**Lesson 4**

What are the different data types possible? What are widgets?

**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:**

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

**Custom packages:**

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

**Task 3: 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”),

)

Then add the variable under player:

agree = RadioGridField(rows=ROWS, values=COLUMNS, require\_all\_fields=True, label=”Do you agree with the statements?”)

**Task 3: Another way: make your own Formfield?**

class MyFormField(models.IntegerField)

choices = [

[1, “agree”],

[2, “agree somewhat”],

[3, “don’t know”],

[4, “disagree somewhat”],

[5, “disagree],

]

class Player(BasePlayer)

question1 = MyFormField(label = “What is?”)

**Task 4: Computing variables**

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

Modify the public good game such that:

Remove contribution1-3 and add variable contribution under the Player class in the models.py file

Add total\_contributions variable under the Group class

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 \* self.group.total\_contibutions (notice unintential linebreak)

Or under Group:

def set\_payoffs(self):

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

player.payoff = Constants.endowment – player.contribution + 2 \* self.total\_contibutions (notice unintential linebreak)

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

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

self.group.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.set\_payoffs()

If stored under Player:

for player in self.group.get\_players()

player.set\_payoffs()

**Session variables**

**Some variables can be made session wide and easily modified by whoever is running the session. For example, treatments…**

Instructions for Exercise 1:  
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.

4. then, 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’]

The rest of the app:

Group size 2.

If the treatment is blue, show a blue hello text to the participants. If the treatment is red, show a red hello text.

Give a vote for each participant: red or blue.

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: we can discuss about the different solutions that people came up with.

Instructions for exercise 2:

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

(Just check that it works for 4 players first).

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 3>]]

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

original = self.get\_group\_matrix()

print(original)

print(original[0][0]]

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

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

Exercise 4. 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.

Exercise 5. Timeouts