Technische Universität München

TUM School of Management

Interdisciplinary Project

Programming on a web-based platform for conducting experiments using the example of dynamic bank runs

Supervisor:	Univ.-Prof. Dr. Robert K. Frhr. von Weizsäcker Lehrstuhl für Volkswirtschaftslehre Finanzwissenschaft und Industrieökonomik Technische Universität München
Advisor:	M.Sc. Christoph Gschnaidtner
Graduate program:	Informatics
Author:	Ozan Pekmezci Steinickeweg 7 App 313 80798 Munich 03636264
Submission date:	27. April 2017

München, 27. April 2017

Contents

List of Figures

Chapter 1

Introduction

This is a project to program an experiment on the oTree framework. The example of bank runs were chosen to implement by the advisors of the author. This chapter explains the logic of the framework and bank runs to the readers.

1.1 Introduction to oTree

oTree is an open-source framework that is designed to conduct interactive experiments on economics. It is based on Django which is a web framework that is designed for Python programming language.

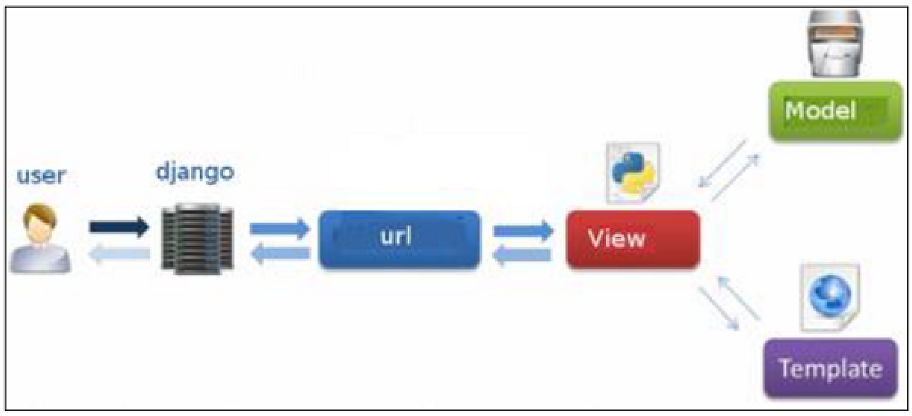


Figure 1.1: Abstraction of MVT Pattern[Django Overview (2017)]

The design pattern used by Django and also by oTree is called `MVT` [Django Overview (2017)]. MVT means `model-view-template` and that terminology explains what types of components the application consists of[Figure 1.1].

This pattern explains that there is a model component that is responsible for the backend in general. It generates the database fields and it is responsible for the business logic. It does calculations, saves values etc.

The template component consists of multiple `HTML` files that are called by the view. It determines how page should look like and what the page shows.

The view on the other hand is a connection between model and template. It sends the template required variables and also receives some information from the model. It is responsible for the working sequence of the application.

1.2 Concepts

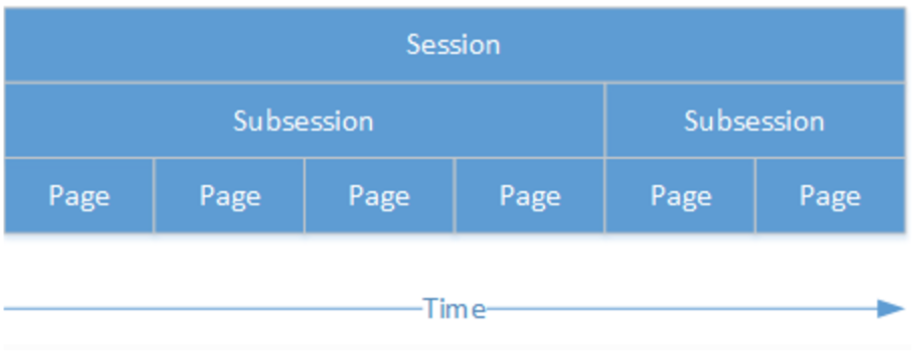


Figure 1.2: Conceptual Overview of oTree[oTree Concepts (2017)]

1.2.1 Session

A session can be thought as an whole experiment. It can consist of many subsessions. For example, a session can be a public goods game plus a questionnaire[oTree Concepts (2017)].

1.2.2 Subsession

Subsessions are part of the session as the name suggests. In the above example, public goods game is one subsession and the questionnaire is another subsession. Each subsession have multiple pages that is going to be explained in the views part[Section 1.3.2]

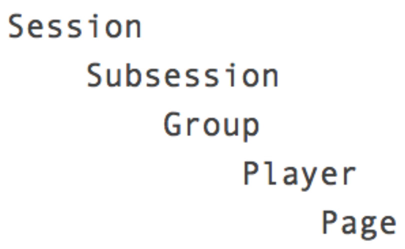


Figure 1.3: Object Hiearchy of oTree[oTree Concepts (2017)]

1.2.3 Group

A group consists of players that are taking part in the same subsession of the session. Therefore, a subsession is divided into a groups of players[Figure 1.3].

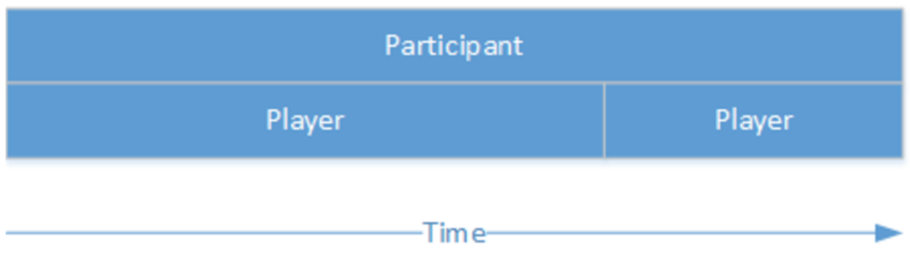


Figure 1.4: Concept of Participant Explained[oTree Concepts (2017)]

1.2.4 Participant and Player

Player in oTree is just an instance of the concept Participant. There is a different Player instance for every round and for each real-life player. These instances can have different payoffs and different attributes. However, each real-life player has one participant for each subsession and their attributes stay the same for the whole subsession[Figure 1.4].

1.2.5 self

As the readers check out the code, they will see the variable self very often. It is a Python-spefic instance that describes the context and can be compared to `this` in Java programming language. oTree functions will mostly expect self as a parameter and these functions can be called by command `self.functionName()`.

This means that if the method accepts self in the `Page` class, self refers to the Page object. Since the class Page is a subclass of the `Player` class[Figure 1.3], Player can be

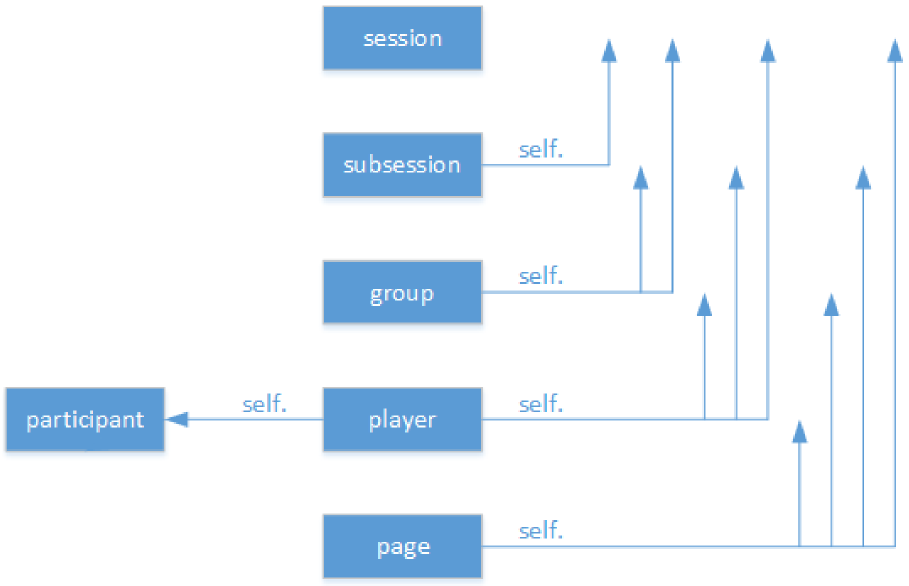


Figure 1.5: Reachable Objects via self[oTree Concepts (2017)]

referenced with `self.player`. This logic is also explained in Figure 1.5. Subclasses can refer to their superclasses but the opposite doesn't work. Extra to that, Player can refer to `Participant` class via self, that is also explained in the subsection 1.2.4.

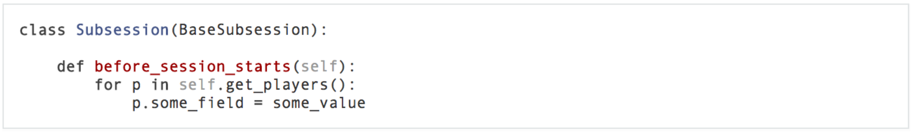
Furthermore, there are also many cases that the developers can refer from superclass to subclass. However, it is a bit different and can be explained easier with an example. Every player is part of only one group but each group consists of multiple players. That's why it is easy to refer to the group just by saying `self.group`, but players in a group need to be referred with `self.get_players()`, that will return a list of players. There are many similar examples on the official documentation of oTree[oTree Concepts (2017)].

1.3 Components

1.3.1 Model

The model component of MVT[Section 1.1] pattern is programmed in the file `models.py`. The file contains four classes: `Constants, Subsession, Group and Player`

Constants class is the place to put variables that don't ever change in the game. The class has three required fields, which are `name_in_url, players_per_group and num_rounds`. `name_in_url` is the name of the the app that is going to appear on the url. `players_per_group` defines how many players can be in each group. If it has the value `None`, all of the players will be in the same group. `num_rounds` describes the amount of rounds that the experiment will go on. It needs to have a static integer value. Constant values should never be modified to prevent unexpected behaviour[oTree Model (2017)].

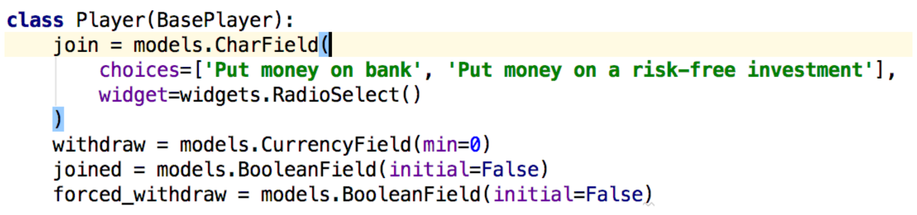


```
class Subsession(BaseSubsession):

    def before_session_starts(self):
        for p in self.get_players():
            p.some_field = some_value
```

Figure 1.6: An Example Usage of the Method[oTree Model (2017)]

In the context of oTree, subsession actually means a round. Therefore, this class contains round-specific information. Its method `before_session_starts` is important because it is the recommeded place to prepare and assign data to player for each round. However, it must be noted that this method is called once for each round `before the session starts`. That's why developers shouldn't use variables that change during the game. A typical usage is looping through players and doing some operations for them[Figure 1.6].



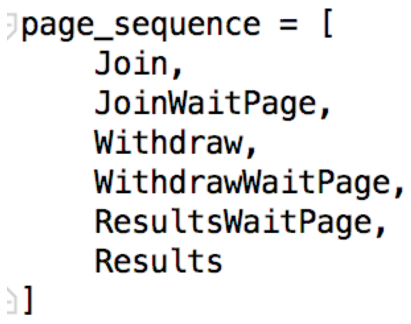
```
class Player(BasePlayer):
    join = models.CharField(
        choices=['Put money on bank', 'Put money on a risk-free investment'],
        widget=widgets.RadioSelect()
    )
    withdraw = models.CurrencyField(min=0)
    joined = models.BooleanField(initial=False)
    forced_withdraw = models.BooleanField(initial=False)
```

Figure 1.7: Player Class of the Bank Run Experiment

Model file also contains fields for the classes that are used to generate database columns for select tables. There are some special types of fields that are inherited from Django and can be used on the starting lines of classes. The field `models.CharField()` can be used for strings, `models.FloatField()` for floats(real decimal numbers), `models.BooleanField()` for boolean values, `models.IntegerField()` and `models.PositiveIntegerField()` for integers. oTree also supports an extra field `models.CurrencyField()` for currencies[Figure 1.7].

1.3.2 View

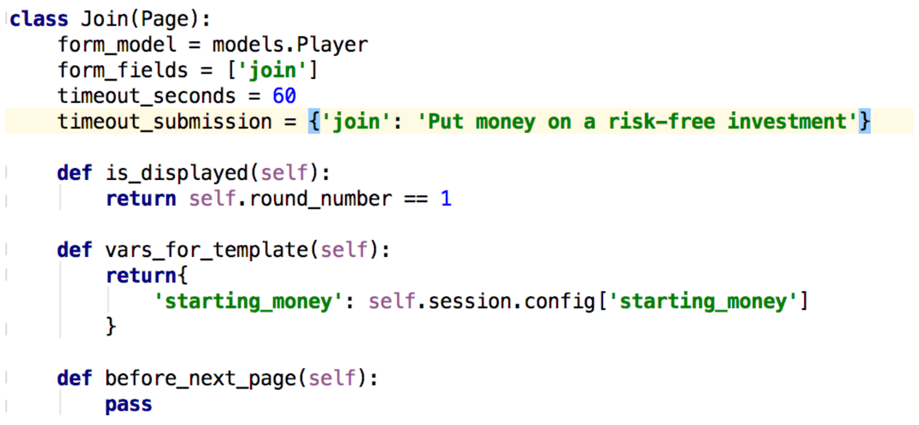
As it was explained in the introduction part of oTree[Section 1.1], the view component takes care of the application sequence. It also is the connector between model and template. The view file contains a class for each page that will appear and the sequence of



```
page_sequence = [
    Join,
    JoinWaitPage,
    Withdraw,
    WithdrawWaitPage,
    ResultsWaitPage,
    Results
]
```

Figure 1.8: Page Sequence List of the Bank Run Experiment

these pages are stated in the page_sequence list in desired order[Figure 1.8]. Pages can be divided into two general categories. They are pages and wait pages. Page classes expect the argument type `Page` and wait pages expect the argument type `WaitPage`.



```
class Join(Page):
    form_model = models.Player
    form_fields = ['join']
    timeout_seconds = 60
    timeout_submission = {'join': 'Put money on a risk-free investment'}

    def is_displayed(self):
        return self.round_number == 1

    def vars_for_template(self):
        return{
            'starting_money': self.session.config['starting_money']
        }

    def before_next_page(self):
        pass
```

Figure 1.9: A Page Class of the Bank Run Experiment

Page

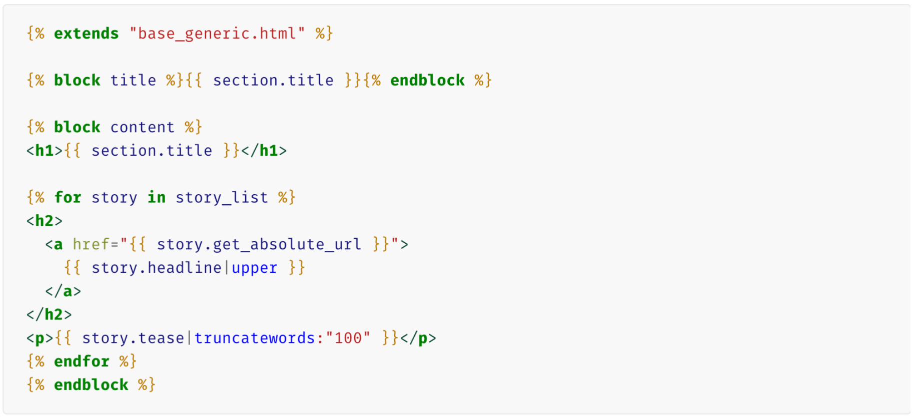
Pages contain information about the code that is going to run on client-side(users browser). Pages can describe the form that is shown on the page and can connect the server-side implementation of the form with the client-side implementation. These get set with `form_model and form_fields` variables[Figure 1.9].

There are also other attributes that can be used inside the page classes. For example, if user wants to set a timeout for users to respond to the form, developers can use `timeout_seconds and timeout_submission` variables. Former one sets the maximal time to respond in seconds and the latter one decides what is going to happen if the user doesn't response in the desired time.

Page classes can also contain some methods that are useful for many applications. One of them is `is_displayed(self)`; it determines in which condition users see the page. `vars_for_template(self)` method returns some custom variables to the template, `before_next_page(self)` method gives developer the option to do some operations after user responds to the form and navigates to the next page.

Wait Page

Wait Pages are used to make sure users start each part of the experiment at the same time. Users that respond fast wait until others respond to that form as well. Users see a loading page while waiting. An important method of the wait page class is `after_all_players_arrive(self)`. This determines what to do when all players reach the desired wait page. It can be used to call some functions in the models file. Developers can also choose to customize the appereance of the wait page[oTree View (2017)].



```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Figure 1.10: A Code Sample of a HTML File[Django Template (2017)].

1.3.3 Template

Templates are coded in HTML files and oTree uses Django's template language[Django Template (2017)]. HTML code can be written on the files directly but python code can be written in a different format. Variables are used in double brackets(`{{example}}`). Tags like `for loops, if clauses` are written between brackets and percent signs [Figure1.10].

1.4 Introduction to Bank Runs

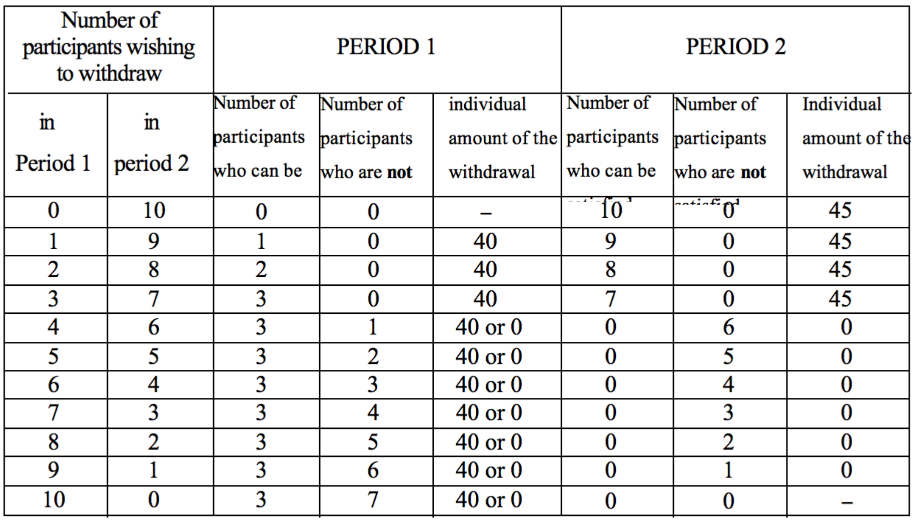
Banks that use fractional-reserve-banking system store only a small fraction of their assets as cash. However, this can be problematic if many people try to withdraw money in a short time. When more people withdraw money and withdrawing gains a momentum for some reason, bank can run out of cash and go bankrupt [Diamond (2007)]. At this point money of everybody that deposited to the bank is gone.

To prevent that, banks may limit money to withdraw or just cancel withdrawing for some time. This text is the documentation of a platform that is aiming to experiment how these bank runs happen.

The starting point of this project are the empirical studies that are conducted to test predictions of theoretical models and investigate the behaviour of people during withdrawal process. One of the empirical studies that is reviewed and tested by the author is `The Madies Study` [Madies (2007)].

1.4.1 The Madies Study

Madies sets an experiment environment with some defined amount of participants, periods, amount to withdraw and an upper limit for bankruptcy. Participants are people that take part in the experiment, periods mean withdrawing the money earlier or later, amount to withdraw differentiate between periods and bank going bankrupt. Lastly, upper limit is the amount of withdrawing people that make the bank go bankrupt. The time model of this experiment is discrete meaning that participants do actions simultaneously.



Number of participants wishing to withdraw		PERIOD 1			PERIOD 2		
in Period 1	in period 2	Number of participants who can be	Number of participants who are **not**	individual amount of the withdrawal	Number of participants who can be	Number of participants who are **not**	Individual amount of the withdrawal
0	10	0	0	–	10	0	45
1	9	1	0	40	9	0	45
2	8	2	0	40	8	0	45
3	7	3	0	40	7	0	45
4	6	3	1	40 or 0	0	6	0
5	5	3	2	40 or 0	0	5	0
6	4	3	3	40 or 0	0	4	0
7	3	3	4	40 or 0	0	3	0
8	2	3	5	40 or 0	0	2	0
9	1	3	6	40 or 0	0	1	0
10	0	3	7	40 or 0	0	0	–

Figure 1.11: Visualisation of the Madies Study[Madies (2007)]

The game is played for thirty rounds and in each round participants are asked if they want to withdraw in period one or two. Advantage of withdrawing on period 1 is the fact that the participants have higher chance to be able to withdraw their money. However, if they choose to withdraw their money on period 2, they will get their money plus interest if the bank didn't go bankrupt before[Figure 1.11].

In this particular environment, bank goes bankrupt if four or more people withdraw in period 1. First three participants can withdraw 40 unit currency and the rest get 0. In the bankruptcy case, participants that choose period 2 can't withdraw any money. If three or less participants choose to withdraw on period 1, they all get 40 unit currency in period 1 and the rest get 45 unit currency in period 2.

The problem with this study are the facts that, participants can only withdraw a static amount of money and bank goes bankrupt depending on amount of people instead of money amount. These drawbacks are considered and imporoved in our experiment which is explained in the next chapter.

Chapter 2

Bank Run Experiment

The aim of this project is experimenting with the oTree framework and test how it actually works. The topic of bank runs is chosen as en example for it.

Bank run experiment can be done in different settings or environments. In this one we want to use variables rather than constants. This means that we want to have modifiable number of player, rounds, thresholds etc.

The main logic is similar to the Madies Study which is an empirical study for bank runs[Section 1.4.1]. It has some more extra features that are going to explained in the upcoming sections.

2.1 Setting and Environment

Like it is said above, the bank run experiment uses as many changeable values as possible to give flexibility to the experimenters. Experimenters can find the optimal values to make correct observations.

This experiment simulates people that have a starting amount that is determined by the experimenter. However, all of the participants receive the same starting money. Participants can store their money in a risk-free investment or they can choose to put their money on the bank. Storing money on the investment has the advantage of not getting affected by a possible bankruptcy but depositing money to the bank gives you an interest per round. The interest is determined by the experimenter as well.

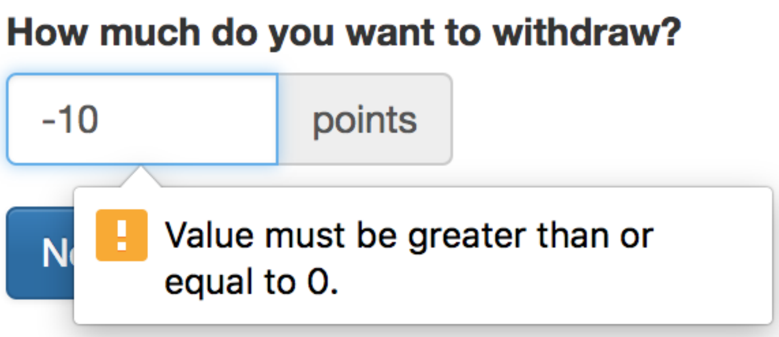
The game is played until the bank goes bankrupt or the last round is reached. The last round of the game is also set by the experimenter but it must be greater than two because the first round of the experiment is deciding if participants want to deposit money on a investment or to the bank. Starting from second round, participants decide if they want to keep their money at the bank or withdraw it. Some participants are forced to withdraw money from the bank. This is totally random and the experimenter can decide the percent of participants that will be forced to withdraw. Experimenters can also decide how much money that the chosen participants need to withdraw minimally.

If threshold of withdrawals is reached, the bank goes bankrupt and the game ends. This threshold can also be set by the experimenter. It is given as a percentage and the threshold is calculated by the multiplication of the percantage and total money that is deposited to the bank at the beginning of the game.

oTree has built-in support of changeable participant number value. No extra work is done for that. Lastly, experimenters can also choose between allowing to watch others or not. If its allowed, users can see current information about other users at the end of each round. Else, they can only see this information when the game ends.

2.2 oTree Platform

Initially, the experiment was planned to be played continuously instead of being discrete. According to the plan, game would have no rounds. Meaning that, users would be able to deposit or withdraw anytime and the application would react accordingly. However, continuous-time games were not part of oTree as of 2017 but the oTree team have stated that they plan to add this feature in the future[oTree Slides (2007)]. That's why the developer of this project enabled withdrawing dynamic amount of money. The game also has non-static number of rounds in contrast to the Madies Study[Section 1.4.1].


How much do you want to withdraw?

-10 points

Value must be greater than or equal to 0.

Figure 2.1: Form From Bank Run Doesn't Allow Negative Values

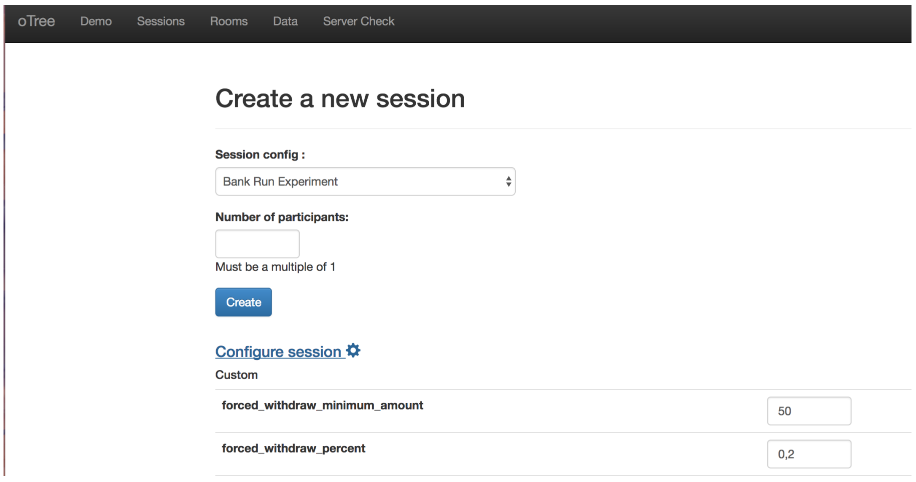
Withdrawals of dynamic values are trivial in oTree, since it requires only one `models.CurrencyField` field in models file and corresponding form fields in the view[Section 1.1]. CurrencyField can accept parameters for minimum and maximum values, which limits users for entering amounts in the desired range. This is really useful because this limitation occurs on the client-side and form only gets submitted when user enters a "correct" value. Problem is the fact that these min and max arguments can only be static values since the code is executed on the setup phase. That's why developers can't set arguments that depend on other values. In the context of bank run experiment, it was aimed to have a limitation between zero and the money that participant deposited before. Unfortunately, the developer couldn't set a max value because it would depend on the money at bank. Therefore, form accepts any value that is positive and the program limits unallowed entries on the backend[Figure 2.1].

It was also important for the experiments to have dynamic number of rounds. The two round system of the Madies Study is limited and doesn't allow much of a difference. However, it could be meaningful for the experimenters to see difference between the participant reactions in different cases. That's why the bank run experiment has variable number of rounds and experiment ends when the bank goes bankrupt or the desired number of rounds is reached. Sadly, oTree needs a static number for number of values in the `Constants` class[Section 1.1]. oTree needs that number, because the framework prepares the models before the game actually starts. It loops for each user and for each round. Only solution is actually a "hack" in computer science terms but it is also recommended on the official oTree documentation, so it is applied by the author of this text as well[oTree Model (2017)]. This method is setting the number of rounds to a big random value(`30` in our experiment). When the actually desired round number is reached, application hides the `next` button for the participants.

Most of the example projects for oTree don't have the concept of a storage like in the bank run experiment. Some of them have some kind of money pool but none of them have a concept similar to personal bank accounts. The basic principle of oTree experiments consist of the calculation of the `payoff` for each player and each round. In the end, sum of these payoffs are the final results. However, we have two distinct values for each player. These are money at hand and money at bank. They can't be merged together because money at bank can gain interest and also it can be nulled in case of bankruptcy. That's why, participant dictionary is used to store these two crucial values[Section 1.2.4]. Parcitipant dictionary stays same for the participant throughout the game but player fields change in each round. For example, money at hand can be reached with the `player.participant.vars["money_at_hand"]` for each participant.

A last concept that influenced the design of the software is the method `before_next_page`. It is located in page classes of the view file[1.3.2]. This function is important to order participants by their response time. In this experiment, we need to order participants by their order of withdrawal because bank can go bankrupt after withdrawal of each partcipant and the bank can't give people money if it has no money left. This function is called for each participant separately when they send responses on a specific page.

2.3 Workflow



Create a new session

Session config : Bank Run Experiment

Number of participants:

Must be a multiple of 1

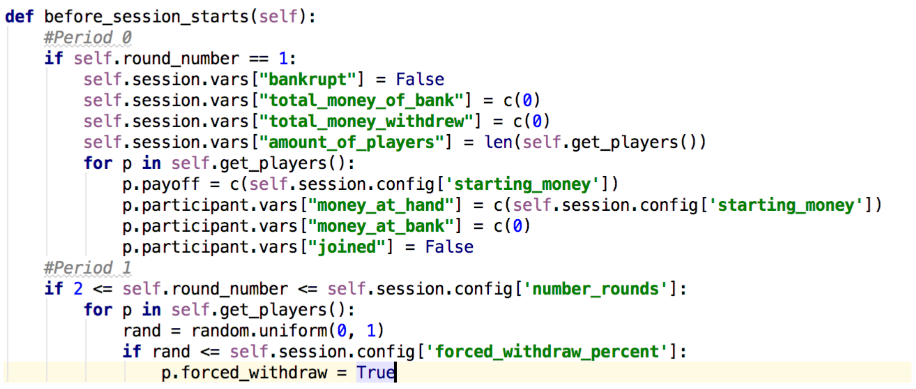
Configure session

Custom

forced_withdraw_minimum_amount	50
forced_withdraw_percent	0,2

Figure 2.2: Sessions Menu Item and Configuration Parameters

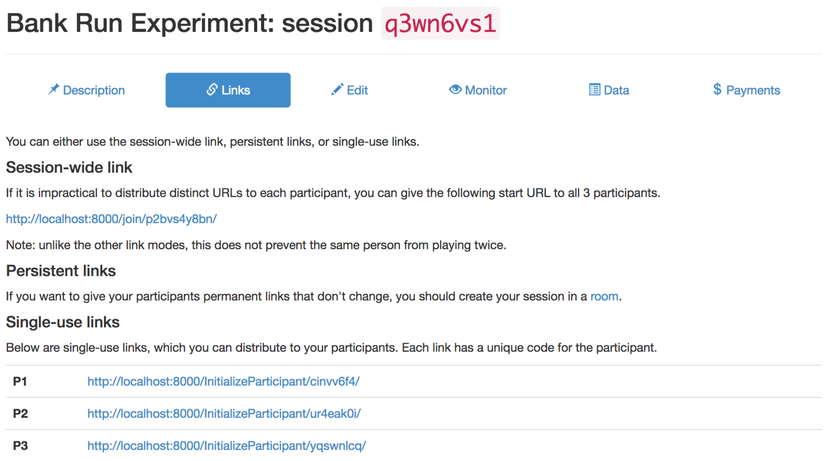
This section assumes that the experimenter has a working oTree server[See Appendix A]. The experimenter chooses `Sessions` on the menu and clicks on `Create new session`. Next page shows the screen to configure the session[Figure 2.2]. Experimenter needs to choose our bank run experiment and configure parameters however they want[Section 2.1]. After that the session gets created when the experimenter clicks on the corresponding button.



```
def before_session_starts(self):
    #Period 0
    if self.round_number == 1:
        self.session.vars["bankrupt"] = False
        self.session.vars["total_money_of_bank"] = c(0)
        self.session.vars["total_money_withdrew"] = c(0)
        self.session.vars["amount_of_players"] = len(self.get_players())
        for p in self.get_players():
            p.payoff = c(self.session.config['starting_money'])
            p.participant.vars["money_at_hand"] = c(self.session.config['starting_money'])
            p.participant.vars["money_at_bank"] = c(0)
            p.participant.vars["joined"] = False
    #Period 1
    if 2 <= self.round_number <= self.session.config['number_rounds']:
        for p in self.get_players():
            rand = random.uniform(0, 1)
            if rand <= self.session.config['forced_withdraw_percent']:
                p.forced_withdraw = True
```

Figure 2.3: Before Session Starts Function of Bank Run Experiment

During creation, the experimenter sees a loading screen. What oTree does is calling `before_session_starts`[Figure 2.3]. Developers use that method to setup their variables since it gets called for every round. In our experiment, this method first sets session dictionary values. Session values hold true for every round that's why it is the most logical place to store values that are about the session itself. After that, our function sets payoff values for each player just for the first round. Participant dictionary is also filled for every player. The reason for using participant dictionary is already explained multiple times in this documentation. Basically, player and group values are different for each round and it makes everything more complicated if the developer wants to store some consistent values like bank account of the participant. In its iterations for next rounds, it also determines participants that will be forced to withdraw their money randomly.



Bank Run Experiment: session q3wn6vs1

You can either use the session-wide link, persistent links, or single-use links.

Session-wide link

If it is impractical to distribute distinct URLs to each participant, you can give the following start URL to all 3 participants.

http://localhost:8000/join/p2bvs4y8bn/

Note: unlike the other link modes, this does not prevent the same person from playing twice.

Persistent links

If you want to give your participants permanent links that don't change, you should create your session in a room.

Single-use links

Below are single-use links, which you can distribute to your participants. Each link has a unique code for the participant.

P1	http://localhost:8000/InitializeParticipant/cinvv6f4/
P2	http://localhost:8000/InitializeParticipant/ur4eak0i/
P3	http://localhost:8000/InitializeParticipant/yqswnlcq/

Figure 2.4: Admin Dashboard of the Bank Run Experiment

Next, oTree shows admin page to the experimenter[Figure 2.4]. This dashboard is one of the best features of oTree and the experimenter can share links to participants, see participant data or advance some slow users. To start with the experiment, experimenter needs to share session-wide or single-use links with the participants.

When participants enter corresponding pages, they encounter the page that asks if they want to put their money on a risk-free investment or a bank. This is the first round of our experiment and the participants have a minute to answer. In the backend, `Join` page is first selected because it is the first element of `page_sequence` list in the view file[Section 1.3.2]. oTree calls `is_displayed` function of the join class decides to show to participants because it is the first round. `vars_for_template` function sends a value to the template and the template file named `Join.html` gets rendered. This file shows user the actual view on browser and the participant chooses an answer.

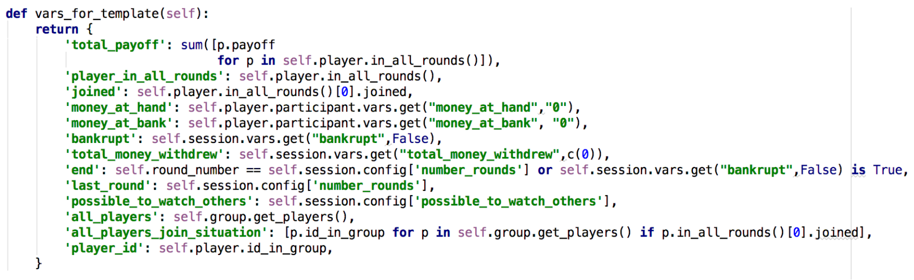
Participants who have answered, encounter a loading screen which is in fact the `JoinWaitPage`. This page is also only shown in the first round. When all players reach this stage, oTree calls automatically `after_all_players_arrive` function. After that, this function calls `set_join` method of the group model. The method takes care of participants responses and sets participant, session values, database field accordingly.

Next pages are `Withdraw` and its wait page. However, they are not displayed because their `is_displayed` function only allows if the current round number is bigger than two and smaller than the number of rounds that is specified in session config by the experimenter. This function also doesn't display page if the participant didn't put their money to the bank.

Following page is `ResultsWaitPage` but it also doesn't run because it only accepts rounds that are bigger than two. Last one is `Results` page. It is shown in every round and shows participants some data.

At this point, first round is finished and oTree starts over with the page sequence. As join and its wait page are only shown in the first round, our view goes on with withdraw page. Page sends variables to template, form is shown and user enters some amount to the form field. `before_next_page` function of the withdraw class is called for every participant when they respond. This step is important because the application checks if participant responded in the desired range and also checks each time if the bank went bankrupt. If that is the case, it doesn't allow participants to withdraw more money etc. Upcoming is the join wait page. After all players arrive, it calls `set_payoffs` function of the group model. As the name suggests, it is responsible for adding that rounds payoffs to each user. It also sets participant values for each user.

The rest is straightforward since result and its wait pages are shown again and their corresponding methods get called etc.



```
def vars_for_template(self):
    return {
        'total_payoff': sum([p.payoff
                             for p in self.player.in_all_rounds()]),
        'player_in_all_rounds': self.player.in_all_rounds(),
        'joined': self.player.in_all_rounds()[0].joined,
        'money_at_hand': self.player.participant.vars.get("money_at_hand","0"),
        'money_at_bank': self.player.participant.vars.get("money_at_bank", "0"),
        'bankrupt': self.session.vars.get("bankrupt",False),
        'total_money_withdrew': self.session.vars.get("total_money_withdrew",c(0)),
        'end': self.round_number == self.session.config['number_rounds'] or self.session.vars.get("bankrupt",False) is True,
        'last_round': self.session.config['number_rounds'],
        'possible_to_watch_others': self.session.config['possible_to_watch_others'],
        'all_players': self.group.get_players(),
        'all_players_join_situation': [p.id_in_group for p in self.group.get_players() if p.in_all_rounds()[0].joined],
        'player_id': self.player.id_in_group,
    }
```

Figure 2.5: Variables That Are Transferred to Result Template

Lastly, the game ends when the bank goes bankrupt or the desired round number is reached. This is secured with the `end` key in the results `vars_for_template` function[Figure 2.5]. If value of the `end` key is `True`, button to proceed is hidden and users see a notification that game did end.

The experimenter can check the data tab of the admin dashboard to see detailed information what participants chose and do some data analysis.

Chapter 3

Conclusion

It is totally possible to conduct experiments on oTree. It has lots of built-in features that were explained in this documentation before. The framework also continues to improve and add functions.

These experiments are also appropriate to perfom in any environment like a class or a meeting since everyone has a smart device at hand and the experimenter just need to share links with people.

To start developing on oTree, developers first need to learn the basics of Python programming language and Django framework. After that they can learn oTree from its own documentation and lastly can start programming with some example applications.

3.1 Evaluation of oTree Framework

oTree is a framework that makes many basic tasks for developing an experiment really easy. Functions like the admin dashboard or the currency field are already available and ready to use. It is also great that they use Python programming language and Django framework. Both are popular, stable and easy to learn. Other poisitive thing about oTree is its documentation. It is really well written and covers many topics with examples.

On the other hand, oTree also has some negative parts. It is developed with some use-cases in mind and if developers want to do something different, they will need to "hack" it. For example setting a round number to a static amount and hiding the button to proceed when the desired round number is reached.

All in all, it can be said that oTree is great for basic experiments and good for more complicated experimental settings. If the developer wants to conduct really complex experiments, it can be more logical to write their own software on top of Django.

3.2 Future Work

The developer of this project has tried to make the software as expandable as possible. The aim was sticking to the official documentation as much as possible to make sure future developers won't have problems with understanding or expanding the code.

The code and logic for this experiment can be used in many different experiments in connection with this documentation. Other experiments will probably have some similarities and code excerpts can be reused.

This experiment can also be improved. For example, visualisation was not one of the main concerns during the implementation of this project and developers can improve it just by changing template files. `HTML` is used for the templates and it is really easy to change or improve even with no coding experience.

In the future, oTree framework will also be improved and new features can also be applied on this experiment. Unfortunately, this experiment only supports discrete rounds but it will be more logical to adapt continous respond system when oTree adds such a feature.

Appendix A

Installing oTree

This chapter gives information how to install oTree on computer.

A.1 Installing Python

This sections explains how to install Python on different operating systems.

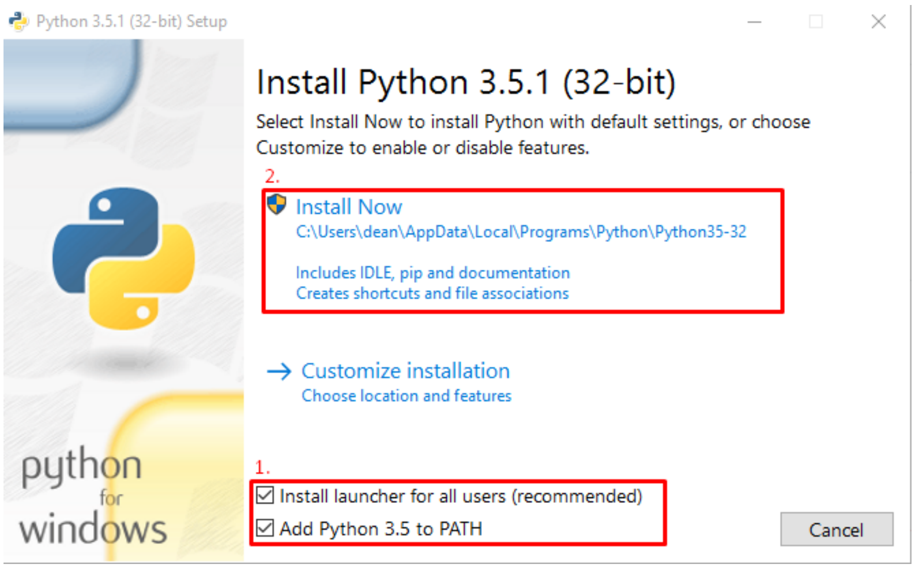
A.1.1 Installing Python on Windows

Windows users can install Python with the setup file that can be downloaded from `https://www.python.org/downloads/release/python-360/`. Users need to choose marked fields(A.1) and the installer will install both python and required tool `pip` to download other packages.

After that users can navigate to `PowerShell` application and test the installation by typing `pip3 -V`. If it shows the right version number, user download other packages etc.

A.1.2 Installing Python on Mac

The official oTree documentation recommends using `Homebrew` package manager on MacOS to install Python. However, mac users need to install `Command line developer tools` first, which is going to install many essential libraries and programs to start developing on mac. After that users can download Homebrew and lastly install Python. Users can test the installation by typing `pip3 -V` on the Terminal just like on Windows [oTree Installation (2017)].



Python 3.5.1 (32-bit) Setup

Install Python 3.5.1 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

2. Install Now
C:\Users\dean\AppData\Local\Programs\Python\Python35-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

1. ☑ Install launcher for all users (recommended)
☑ Add Python 3.5 to PATH

Figure A.1: Python Installation Screen on Windows

A.1.3 Installing Python on Linux

Python can be installed easily on a Debian/Ubuntu based Linux distribution by typing the command `sudo apt-get install python3-pip` on Terminal. The installation can also tested with the same command like on other operating systems.

A.2 Installing oTree

Users can install otTree easily by typing the command `pip3 install -U otree-core` on Powershell(for Windows) or on Terminal for Unix based operating systems. The same command is also used to upgrade the oTree versions in the future.

A.3 Running oTree

Users that run oTree for the first time need to type in the command `otree startproject oTree`, which will generete an `oTree` folder containing necessary files and example projects. After that users can navigate to the newly generated folder by using the command `cd oTree`.

When you are in the right folder, users first need to migrate and reset the database. This is done with the command `otree resetdb` and it needs to be applied everytime when there is a change in the model file[Section 1.3.1]. Finally, users can run the server with command `otree runserver`. To stop the server, pressing on both `ctrl and c` keys is enough. On the same terminal tab, oTree states the local address that the oTree is running on, which is mostly `http://127.0.0.1:8000/`. Users can visit this link on their own browser.

A.4 The Development Environment

Although Python code can be written on any text editor even on Terminal, the IDE `PyCharm` is recommended by both the official documentation and the author of this documentation. It provides autocompletion and also makes it much harder to make errors. PyCharm has both free and paid versions but students or teachers(including teaching assistants) can get the paid version for free. The paid version features Django support that the oTree is built on. Developers just need to import the oTree folder, enable Django support on the settings and set the root folder of oTree as `Django project root` .

A.5 Using oTree on a Production Environment

This documentation mainly focuses on using oTree on development environment which is suitable for testing and research purposes. However, if experimenters will want to use production when they conduct a real study.

Developers of oTree knows this as well and they have prepared guides to help users run oTree on a production server. This guide can be found on under the `Server Check` tab when running oTree on browser. That screen shows users some steps to get their production environment ready. More detailed information can be found on their website[oTree Server (2017)].

The author of this documentation has also tried running server on `Heroku` hosting platform to test and encountered no problems while doing that. However, with the free version of Heroku it is way too slower than running the server locally. That's why it is recommended to invest in paid versions.

A.6 Troubleshooting

A.6.1 Developer needs to have multiple python versions installed

Some developers may have an issue that they need to have a one specific Django/Python version for a not oTree related project and another one for the oTree. In that case users won't be able to run one of the projects or they will need to install/uninstall every time. However, there is a solution for that.

Developers can install the Python package `virtualenv` just with `pip install virtualenv` command. Like the name suggests, this little program creates virtual environments for each folder structure so that users can install different python/Django or any other package without interfering with each other. To use it, developers must navigate to the desired folder and type the command `virtualenv env` which will create the environment `env` folder. After that developers need to type the code `source env/bin/activate` every time they want to run the environment.

A.6.2 Developer needs to use dictionary values in template files

Normally dictionary values can be reached with commands like `model["key"]` on Python. However, Django's syntax for that in template files is like `model.key`.

Bibliography

Chen, D.L., Schonger, M., Wickens, C., 2016. oTree - An open-source platform for laboratory, online and field experiments. Journal of Behavioral and Experimental Finance, vol 9: 88-97

Heston, S. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. The Review of Financial Studies, Volume 6, Issue 2, pp. 327-343

Sharpe, W. (1964). Capital Asset Prices – A Theory of Market Equilibrium Under Conditions of Risk. Journal of Finance, Vol. 19, pp. 77-91

oTree. (2017). Official Documentation of oTree, `otree.readthedocs.io`

oTree. (2017). Official Documentation of oTree, `otree.readthedocs.io/en/latest/install.html`

Tutorialspoint. (2017). Django Tutorial, `https://www.tutorialspoint.com/django/django_overview.htm`

oTree. (2017). oTree Concepts, `http://otree.readthedocs.io/en/latest/conceptual_overview.html`

oTree. (2017). oTree Model, `http://otree.readthedocs.io/en/latest/models.html`

oTree. (2017). oTree View, `http://otree.readthedocs.io/en/latest/view.html`

Django. (2017). Django Template, `https://docs.djangoproject.com/en/1.8/ref/templates/language/`

Diamond, D. (2007). Banks and Liquidity Creation: A Simple Exposition of the Diamond-Dybvig Model Economic Quarterly, Volume 93, Number 2, pp. 189-200

Philippe Madi‘es. Self-fulfilling bank panics : how to avoid them ? an experimental study. Working Paper du GATE 2001-04. 2001. ¡halshs-00179997¿

Chris. oTree:An open-source platform for lab,web and field experiments.

oTree. (2017). oTree Server,
`http://otree.readthedocs.io/en/latest/server/intro.html`