**Lesson 4: Fields and customizing oTree**

What are the different data types possible? What are widgets? How to customize Group and Player models

**Field types**

Here are the main field types:

* BooleanField (for true/false and yes/no values)
* CurrencyField for currency amounts;
* IntegerField / PositiveIntegerField
* FloatField (for real numbers)
* StringField (for text strings, one-line text area)
* LongStringField (for long text strings; its form widget is a multi-line text area)

More can be found on: <https://docs.djangoproject.com/en/2.1/ref/models/fields/#field-types>

**Choices**

When you want to limit the list of choices, you can define them using the choices option. Numbers are an easy way to code the choices, but you can also use StringField and text names for the options.

dynamic = models.IntegerField(  
 label="I can dynamically change the context of a web page",  
 choices=[  
 [1, 'Unsure'],  
 [2, 'Agree partially'],  
 [3, 'Agree'],  
 ],   
 widget=widgets.RadioSelect,)

**Widget types**

The default “widget” for choices is a dropdown menu.

Other options available:

* CheckboxInput
* RadioSelect
* RadioSelectHorizontal
* Slider

**Task 1: Test out all Fields and Widgets**

Create variables for each field type listed here. For example:

Age: (Positive)IntegerField

Name: StringField

Occupational description: LongStringField

Student (yes or no), BooleanField + choices

Department: IntegerField + choices

How many points do you expect to earn in this task? CurrencyField

How many digits of the mathematical constant Pi do you recall? FloatField

With choices it is recommended to you widgets – try them out!

**Adding custom packages**

You can add custom features to your applications. As example, you can download a custom Field type:

**Task 2: Adding packages**

You can add custom made features to your oTree environment by downloading them, adding the folder in your project folder (the one with settings.py) and the settings, under installed apps:

INSTALLED\_APPS = ['otree']

1. download the files from <https://github.com/chapkovski/radiogrid-example-otree>
2. Copy-paste the folder ‘radiogrid’ it to your project folder.
3. Add ‘radiogrid’ to the installed apps in the settings.py file

INSTALLED\_APPS = ['otree’, ’radiogrid']

1. Import radiogrid to your models.py

**from** radiogrid **import** RadioGridField

Radiogrid helps you to do neat questionnaires where the answer space stays the same. For example, when you ask in a 7-point scale how much people agree with a set of sentences.

You need to define the rows, here: statements to agree, outside any of the class structures, and you need to define the columns, the scale of measurement, similarly.

ROWS = (  
(1, **"I like blue"**),  
(2, **"I like red"**),  
)  
COLUMNS = (  
(1, **"Agreed"**),  
(2, **"Disagreed"**),  
)  
  
**class** Player(BasePlayer):  
 agree = RadioGridField(rows=ROWS, values=COLUMNS, null=**True**, require\_all\_fields=**True**, verbose\_name=**"Do you agree?"**)

**Task 3: Another way: make your own Formfield (suggested to do with a function)**

**def** make\_field(label):  
 **return** models.IntegerField(  
 choices=[  
 [1, **"agree"**],  
 [2, **"agree somewhat"**],  
 [3, **"don't know"**],  
 [4, **"disagree somewhat"**],  
 [5, **"disagree"**]  
 ],  
 label=label,  
 widget=widgets.RadioSelect,  
 )

agree = make\_field(**"Do you agree?"**)

**Task 4: Computing variables**

Variables do not need to be inputs by the participants, you can calculate them from other variables etc. This is what we do with the payoffs, for instance.

Modify the public good game (lesson2) such that:

1. Remove the group variables contribution1-3 and add variable contribution under the Player class in the models.py file. Note this can be a currency field.
2. Add a total\_contributions variable under the Group class.
3. Remove or modify the old payoff function to make a new one. Either have it under Player:

**def** set\_payoffs(self):  
 self.payoff = Constants.endowment - self.contribution + 2 / 3 \* self.group.total\_contributions (unintentional line break)

Or under Group:

**def** set\_payoffs(self):  
 **for** each **in** self.get\_players():  
 each.payoff = Constants.endowment - each.contribution + 2 / 3 \* self.total\_contributions (unintentional line break)

1. To calculate the total contributions, add on the wait page, after all players arrive:

**def** after\_all\_players\_arrive(self):  
 **for** player **in** self.group.get\_players():  
 self.group.total\_contributions = self.group.total\_contributions + player.contribution

Once the group variable has been set, you can trigger the set\_payoffs() function

If stored under Group:

self.group.set\_payoffs()

If stored under Player:

**for** player **in** self.group.get\_players():  
 player.set\_payoffs()

**Lesson 5: Session variables, randomization and group structures**

What are session variables? Refreshing how to start an app, how to construct an app. Multiple groups, changing group structure, randomizing roles

**Session variables**

Some variables can be made session wide and easily modified by whoever is running the session. For example, treatments… to demonstrate how to read variables from the session configurations, here’s a silly, simple, voting game

**Exercise 1: Voting app**

1. otree startapp vote

2. add the app to settings.py in the usual way, plus add a ‘treatment’ = ‘blue’ (or red) with a note

# treatment can be blue or red

3. models.py: to read the treatment, create a subsession variable color that is a string variable:

Under Subsession, using the function creating\_session(self): set the color of the subsession to the one set in the treatment. You can find the treatment variable under self.session.config[‘treatment’]:

**class** Subsession(BaseSubsession):  
 color = models.StringField()  
 **def** creating\_session(self):  
 self.color = self.session.config[‘treatment’]

4. If the color of the subsession is blue, show participants a blue hello in the first page. If the color of the subsession is red, show participants a red hello.

Rest of the app:

5. Group size 2.

6. Give each participant a vote: red or blue.

7. Payoffs: Each player should get zero if they vote differently and they should get a positive payoff if they vote the same.

Hopefully we will get a lot of variety in how you code this problem: we can discuss about the different solutions that people came up with.

**Exercise 2: Change groups**

1. Expand the voting game for 2 rounds and 4 players.

Do this by adjusting the number of demo participants in the settings.py file.

Remember to check that the code works with 4 players first, before going to the next step.

1. After the first round, reshuffle the groups.

If you choose a random reshuffle, there is still a 33 % chance (?) that the groups do not actually change.

class Subsession(BaseSubsession):  
 def creating\_session(self):  
 self.group\_randomly()

You can make a 50-50 choice between two shuffles, 13 24 or 14 23 (order doesn’t matter here).

You must define your own shuffle function that creates options 13 24 and 14 23 and a random variable that determines which option is taken.

Groups are stored under subsession, you can access them via self.get\_group\_matrix() and they look the following

[[<Player 1>, <Player 2>],

[<Player 3>, <Player 4>]]

What to do?

1. Save the original grouping under the matrix “original”. Notice that the group matrix is a list of lists.

original = self.get\_group\_matrix()

1. You can double check that everything looks ok with the print function, and that you remember the right syntax. original[0][0] should get you player element 1. The models.py Subsession class is run when you start a session – you will see the prints in the command window.

print(original)

print(original[0][0]]

1. You should define the option1324 and option1423 using these player elements. (I’m quite sure numbers won’t do).

option1324 = [[original[0][0], original[1][0]],

[original[0][1], original[1][1]]]

print(option1324)

1. Draw a random number and using the self.set\_group\_matrix() function, after round 1, set the group matrix to be one of the options.

To draw a random number, import random

You can again google instructions on how random numbers are generated.

For example, generate a random number between 0 and 1, and if it is less than 0.5, choose option 1324, and if it is more than or equal to 0.5, choose 1423.

Again, there is more than 1 way to do this.

Round number can be accessed “self.round\_number” under the subsession.

New group matrix can be set with self.set\_group\_matrix(the new matrix), under the subsessions creating session function.

**Exercise 3: Random roles**

Go back to the trust game and randomize the trustee and trustor roles. At the moment they depend on the ID in group.

You can use the same logic as above.

What if you have to randomize 3 out of 5 participants to be type 1 and the remaining two to be type 2?

### How does oTree work?

oTree is based on Django, so to ask how oTree works is essentially to ask how Django works.

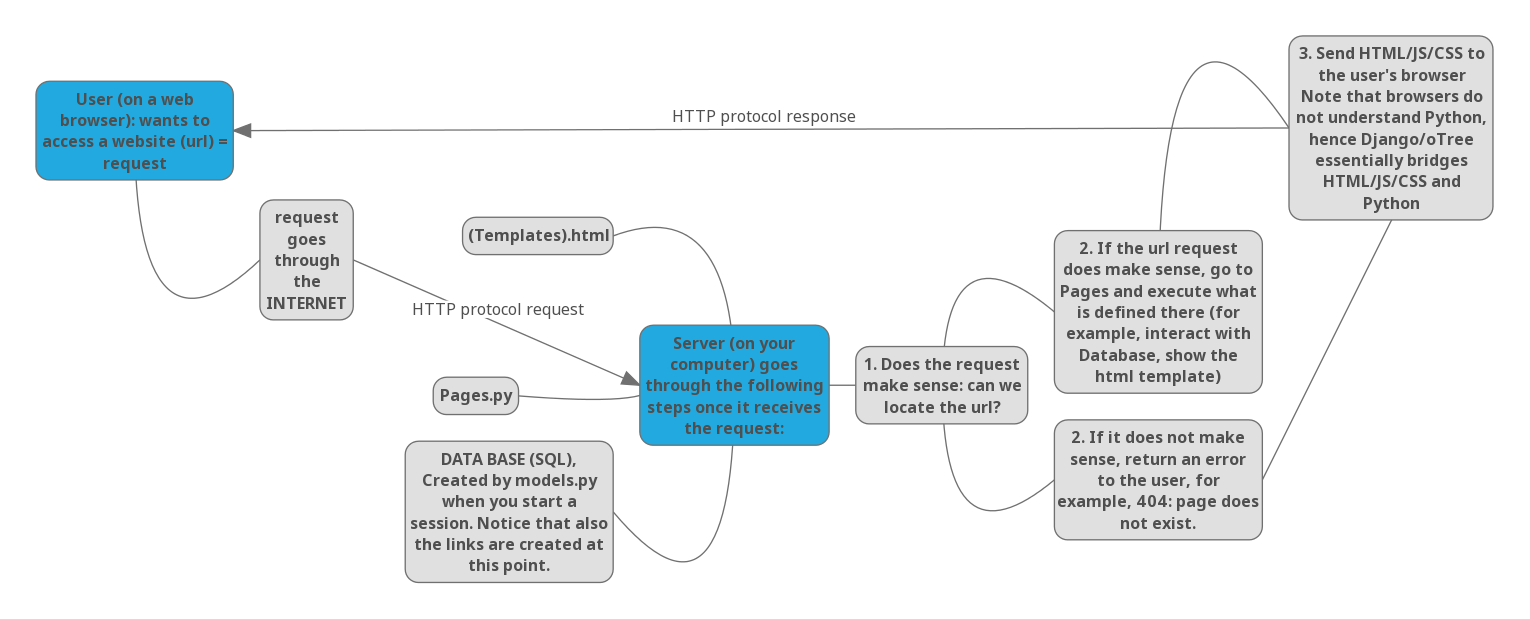
To make oTree user friendly for beginners, a lot of these functions are hidden in a safe place. You can find them in the oTree and Django core python files that are stored in the python library of site packages. They are visible in the PyCharm project menu under External Libraries

For me the packages are stored in in the main user file:

…\AppData\Local\Programs\Python\Python36\Lib\site-packages

There you will find folders for Django and oTree, and many other packages.

Flow chart of what happens with user requests (user accessing a web page url)



One video that explains for Django works <https://www.youtube.com/watch?v=JOLs3X8od5g>

* note that views.py corresponds to pages.py with oTree
* note that settings.py file in the current oTree version is “cleaned” up of all technical stuff. However, through the installed oTree app, I presume, we get to the settings.py of otree and otree\_startup

Django and oTree are made so that you do not need to worry about the technical stuff – and as the platforms have progressed on and become more advanced, the basic functions are starting to look quite intractable…

### Lesson 6: Timeouts, Django custom models and real effort tasks

**Timeouts**

You can limit participants’ time on a page by using timeouts.

class Page1(Page):

timeout\_seconds = 60

When timeout happens, you must often define, how the variables play out – however, these functions are somewhat unreliable sometimes…

def before\_next\_page(self):

if self.timeout\_happened:

self.player.my\_random\_variable = random.random()

Or in similar way to defining timeout seconds, you can define timeout submission

class Page1(Page):

timeout\_seconds = 60

timeout\_submission = {'accept': True}

### Task 1: Add timeout\_seconds to any page and see what happens

### Real effort task

1. Download minimum folder from <https://github.com/chapkovski/jbef_ret>
2. Add a line to the settings.py:   
   EXTENSION\_APPS = [**'minimum'**]

Real effort tasks require participants some activity, instead of, for example, stated effort.

The minimum task downloaded above requires participants to do some math tasks.

Go through what is happening in the real effort task file:

On top of the usual, we have routing.py and consumers.py files in the otree\_extensions folder. These files define some pathways that facilitate either:

participant-server-participant real time interactions OR

participant-server-participant**s** real time interactions a.k.a. communication.

Essentially, the server can decide to push information on the user’s webpage without the user having to do anything. Without these files, user’s webpage can only change if the click/submit/refresh the page.

This is due to the fact that we open WebSockets between the user and the server. The WebSockets are created on the user end by JavaScript, and with python on the server side, in consumers.py file. The routing file connects the two.

### Exercise: Transformation

The goal is to keep the format of the application, but change the task into a summation of two two-digit numbers. What do we need to change in the application?

We go first through answers to a more difficult task together (stroop), before you tackle the summation task.

But before we get started with the stroop, let’s identify functions and variables in each of the key files that need to be changed, added or deleted.

Start with Models.py file: What do you need to change this is file? What new variables should be created or replaced? Which functions?

Pages.py: What needs to be changed? Would adding more time make modifying the code easier?

Consumers.py: What needs to be changed here? The consumer’s task is to “consume” the messages sent through the websockets to the server. Here, the consumer checks if the answer given by the user is correct, it keeps track of the correct answers, asks for a new task to be created, and sends a new task to the user while time remains.

WorkPage.html: What needs to be changed here? Instructions? Question? Answer buttons?

Routing.py: What needs to be changed?

JavaScript in channels-js.html: What needs to be changed? What information is to be collected, what is to be received?

A stroop test: to name the color or the text.

First, add to model constants the choices available (we want to keep the format flexible, but let’s start with 4 choices):

**class** Constants(BaseConstants):  
 name\_in\_url = **'minimum\_ret'** players\_per\_group = **None** num\_rounds = 1  
 choices = [**'green'**, **'pink'**, **'orange'**, **'blue'**]

1. Models.py: Task body itself: instead of number, we need to generate color and text from the choices defined above.

task\_text = models.StringField()  
task\_color = models.StringField()  
last\_answer = models.StringField()  
  
**def** create\_stroop(self):  
 self.task\_text = random.choice(Constants.choices)  
 self.task\_color = random.choice(Constants.choices)  
 self.last\_answer = self.task\_text  
 *# if you want to track the text instead, you can change to the text variable instead of the color*

Note: The function must also be triggered in the Subsession.

1. Consumers.py: The way we handle the answer must change…

**def** receive(self, text=**None**, bytes=**None**, \*\*kwargs):  
 *# using the keyword we get the player* p = Player.objects.get(id=self.kwargs[**'player\_id'**])  
 *# we receive the answer* answer = text.get(**'answer'**)  
 print(answer)  
 *# if the answer is not empty....* **if** answer:  
 p.num\_tasks\_total += 1  
 **if** str(answer) == p.last\_answer:  
 p.num\_tasks\_correct += 1  
 p.create\_stroop()  
 p.save()  
 self.send({**'task\_color'**: p.task\_color,  
 **'task\_text'**: p.task\_text,  
 **'num\_tasks\_correct'**: p.num\_tasks\_correct,  
 **'num\_tasks\_total'**: p.num\_tasks\_total, })

1. The html task must be updated:

<**div**> What is the color you see?  
  
<**h1 id="stroop" class="question" style ='color**:{{ **player**.**task\_color** }}**'**> {{ **player**.**task\_text** }} </**h1**>  
  
 {% **for color in Constants**.**choices** %}  
 <**button class="answer btn btn-success" data-color=**{{ **color** }} **type="button"   
 style="background-color**:**lightgoldenrodyellow**; **color**: {{ **color** }}**"**><**b**>{{ **color** }}</**b**></**button**>  
{% **endfor** %}  
  
</**div**>

1. The JavaScript must be changed to facilitate a different set of messages going through

socket.onmessage = **function** (event) {  
 *{# We convert the data into a javascript object: #}* **var** obj = **jQuery**.**parseJSON**(event.**data**);  
 ***console***.log(**"color, obj.task\_color"**);  
 *{# We show a new task #}* **$**(**'h1.question'**).html(obj.task\_text);  
 ***document***.getElementById(**"stroop"**).**style**.**color**=obj.task\_color;  
 *{# ...and we update counters of correct and total answers #}* **$**(**'span#num\_tasks\_total'**).html(obj.num\_tasks\_total);  
 **$**(**'span#num\_tasks\_correct'**).html(obj.num\_tasks\_correct);  
};  
*{# The following is triggered when a user clicks on the answer button #}***$**(**"button"**).on(**"click"**, **function** () {  
 **var** msg = {**'answer'**: **$**(**this**).data(**'color'**)};  
 **if** (socket.**readyState** === ***WebSocket***.**OPEN**) {  
 *{# we convert it to json format and send it back to the server #}* socket.send(***JSON***.stringify(msg));  
 }  
});

**Django custom models**

Besides using Player and Group, you can set up your own model structures – however, word of warning, these models are not stored into the data base without some extra steps.

Why would you need another level of data? For example, in my experiment, I had groups of employers and workers that formed pairs that change every round. Hence, I was already using Player and Group, but needed a model for contracts, to store variables like wage and effort under it, as well as which employer and worker are part of the contract.

**class** JobContract(djmodels.Model):  
 employer = djmodels.ForeignKey(Player, related\_name=**'contract'**, unique=**True**)  
 worker = djmodels.ForeignKey(Player, blank=**True**, null=**True**, related\_name=**'work\_to\_do'**)  
 wage = models.IntegerField()  
 accepted = models.BooleanField()  
 effort = models.PositiveIntegerField(initial=0)

The data on this JobContract model can be accessed under Player model (ForeignKey), that for employer is for example: self.contract.wage and for worker: self.job\_to\_do.wage

I stored the whole object onto a LongStringField() under the Player object, using the values() function:

contract\_dump = self.contract.values()

Note that to work, you need to import some Django elements to the models.py file:

from django.db import models as djmodels

### Lesson 7: Interactive app: chatroom

The otree extensions, consumers.py and routing.py allow for interactions between participants. In the above application we do not use it up to the fullest.

The plan was for you to create a Chatroom application, but while I was coding it, I realized it’s a little bit too complicated. Instead, we can go through the application, it’s parts, and discuss modifications that we might want to execute.

For example, we can add time limits, purpose for the discussion, add decisions to be made…

### How to go forward with your own application?

Work together: even if you are working on different apps, an outside set of eyes is always useful.

Find applications that are already similar to yours. (Ask permission from the authors to use the code.) Modify.

Make small goals, break down the experiment into small parts.

Don’t get stuck. Ask for help.

Use print and console.log for finding errors.

Keep your old codes (and versions) for reference.

Pilot test for errors.

### Resources:

<https://otree.readthedocs.io/en/latest/index.html>

<https://www.w3schools.com/>

[oTree help & discussion](https://groups.google.com/forum/#!forum/otree) – Google Group – <https://groups.google.com/forum/#!forum/otree>

You can always Google for applications, GitHub is a huge code repository – you might find good starting places for your code