait page already has a default template, therefore we do not need to make one ourselves. You can change the wait page text and title by declaring it. *In pages.py, under class ResultsWaitPage(WaitPage):*

```
In [ ]: class ResultsWaitPage(WaitPage):
            title_text = "Custom title text"
            body_text = "Custom body text"
```

You should not, however, named a wait page as WaitPage, because there might be multiple wait pages depending on your experiment, this also clashes with the parent class WaitPage the developer has created. For this project, let's call it ResultsWaitPage.

Changing the appearances of the wait page and other pages' templates is possible, however, it will only be discussed briefly later on (since it's just styling but also it might require some javascript, HTML and CSS knowledge). It is outside the scope of this lecture to discuss in-depth about web pages styling.
Only in cases when you want to customize the appearance of the wait page that you make a template for it!

To activate the .set_payoffs() method of the class Group, you can use the special method after_all_players_arrive. With current Otree version, you can just make this a class attributes. *In pages.py.py, under class ResultsWaitPage(WaitPage):*

```
In [ ]: after_all_players_arrive = 'set_payoffs'
```

In the older version of Otree 2.3, you need to declare this as a method under the class ResultsWaitPage(WaitPage). This is still compatible with the current Otree version. You might find this easier to remember and understand:

```
In [ ]: def after_all_players_arrive(self):
            self.group.set_payoffs()
```

Third page
The final page will the the Results.html page. Similar to the first project, we just need to simply display the result.
In Results.html, type the following within the content block:

In [ ]:
```
<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>
```

Step 4: Putting templates to work

In pages.py, type:

In [ ]:
```
class Contribute(Page):
    form_model = 'player'
    form_fields = ['contribution']
```

Finally, in pages.py, under page_sequence, make sure:

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

Step 5: Finalizing your session

Similar to Project 1, we just need to put this on our devserver and test it

Open the settings.py file and edit SESSION_CONFIGS as follow:

In [21]:
```
SESSION_CONFIGS = [
    dict(
        name='proj_simple_survey',
        num_demo_participants=3,
        app_sequence=['proj_simple_survey']
    ),
    dict(
        name='proj_public_goods',
        num_demo_participants=3,
        app_sequence=['proj_public_goods']
    ),
]
```

Project 3: Trust Game

Here is the Trust Game's spec: 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.

As usual I start the app with:

In [ ]:
```
otree startapp proj_trust_game
```

Step 1: Declare some constants

In [ ]:
```
class Constants(BaseConstants):
    name_in_url = 'my_trust'
    players_per_group = 2
    num_rounds = 1

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

Step 2: Declare models

Then we add fields to the player and group. There are 2 critical data points to record: 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:

In [ ]:
```
class Player(BasePlayer):

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

However, the problem 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. (though you can technically enforce it and use another variable to indicate which player is it).

Nevertheless, it is easier to declare these fields in the class Group(), as there is only 1 P1 and 1 P2.

```
In [ ]: class Group(BaseGroup):

            sent_amount = models.CurrencyField(
                label="How much do you want to send to participant B?"
            )
            sent_back_amount = models.CurrencyField(
                label="How much do you want to send back?"
            )
```

Furthermore, here I want to give an example of other types of form field (so far we were just working with typed in text, number, and boolean selection). Other Form Fields can be found here: https://otree.readthedocs.io/en/latest/forms.html#dynamic-validation (https://otree.readthedocs.io/en/latest/forms.html#dynamic-validation)

Under class Group(BaseGroup): let's define the choices for sent_back_amount:

```
In [ ]: def sent_back_amount_choices(self):
            return currency_range(
                c(0),
                self.group.sent_amount * Constants.multiplication_factor, #WRONG CODE
                c(1)
            )
```

.currency_range() is a built-in method of oTree, it works just like normal Python range(), however, it also includes the last number. These developers are so inconsistent, isn't it? Actually the oTree developers are originally from the background of web development languages not just Python, therefore, you will likely catches some differences here and there (e.g. naming convention and functions)
IMPORTANT: self.group.sent_amount will result in error here. This is because we are inside the class definition of Group. There is no need to use .group . The correct code would be self.sent_amount (without .group). Currently, Otree documentation (https://otree.readthedocs.io/en/latest/tutorial/part3.html)still display this, but it has been confirmed as a bug/typo.

Similar to the previous project, we want to calculate the result in class Group by declaring a method set_payoffs(). *Under class Group(BaseGroup):*

```
In [ ]: 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
```

Step 3: Making templates

For this project, we need 3 pages for:

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

Send page
In Send.html, type the following within the content block:

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

{% formfields %}

{% next_button %}
```

SendBack page
In SendBack.html, type the following within the content block:

In [ ]:
```
<p>
    You are Participant B. Participant A sent you {{group.sent_amount}}
    and you received {{tripled_amount}}.
</p>

{% formfield group.sent_back_amount %}

{% next_button %}
```

Results
In Results.html, type the following within the content block:

In [ ]:
```
{% 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>
```

We use Django tags to make the Results page appears differently for P1 and P2

Step 4: Putting templates to work

Send page

In [ ]:
```
class Send(Page):

    form_model = 'group'
    form_fields = ['sent_amount']

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

We use is_displayed() to only show this to P1; P2 skips the page. the id_in_group is a built-in method that gives you the id of the player in a group. Otree automatically named the first player's ID whose joined the group starting from 1. More info regarding to these Otree methods, you can find it here: https://otree.readthedocs.io/en/latest/multiplayer/groups.html#groups (https://otree.readthedocs.io/en/latest/multiplayer/groups.html#groups)

SendBack page
In pages.py, type the following within the content block:

In [ ]:
```
class SendBack(Page):

    form_model = 'group'
    form_fields = ['sent_back_amount']

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

    def vars_for_template(self):
        return dict(
            tripled_amount=self.group.sent_amount * Constants.multiplication_f
actor
        )
```

vars_for_template() is another built-in method in Otree. It basically lets you perform a calculation before displaying it in HTML page. Although you can use javascript to do this, this is how it should be handled using only Otree.

Results page
In pages.py, type the following within the content block:

In [ ]:
```
class Results(Page):
    pass
```

Since we do not have to ask for any information (via form field) in the result page. You can just simply put pass

Wait pages
In pages.py, type the following within the content block:

In [ ]:
```
class WaitPageP1(WaitPage):
    title_text = "Custom title text"
    body_text = "Custom body text"

class ResultsWaitPage(WaitPage):
    after_all_players_arrive = 'set_payoffs'
```

We need 2 wait pages here because, the first player will play first, and the second player will have to wait for this player 1. Then during the time when player 2 is making the decision, player 1 has to wait.

Finally, we have to make sure that the order page_sequence is set correctly:

In [ ]:
```
page_sequence = [
    Send,
    WaitPageP1,
    SendBack,
    ResultsWaitPage,
    Results,
]
```

Step 5: Finalizing your session

Similar to Project 1 and 2, we just need to put this on our devserver and test it

In [ ]:
```
SESSION_CONFIGS = [
    dict(
        name='proj_simple_survey',
        num_demo_participants=3,
        app_sequence=['proj_simple_survey']
    ),
    dict(
        name='proj_public_goods',
        num_demo_participants=3,
        app_sequence=['proj_public_goods']
    ),
    dict(
        name='proj_trust_game',
        num_demo_participants=2,
        app_sequence=['proj_trust_game']
    ),
]
```

By now you should be familiar with the basic steps to create an experiment in Otree. In this lecture, we'll apply these steps in some of the common games and using some more advanced functions in Otree while doing so.

The trustworthy source for your reference would still be: https://otree.readthedocs.io/en/latest/conceptual_overview.html (https://otree.readthedocs.io/en/latest/conceptual_overview.html). However, some minor mistakes in Otree documents are unavoidable, please check the bug report if you think it is a mistake: https://github.com/oTree-org/oTree/issues?q=is%3Aissue+is%3Aclosed (https://github.com/oTree-org/oTree/issues?q=is%3Aissue+is%3Aclosed)

Project 4: Prisoner's Dilemma with chat

Project's spec: This is a one-shot "Prisoner's Dilemma". Two players are asked separately whether they want to cooperate or defect. Their choices directly determine the payoffs. Players are allowed to chat with each other in a chat room before giving their decision. Players' identities are kept anonymous.

As usual, we start our project with:

In [ ]:
```
otree startapp proj_prisoner_wchat
```

Step 1: Declare some constants

In [ ]:
```
class Constants(BaseConstants):
    name_in_url = 'prisoner'
    players_per_group = 2
    num_rounds = 1

    instructions_template = 'proj_prisoner_wchat/instructions.html'

    # payoff if 1 player defects and the other cooperates""",
    betray_payoff = c(300)
    betrayed_payoff = c(0)

    # payoff if both players cooperate or both defect
    both_cooperate_payoff = c(200)
    both_defect_payoff = c(100)
```

Here "instruction_template" is a new element/attribute. We will reuse this link, therefore assigning it to a constant variable for an easier call next time we need it.

Step 2: Declare models

```
In [ ]: class Player(BasePlayer):
            decision = models.StringField(
                choices=[['Cooperate', 'Cooperate'], ['Defect', 'Defect']],
                doc="""This player's decision""",
                widget=widgets.RadioSelect,
            )

            def other_player(self):
                return self.get_others_in_group()[0]

            def set_payoff(self):
                payoff_matrix = dict(
                    Cooperate=dict(
                        Cooperate=Constants.both_cooperate_payoff,
                        Defect=Constants.betrayed_payoff,
                    ),
                    Defect=dict(
                        Cooperate=Constants.betray_payoff, Defect=Constants.
        both_defect_payoff
                    ),
                )

                self.payoff = payoff_matrix[self.decision][self.other_player
        ().decision]
```

Here a new element is how we configured StringField. Without further configuration, models.StringField() will just return a blank space so that the player can type in something.

However, under the above configuration, specifically: "choices", "widget", and "doc".

- "choices": in Otree, this will create a dropdown menu with elements that you put in this list. As mentioned in the previous lecture, if you want to put a "facade/label" for the value of choices, you can do something like this: choices=[[1, 'Low'], [2, 'Medium'], [3, 'High'],]

  In our case, we want the value to be exactly the same as what is displayed, hence, I put ['Cooperate', 'Cooperate']. The first element is for value, and the second element is only for display [value, display]

  You can of course, put it as [1, 'Cooperate'] and [0, 'Defect']. This might accelerate with data analysis afterward, however, using a string as the value might help you remember what the value is for. In other words, it conveys more meaning. Plus, changing the string 'Cooperate' to 1 during your data analysis process isn't a computationally intensive task for computers nowadays.

- "widget": in Otree, widget only comes in two forms: RadioSelect or RadioSelectHorizontal. Radio select comes from the actual radio, it simply means you have to click and select what you want. Instead of a drop-down menu you would see something like this:

○ Male

○ Female

- "doc": is similar to the python """ """ docstring for function or class definition. It is a good way to remind you and others later what is the field for

One thing you might find confusing here is the use of dict(). This is a Python way to convert key, value pair into a dictionary type. You might recall in Block 1, we often declare a dictionary type directly such as dict1 = {'key1': 'value1, 'key2': 'value2'}. This is just another method to make a dict (https://docs.python.org/3/tutorial/datastructures.html (https://docs.python.org/3/tutorial/datastructures.html)). You can, of course, declare the dict type directly, without having to go through the built-in function dict().
I will, however, stick to the Otree convention instead. Under this convention, and Python dict() function, the key will always be a string.

Further explanation why the code is seemingly messy:

It might be tempting to do something like this for payment calculation:

```
In [ ]: # YOU SHALL NOT DO THIS!
            def set_payoff(self):
                payoff_matrix = dict(
                    Cooperate=dict(
                        Cooperate=c(200),
                        Defect=c(0),
                    ),
                    Defect=dict(
                        Cooperate=c(300), Defect=c(100)
                    ),
                )
```

It looks nicer, easy to understand and read (for now). But, if you remember this concept in Block 2, you know that it is a very bad practice. After a while, when you come back to the game, you will find it extremely time-consuming to teach yourself again how to change the payment scheme. And even if you remember how to do it perfectly, you will have to change it in several places, not just this one. Thus, refactoring elements that should be refactored is a good practice. The Constants class in Otree is there to remind you of this, so use it.

Since we are playing a 1 shot game, the class Subsession(): can be passed

As for the class Group(): we should have a method in place to calculate for the payments of the players in group

```
In [ ]: class Group(BaseGroup):
            def set_payoffs(self):
                for p in self.get_players():
                    p.set_payoff()
```

You might ask: "But, I did make a payment method for the individual player, what is this for, isn't it redundant?"

The answer is no. Since we only calculate the pay_off the moment we get all the inputs from the player. If only 1 player submitting the decision, you cannot calculate the payoff. Thus, most of the time you will want to put the calculation during the WaitPage for the entire group. We will come back to this in Step 4.

Step 3: Making templates

For this project, I will make the instruction appears not only at the beginning when you first started the experiment, but also during the decision-making screen. This might help participant less confused in case they forgot what they have to do.
Therefore, I create an "instructions.html" page that contains the instructions for this prisoner's dilemma game.

```
In [ ]: {% load otree %}

        <div class="card bg-light m-3">
            <div class="card-body">

            <h3>
                Instructions
            </h3>

            <link rel="stylesheet" type="text/css"
                  href="{% static 'global/matrix.css' %}"/>

            <p>
                In this study, you will be randomly and anonymously paired w
        ith another
                participant.
                Each of you simultaneously and privately chooses whether you
        want to
                cooperate or defect.
                Your payoffs will be determined by the choices of both as be
        low:
            </p>
            <p><i>In each cell, the amount to the left is the payoff for
                you and to the right for the other participant.</i></p>

            <table class='table table-bordered text-center'
                   style='width: auto; margin: auto'>
                <tr>
                    <th colspan=2 rowspan=2></th>
                    <th colspan=2>The Other Participant</th>
                </tr>
                <tr>
                    <th>Cooperate</th>
                    <th>Defect</th>
                </tr>
                <tr>
                    <th rowspan=2><span style="transform: rotate(-90deg);">Y
        ou</span></th>
                    <th>Cooperate</th>
                    <td>{{ Constants.both_cooperate_payoff }}, {{ Constants.
        both_cooperate_payoff }}</td>
                    <td>0, {{ Constants.betray_payoff }}</td>
                </tr>
                <tr>
                    <th>Defect</th>
                    <td>{{ Constants.betray_payoff }}, 0</td>
                    <td>{{ Constants.both_defect_payoff }}, {{ Constants.bot
        h_defect_payoff }}</td>
                </tr>
            </table>

        </div>
        </div>
```

The <div> tag defines a division or a section in an HTML document. This is usually used to segment your page or code for styling or perform certain tasks with javascript. It conveys no extra information, thus, some web developers will advise against using it too often. Since we are using Otree to create experiment and not making a commercial semantic webpage, we should focus on making it works rather than worrying about breaking convention.

----Starting section ----Warning: If you are interested in styling read on, otherwise, skip to the end

What is new here are these tags:
<div class="card bg-light m-3">
<div class="card-body">

It basically creates a card (like a 2D box wrapping around the text of the instructions). The card is set to background light color which you can see in here: https://www.w3schools.com/bootstrap4/bootstrap_colors.asp (https://www.w3schools.com/bootstrap4/bootstrap_colors.asp)
"card-body" is often used to create padding for a section within a card. More information here: https://getbootstrap.com/docs/4.1/components/card/ (https://getbootstrap.com/docs/4.1/components/card/)

These are just for styling, the experiment will still work without these configurations. Minimally, you can just put all the instructions inside a <p> tag.

These styling steps are often done at the end of your development cycle. You should not feel pressured to make a perfect HMTL template for your experiment right at the beginning.

You can easily find CSS and bootstrap materials online should you need to "beautify" your page. However, make sure that you don't accidentally introduce noise to the experiment. For example, color red could mean danger and alert, or a picture of a puppy/kitten will make people feel differently towards the experiment you are trying to conduct.

Furthermore, since these are primarily mark-up language and not programming, I will not focus on these in the lecture.

The good thing is if you just stick to the plain HTML display, the app making process will be a lot easier as well.

----Ending section ----

First page
After creating the instruction template. We need to create a template for the first page, the Introduction.html

In [ ]:
```
{% extends "global/Page.html" %}
{% load otree %}

{% block title %}
    Introduction
{% endblock %}

{% block content %}

    {% include Constants.instructions_template %}

    {% next_button %}

{% endblock %}
```

Aside from the normal template that we used in the previous lecture, we have a new tag {% include Constants.instructions_template %}. This tag basically takes everything in instructions_template (which is the 'prisoner/instructions.html') and put it in this template.

Second page
After reading the instructions the player now have to make a decision, we will now create a template for this page, which I named Decision.html

```
In [ ]: {% extends "global/Page.html" %}
        {% load otree %}

        {% block title %}
            Your Choice
        {% endblock %}

        {% block content %}

            <div class="form-group required">
                <table class="table table-bordered text-center" style="widt
        h: auto; margin: auto">
                    <tr>
                        <th colspan="2" rowspan="2"></th>
                        <th colspan="2">The Other Participant</th>
                    </tr>
                    <tr>
                        <th>Cooperate</th>
                        <th>Defect</th>
                    </tr>
                    <tr>
                        <th rowspan="2"><span>You</span></th>
                        <td><button name="decision" value="Cooperate" class
        ="btn btn-primary btn-large">I will cooperate</button></td>
                        <td>{{Constants.both_cooperate_payoff}}, {{Constant
        s.both_cooperate_payoff}}</td>
                        <td>{{ Constants.betrayed_payoff }}, {{Constants.bet
        ray_payoff}}</td>
                    </tr>
                    <tr>
                        <td><button name="decision" value="Defect" class="bt
        n btn-primary btn-large">I will defect</button></td>
                        <td>{{Constants.betray_payoff}}, {{ Constants.betray
        ed_payoff }}</td>
                        <td>{{Constants.both_defect_payoff}}, {{Constants.bo
        th_defect_payoff}}</td>
                    </tr>
                </table>
            </div>

            <p>Here you can chat with the other participant.</p>

            {% chat %}


            {% include Constants.instructions_template %}

        {% endblock %}
```

<div class="form-group required"> is a class used mainly for styling https://getbootstrap.com/docs/4.0/components/forms/ (https://getbootstrap.com/docs/4.0/components/forms/) . "required" means the input of the form is mandatory, the player has to give some input before continuing.

Here I put a table on our HTML page. You can check the basic HTML table here: https://www.w3schools.com/html/html_tables.asp (https://www.w3schools.com/html/html_tables.asp) , and here for the bootstrap table which I'm using: https://getbootstrap.com/docs/4.0/content/tables/ (https://getbootstrap.com/docs/4.0/content/tables/)

Another point worth noting here is the HTML button tag. When you use such tag, you almost always have to specify other attributes such as: "value" and "name". In this case, if they player click on the button with such tag:

<button name="decision" value="Cooperate" class="btn btn-primary btn-large"> I will cooperate </button>

the value="Cooperate" will be submitted. The class="btn btn-primary btn-large" is again for styling https://getbootstrap.com/docs/4.0/components/buttons/ (https://getbootstrap.com/docs/4.0/components/buttons/). You can simply don't use it and a default button will be displayed instead. Try it!

Normally, you will see the Django tag {% formfields %}, which automatically puts the form fields inside the template for you. However, we don't see it in this case. So how does this HTML page knows where to record the data?

This is when "name" attribute comes in. The name attribute such as "decision" must be the same as what you declared in pages.py and models.py, as usual, we will come to pages.py after we are done with the templates.

Another interesting point in this template is the tag {% chat %}. This will make a chat box so that the participant in the group can chat with each other. It is extremely convenient and easy to use as the chat_log will also be automatically recorded. You don't have to do anything else.

Third page
After making the decision the player will have to wait for the other player in the group. Remember, we don't have to make a template for wait pages unless you want some customization. For this third page, I just want to display the result on the screen. In Results.html:

In [ ]:
```
{% extends "global/Page.html" %}
{% load otree %}

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

{% block content %}

    <p>
        {% if same_choice %}
            Both of you chose to {{ my_decision }}.
        {% else %}
            You chose to {{ my_decision }} and the other participant
chose to {{ opponent_decision }}.
        {% endif %}
    </p>

    <p>
        As a result, you earned {{ player.payoff }}.
    </p>

    {% next_button %}

    {% include Constants.instructions_template %}

{% endblock %}
```

Step 4: Putting templates to work

I will stick to the orders of pages as in the previous step. So we'll start first with the Introduction page:

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


class Introduction(Page):
    timeout_seconds = 100
```

Here, since we don't have any forms to fill we could have just written "pass". But if you want to limits the time the players can spend on this page you can use: timeout_seconds. This is another built-in method of Django.

For those of you who may know Javascript, this is really in seconds not milliseconds!

Next, I would like to declare the class for the Decision page:

In [ ]:
```
class Decision(Page):
    form_model = 'player'
    form_fields = ['decision']
```

Nothing really new here, so I'll continue to our wait page:

```
In [ ]: class ResultsWaitPage(WaitPage):
            after_all_players_arrive = 'set_payoffs'
```

"after_all_players_arrive" is another built-in method, it simply triggers the method 'set_payoffs' after all the players have reached the WaitPage.

The Otree developers try to make methods call simpler for non-programmers. Therefore, often you'll see strange conflicts with what we have learned about Python in Block 1. Such as method calls (set_payoffs) without parentheses (e.g "set_payoffs()") or where it actually comes from.

```
In [ ]: class Results(Page):
            def vars_for_template(self):
                me = self.player
                opponent = me.other_player()
                return dict(
                    my_decision=me.decision,
                    opponent_decision=opponent.decision,
                    same_choice=me.decision == opponent.decision,
                )
```

A new element in this Results page is the vars_for_template() method. This method is here because the HTML template should not be used to performed math or calculation. (You can, however, use javascript for this purpose directly on the template).

As the name suggests, if you want to get some traditional Python coding, logic or calculation, you should put it inside this method. Then use the variable name to display it on the page.

For example, in this case, we are configuring the Results page, and determine the value of "my_decision" and "opponent_decision". As you can see here, these can be displayed back to the HTML page:

```
In [ ]:     <p>
                {% if same_choice %}
                    Both of you chose to {{ my_decision }}.
                {% else %}
                    You chose to {{ my_decision }} and the other participant
        chose to {{ opponent_decision }}.
                {% endif %}
            </p>
```

Finally, we configure the page sequence:

```
In [ ]: page_sequence = [Introduction, Decision, ResultsWaitPage, Results]
```

Step 5: Finalizing your session

As usual, we have to put this on settings.py for the demo to work:

In [ ]:
```
SESSION_CONFIGS = [
    dict(
        name='proj_prisoner_wchat',
        num_demo_participants=2,
        app_sequence=['proj_prisoner_wchat']
    ),
]
```

Project 5: Ultimatum Game

Since we are already somewhat familiar with Otree and the steps outlined in the previous project. I will quickly go through the process of creating the ultimatum game in Otree. Later on, we will use this game to experiment with further settings in Otree for multiple rounds, treatments, and randomization.

Project's spec: There are two-player, one in the role of a proposer (A) and the other in the role of a responder (B). The proposer proposes how to split an endowment of 100 Euro (in the step of 10 Euro). The responder after seeing the proposal decides whether to accept or reject it. If the responder accept it, they will follow with the proposer's plan. If the responder rejects it, both receive nothing.

In [ ]:
```
otree startapp proj_ultimatum
```

Step 1: Declare some constants

In [ ]:
```
class Constants(BaseConstants):
    name_in_url = 'Ultimatum_Game'
    instructions_template = 'proj_ultimatum/instructions.html'
    players_per_group = 2
    num_rounds = 2
    endowment = c(100)
```

It might be tempting to define the keep_give ratio here in Constants (there are some materials online that would do so). It is very messy coding and goes against what is recommended in Otree (https://otree.readthedocs.io/en/latest/models.html#constants (https://otree.readthedocs.io/en/latest/models.html#constants)). Constants should be a place to put variables that is unlikely to change across players.

In [ ]:
```
# YOU SHALL NOT DO THIS!
keep_give = [(i, c(100) - i) for i in currency_range(c(0), c(100), c
(10))]
```

Step 2: Declare models

Note that in this game we play 2 rounds, which means 2 sub-sessions. However, since there are no changes between rounds, we put pass in the definition of class Subsession():

```
In [ ]: class Subsession(BaseSubsession):
            pass
```

```
In [ ]: class Group(BaseGroup):
            send_amount = models.CurrencyField(label='How much would you lik
        e to offer player B?',
                                               choices = currency_range(c
        (0), c(100), c(10)))
            offer_accept = models.BooleanField(choices=[[True, 'Yes'], [Fals
        e, 'No']], label='Do you accept this offer?')


            def creating_session(self):
                self.group_randomly(fixed_id_in_group=True)

            def set_payoffs(self):
                proposer = self.get_player_by_role('proposer')
                responder = self.get_player_by_role('responder')
                if self.offer_accept:
                    proposer.payoff = Constants.endowment - self.send_amount
                    responder.payoff = self.send_amount
                else:
                    proposer.payoff = 0
                    responder.payoff = 0
```

```
In [ ]: class Player(BasePlayer):

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

The method role(self) **must be named exactly** as "role" and not other names!

Step 3: Making templates

instruction.html

```
In [ ]: {% load otree %}

        <div class="card bg-light m-3">
            <div class="card-body">

            <h3>
                Instructions
            </h3>

            <link rel="stylesheet" type="text/css"
                  href="{% static 'global/matrix.css' %}"/>

            <p>
                In this study, you will be randomly and anonymously paired w
        ith another
                participant.
                First, every player will be randomly paired with another pla
        yer. In other words, you will have a counterpart, but you will not b
        e told who it is. Your identity will also remain hidden from your co
        unterpart.
                The two participants in a pair will have two different role
        s: the Proposer and the responder. You will be assigned randomly to
        a role, and it will be displayed on the next page.
                The two of you will together receive {{Constants.endowment}}
        Euro. The experiment is about how to divide this amount. The propose
        r will make the responder a take-it-or-leave-it offer, which the res
        ponder can accept or reject. If the offer is rejected, both will rec
        eive 0.00 Euro.

            </p>
        </div>
```

Introductions.html

```
In [ ]: {% extends "global/Page.html" %}
        {% load otree %}

        {% block title %}
            Introduction
        {% endblock %}

        {% block content %}

            {% include Constants.instructions_template %}

            {% next_button %}

        {% endblock %}
```

Send.html

In [ ]:
```
{% extends "global/Page.html" %}
{% load otree %}

{% block title %}
    Your Choice
{% endblock %}

{% block content %}

<p>
    You are the Proposer. You have to divide {{Constants.endowment}}
Euro. How much do you want to send to the other participant?
</p>

{% formfield group.send_amount %}

{% next_button %}

    {% include Constants.instructions_template %}

<p>YOU MIGHT WANT TO ADD EXTRA INSTRUCTION FOR THE PROPOSER HERE!</p
>

{% endblock %}
```

Receive.html

In [ ]:
```
{% extends "global/Page.html" %}
{% load otree %}

{% block title %}
    Your Choice
{% endblock %}

{% block content %}

<p>
    You are the Responder. The Proposer wants to give you {{group.se
nd_amount}}
</p>

{% formfield group.offer_accept %}

{% next_button %}

    {% include Constants.instructions_template %}

<p>YOU MIGHT WANT TO ADD EXTRA INSTRUCTION FOR THE RESPONDER HERE!</
p>

{% endblock %}
```

Results.html

```
In [ ]: {% extends "global/Page.html" %}
        {% load otree %}

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

        {% block content %}

            <p>
                {% if group.offer_accept %}
                    The Responder accepts the offer.
                {% else %}
                    The Responder declines the offer.
                {% endif %}
            </p>

            <p>
                As a result, you earned {{ player.payoff }}.
            </p>

            {% next_button %}

        {% endblock %}
```

Step 4: Putting templates to work

```
In [ ]: class Introduction(Page):
            pass
```

```
In [ ]: class Send(Page):
            form_model = 'group'
            form_fields = ['send_amount']

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

```
In [ ]: class Receive(Page):
            form_model = 'group'
            form_fields = ['offer_accept']

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

```
In [ ]: class WaitProposer(WaitPage):
            pass
```

```
In [ ]: class ResultsWaitPage(WaitPage):
            after_all_players_arrive = 'set_payoffs'
```

```
In [ ]: class Results(Page):
            pass
```

Step 5: Finalizing your session

```
In [ ]: SESSION_CONFIGS = [
            dict(
                name='proj_prisoner_wchat',
                num_demo_participants=2,
                app_sequence=['proj_ultimatum']
            ),
        ]
```

You have made the experiment and it works. Now we will discuss how to put your experiment online.

There are 3 main ways to do this:

1. Turn your computer into a server
2. Use the existing hosting server of your company/institutes
3. Use an online hosting server

We will only discuss the third option since it is probably the easiest option out the 3 and you can feel relatively safe not to worry about security issues or personal files on your computer being exposed.

Since oTree recommends Heroku as the default hosting server of choice, we will do the same in the lecture. Another advantage of using the default (Heroku) for Otree apps is that when you have problems with the server, there are many more people you could ask for help.

Setting up Heroku

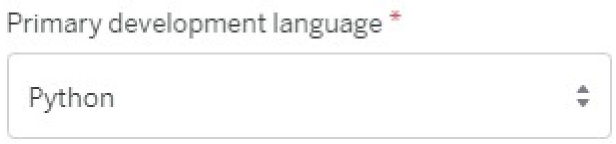
In this section, we will attempt to set up Heroku server for Otree apps manually.

Side note: Another option is to use Heroku Hub (https://www.otreehub.com/ (https://www.otreehub.com/)) which will automate the setup process for you, however, it is only for free for public projects. Furthermore, the developers admittedly warned that the service could end or change at some point. Thus, we will skip Otreehub in this lecture.

Step 1: Sign up for a free account at heroku

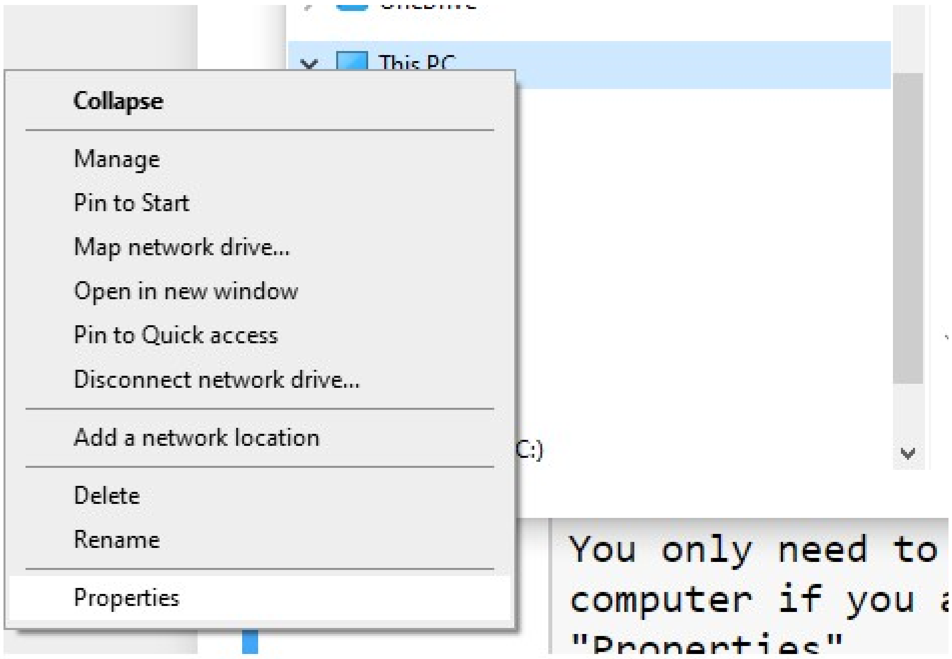
You will want to sign up here: https://signup.heroku.com/ (https://signup.heroku.com/)

Make sure you choose Python as the primary language


Primary development language *
Python

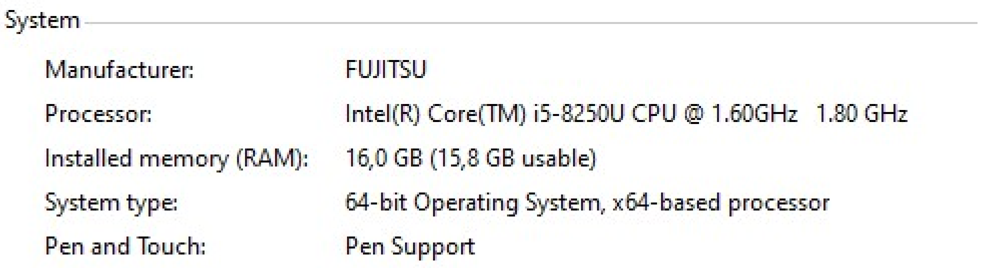
Step 2: Installing Heroku CLI

You will also need to install the Heroku CLI: https://devcenter.heroku.com/articles/heroku-cli (https://devcenter.heroku.com/articles/heroku-cli)
You only need to download the package and install it on your computer. Make sure you download the correct bit system of your computer if you are using Windows system. You can find this information by right-clicking "My PC" or "This PC" and select "Properties"



You only need to
computer if you a
"Properties"

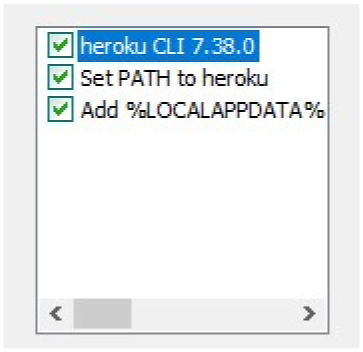
In my case, it is a 64 bit system:



System

Manufacturer:	FUJITSU
Processor:	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz
Installed memory (RAM):	16,0 GB (15,8 GB usable)
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	Pen Support

When you install Heroku CLI, make sure all the options are checked:

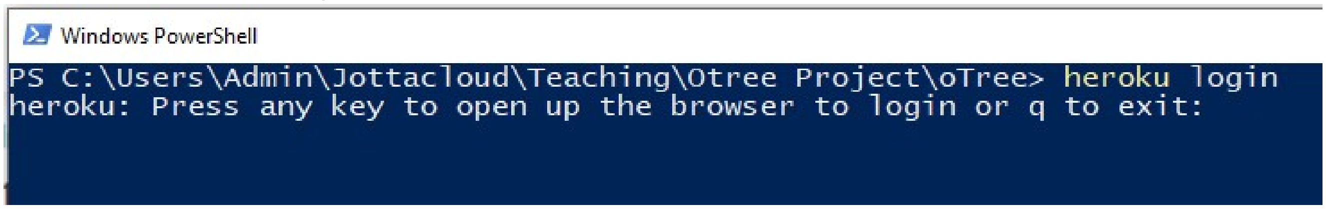


- [✓] heroku CLI 7.38.0
- [✓] Set PATH to heroku
- [✓] Add %LOCALAPPDATA%

Now, go to your root project folder (where "settings.py" is) and open PowerShell. Type:

```
In [ ]: heroku login
```

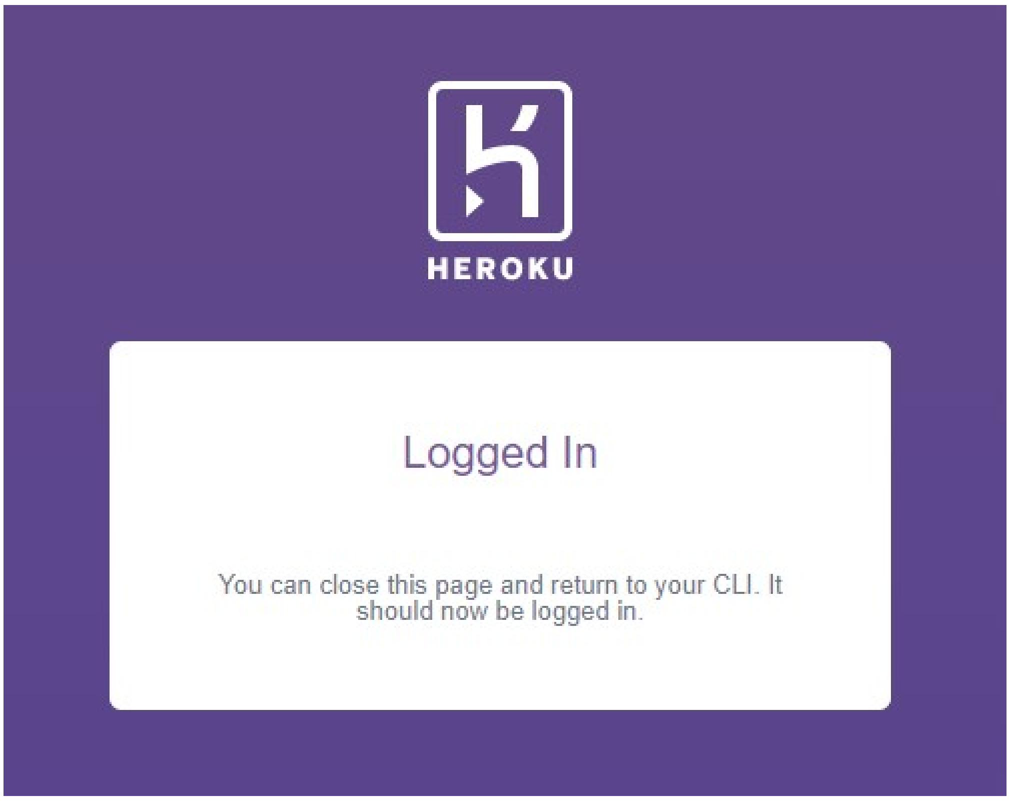
You should see something like this:



Windows PowerShell

```
PS C:\Users\Admin\Jottacloud\Teaching\Otree Project\oTree> heroku login
heroku: Press any key to open up the browser to login or q to exit:
```

Press any keys, then a web browser windows will open. You just need to login and should see something like this afterward:



HEROKU

Logged In

You can close this page and return to your CLI. It should now be logged in.

You just need to click on the HEROKU logo, it will lead you to the Heroku dashboard. But before we get further into Heroku dashboard, you should install Git if you have not done so: [https://git-scm.com/download/win (https://git-scm.com/download/win)](https://git-scm.com/download/win)

Step 3: Installing and using Git

In PowerShell type:

```
In [ ]: git init
```

Now you need to create an app on Heroku. This might be misleading as Heroku app is not equivalent to otree app. You can think of Heroku app as the whole Otree Project (which may comprise of many otree apps).
To do this, type:

```
In [ ]: heroku create my-app-name
```

This will create a website with part of the link like this: my-app-name.herokuapp.com
In terms of experimental economics, you should name your heroku app something neutral, names such as: *prisoner-dilema or ultimatum-game or trust-game* may suggest players what the experiment is about (which is a no-no). Of course, you can hide the actual link name with some extra steps, however, why not avoid it now?
In actuality, your Heroku app's name must be unique across all projects that are currently registered in Heroku. So you will unlikely be able to name your heroku app as *prisoner-dilema* or similar. These name are usually already taken by someone else or by Otree developers to show case the game.

To deploy your app on Heroku, you will need a credit card. If you do have a visa card and really want to deploy your game/study later. Then, the next step is to install Redis add-on. You can think of redis simply as intermediate database storage (cache) that can help you store and retrieve data quickly. Data stored in Redis by key-value pairs like a Python dictionary, not rows or columns. The redis add-on in Heroku will also provide you analytics on your website traffic and memory usage. In layman term, it shows you if your website is getting a traffic jam (because so many people are trying to use it) or not. The free redis add-on can handle a handful of players at the same time, therefore, it is usually used for testing only. In the real implementation of a study, you may want to upgrade to a paid plan so that your Otree app can handle more players simultaneously.

The cost of running a server on Heroku is very cheap *(if you remember to turn off all the dynos and add-ons when not used)*. Therefore, it is also perfect for students who want to conduct a "traditionally laboratory" experiment online. Always check out the Heroku dashboard, the information about billing should be visible!

Redis is a must for oTree, without redis add-on Otree app on Heroku won't be able to run!

To install redis add-on type:

```
In [ ]: heroku addons:create heroku-redis:hobby-dev
```

Redis add-on comes with many pricing scales, hobby-dev means it's free. However, you still need to enter your VISA card number. If you don't have a visa card or have trouble open one, you can send the Heroku Support a message.

In order to ensure that your app works with otree. You need to check the otree version that you are using. Type:

```
In [ ]: otree --version
```

Then in the root folder (where settings.py is), find this file *requirements_base.txt*. Open it and replace "everything" in it with:

```
In [ ]: otree>=X.X.X
```

X.X.X is the otree version that you saw after typing otree --version in the Command Prompt or PowerShell. As mentioned previously, we are using:
otree >= 3.3.11

Further note: The content in requirements_base.txt will tell Heroku which packages to use to compile your program. Therefore, if you were using additional packages in Python, such as numpy or pandas, they need to be added to your requirements_base.txt also.

Next, you need to specify which version of Python Heroku should be using to run your app. To do this, create a .txt file name: "runtime.txt" inside the root folder with the contents:

```
In [ ]: python-3.7.9
```

Although I was using python 3.7.6 (for example). Here I will still have to specify python 3.7.9. There are only a few Python supported runtime. Please check them here first: https://devcenter.heroku.com/articles/python-support#supported-runtimes (https://devcenter.heroku.com/articles/python-support#supported-runtimes) . Consider this not as an inconvenience, but another layer of information as Python, unlike other programming languages, changes quite often over time. Keeping track of it from both the server-side and development side (the python version in your computer) is a good thing.

Now, we will really push your files onto Heroku using Git. To do this, type:

```
In [ ]: git add .
        git commit -am "your commit message"
```

Commit message is somewhat like variable naming. It definitely will not help if you write Commit 1/Commit 2/Commit 3 and so on, as it does not provide you any information. The reason why you need to provide a commit message is that when something wrong happens with your app, and you want to revert it back to a specific commit, the message will help tremendously. Therefore, something such as: "Fixing error in runtime.txt" is a much better commit message.

Finally, you can really push all the necessary files on Heroku and share it on the internet:

```
In [ ]: git push heroku master
```

To open your Heroku app, type:

```
In [ ]: heroku open
```

Step 4: Further updating and deploying </h2>

When you just want to update your games, or including new games. You need to just follow this order:

```
In [ ]: git add .
        git commit -am "my commit message"
        git push heroku master
        # next command only required if you added/removed a field in models.
        py
        heroku run "otree resetdb"
```

You will need to turn off the debug mode when you are ready to run the experiment for real, to do this, type:

```
In [ ]: heroku config:set OTREE_PRODUCTION=1
```

You will also need to set up an admin password when you run the real study:

```
In [ ]: heroku config:set OTREE_AUTH_LEVEL=STUDY
```

To only showcase your work, you can also set this to demo. Anyone interested can check out your Heroku application online. You can, of course, downgrade all add-ons and Postgres to the free version for this purpose.

```
In [ ]: heroku config:set OTREE_AUTH_LEVEL=DEMO
```

You can find the default password for the admin account inside settings.py file.

When you run a study, don't forget you need to back up your file frequently. You can do this by clicking on Heroku Postgres on the Heroku Dashboard. Then click on PG Backups. For more information, check here: https://devcenter.heroku.com/articles/heroku-postgres-backups (https://devcenter.heroku.com/articles/heroku-postgres-backups)

```
In [ ]:
```