Repeating it & Getting Paid

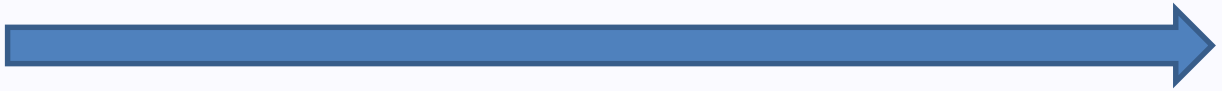
Remixed from material by Ali Seyhun Saral & Philipp Chapkovski

Repeating the choice

- Sometimes participants have to take the “same” decision multiple times
  - Repeated Prisoners Dilemma or Public Goods Game
  - Real effort tasks
  - Multiple periods in a market game
  - …
- Sometimes there are multiple decision problems that are similar but differ slightly
  - Lottery choices
  - Investment decisions
  - Intelligence test parts (Raven matrices)
  - ….

Repeating the choice

- oTree apps can be played multiple rounds
- Each round repeats the page sequence



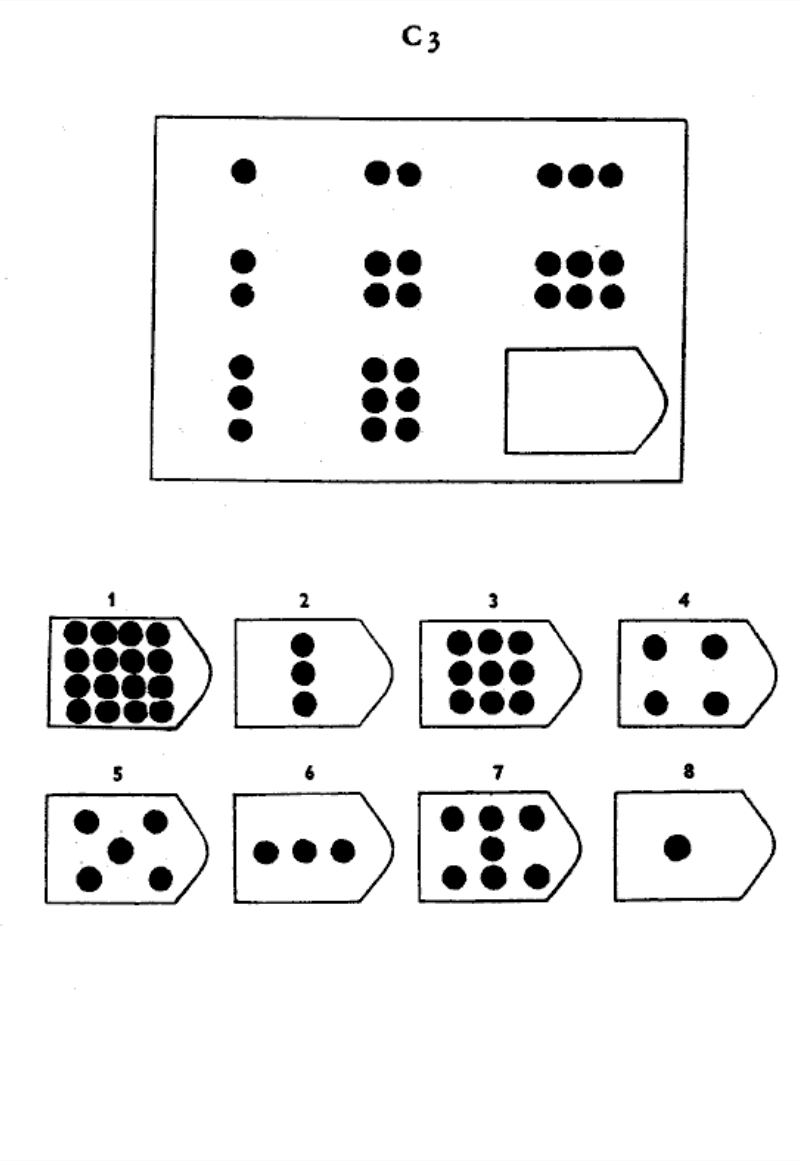
page_sequence = [WelcomePage, Ask, Results]

In this lecture:

- How to design such apps
- Best practices
- Build-in methods to use
- Common pitfalls
- How to show static files

Running example: Raven's Progressive Matrices

- Used to test fluid intelligence
- Different matrices across the experiment
- Different picture for each decision
- Correct answer differs from matrix to matrix
- Underlying structure is the same for each matrix



Running example: Raven's Progressive Matrices

Naïve solution:

- One giant app with one specific page for each matrix
- Disadvantages:
  - Lots of repeated code
  - Error prone
  - Bad practice

Better solution:

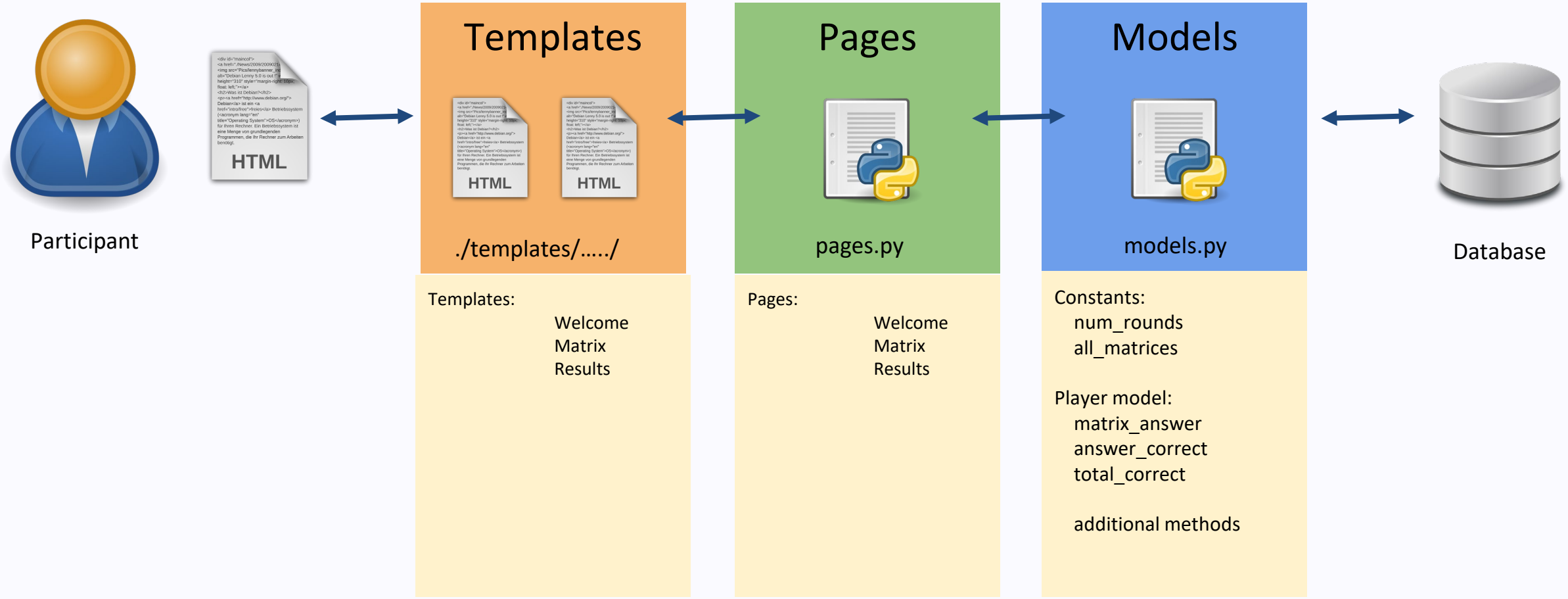
- Use multiple rounds with the same page that changes
- One page that shows a different matrix for each round

Solution we won’t cover:

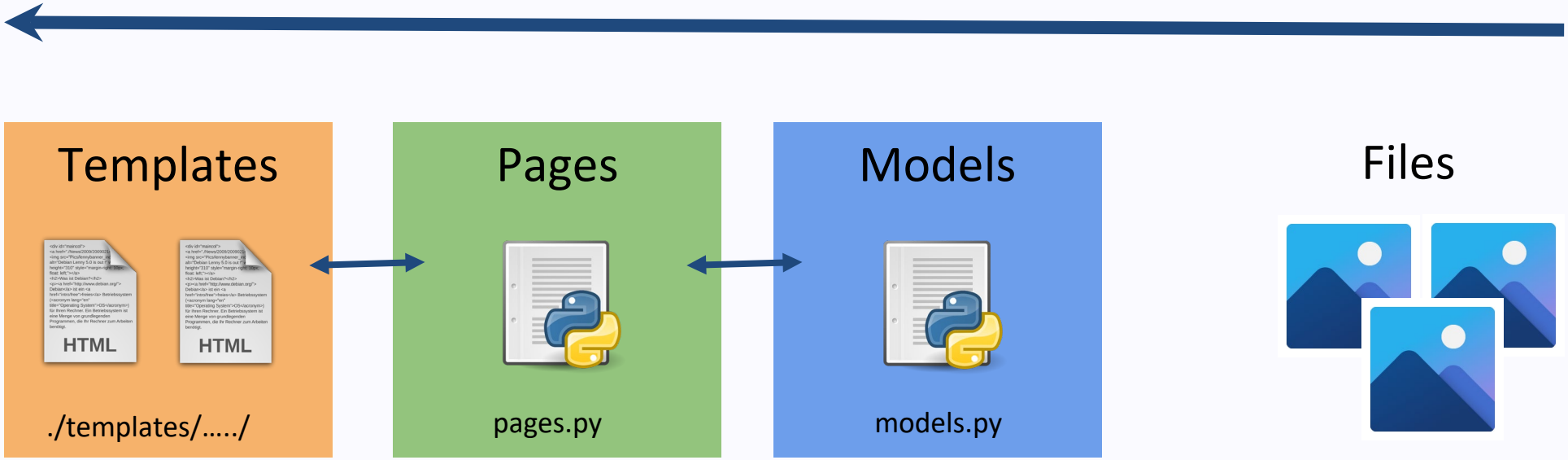
- Use JavaScript & oTrees live pages feature

Raven's Progressive Matrices

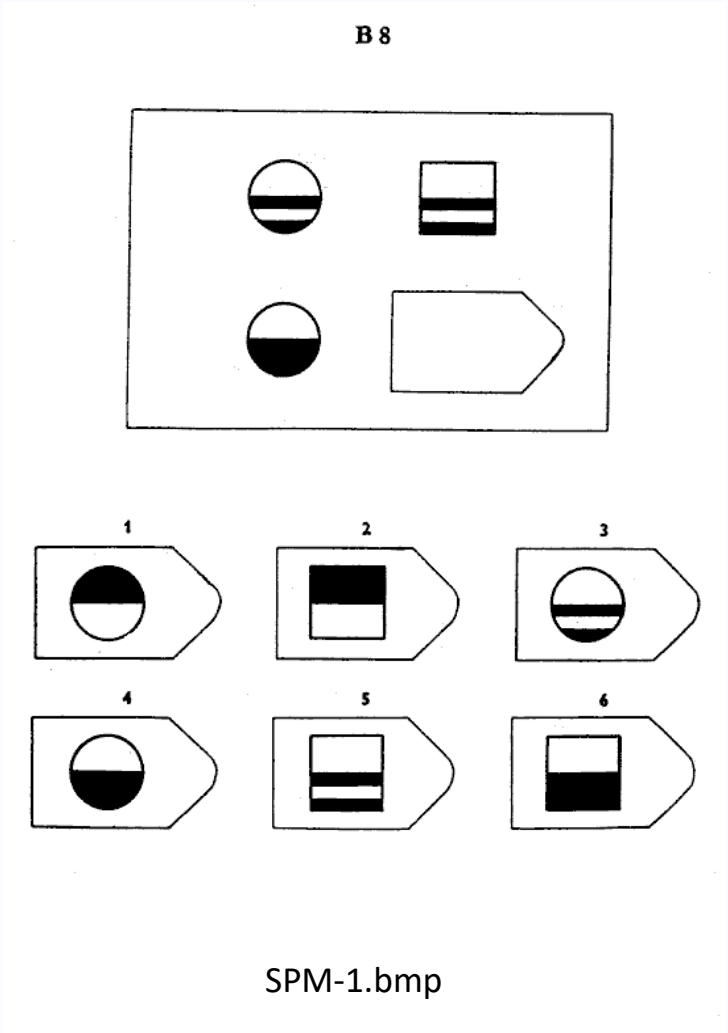
Task: Program a RPM app.
Page 0: Welcome Page
Page 1: Show the Matrix for the current round and record the answer
Page 2: Show the final result



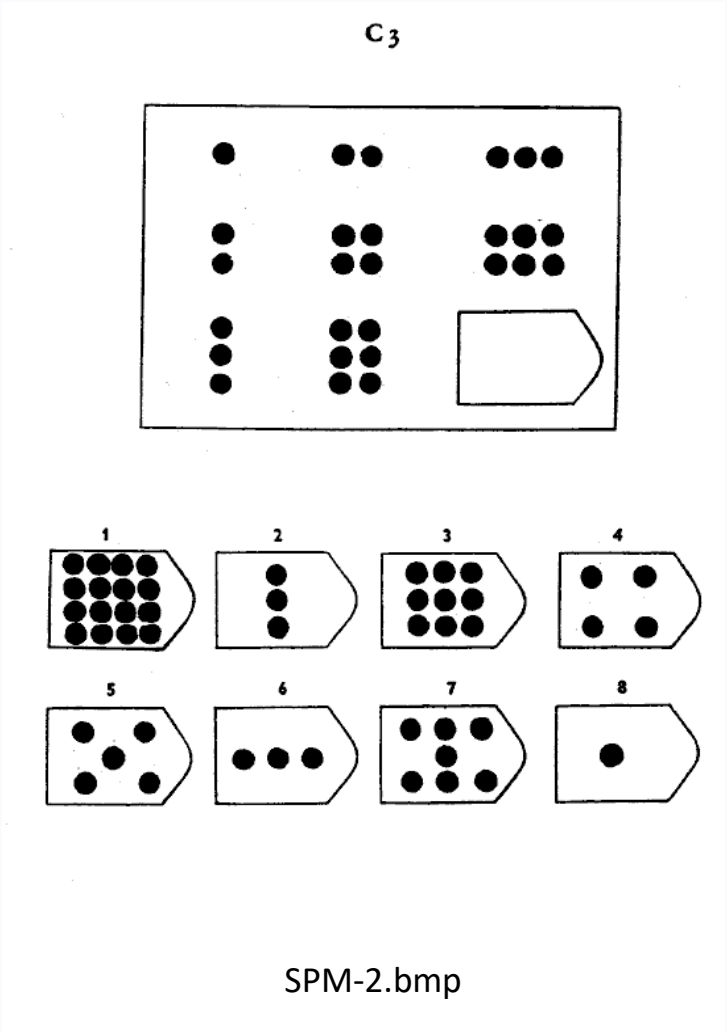
Planning for building the app



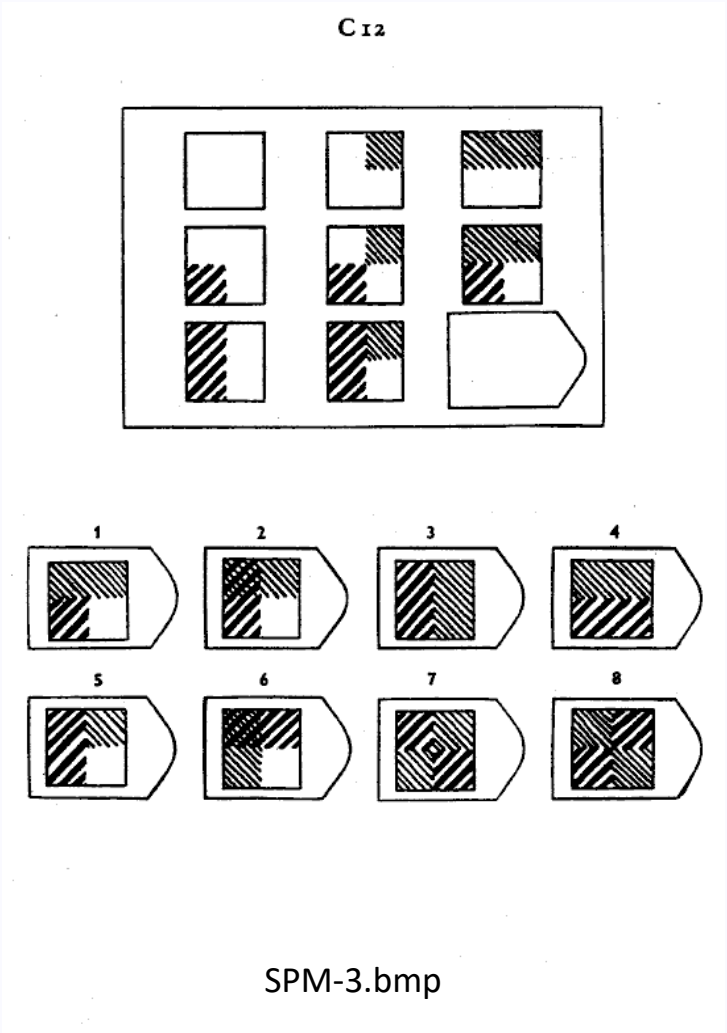
Matrices as image files



SPM-1.bmp



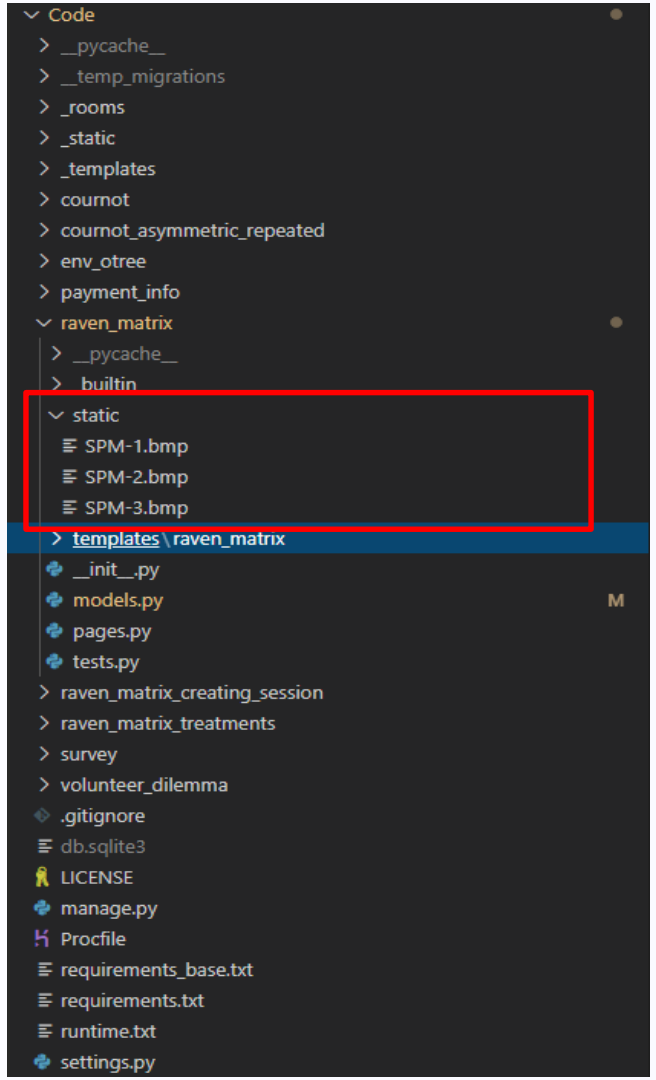
SPM-2.bmp



SPM-3.bmp

Where to store static files?

- All *static* files should be stored in the ./static folder of an app
- We can use access folder in the html file later by using the static-tag
  - E.g. {% static "./SPM-1.bmp" %} in the template
- Same folder if you store other files (csv-files etc.)



Building plan - RPM

Models

models.py

Manages Data Structure

(Additional) Constants:

- **Group size (?)**
- **Number of rounds**
- **Payment per matrix**
- **Some structure to save info about matrices**

Player class

- **…**

Building plan - RPM

Models

models.py

Manages Data Structure

Constants

- **name_in_url (String)**
- **players_per_group**
- **num_rounds (Integer)**
- **payment_per_matrix (Currency)**
- **all_matrices (Dictionary)**

```
class Constants(BaseConstants):
    name_in_url = 'raven_matrix'
    players_per_group = None
    num_rounds = 3 # Number of matrices
    payment_per_matrix = c(10)
    all_matrices = ...
```

Building plan – RPM – Dictionaries to store data

- Dictionaries useful to store (nested) data
- All matrices have a unique key (Round number)
- Each value is another dictionary with all information for the respective matrix
- Other possibilities here?

```
class Constants(BaseConstants):
    name_in_url = 'raven_matrix'
    players_per_group = None
    num_rounds = 3
    payment_per_matrix = c(10)
    all_matrices = {
        1: {
            'file': 'SPM-1.bmp',
            'number_of_answers': 6,
            'correct_answer': 6,
            'id': 'B8'
        },
        2: {
            'file': 'SPM-2.bmp',
            'number_of_answers': 8,
            'correct_answer': 3,
            'id': 'C3'
        },
        3: {
            'file': 'SPM-3.bmp',
            'number_of_answers': 8,
            'correct_answer': 2,
            'id': 'C12'
        }
    }
```

Building plan - RPM

Models

models.py

Manages Data Structure

Player class

- **matrix_answer: Integer (IntegerField)**
- **total_correct: Number (IntegerField)**
- **answer_correct: True/False (BooleanField)**
- **Methods that are defined on Player level**

```
class Player(BasePlayer):
    matrix_answer = models.IntegerField(label="Your answer:")
    answer_correct = models.BooleanField()
    total_correct = models.IntegerField()
```

Where are the choices for martix_answer?

Note: Created automatically for each round!

Building plan - RPM

Models

models.py

Manages Data Structure

Player methods

- **Check_answer() [Is the answer correct?!]**
- **Marix_answer_choices() [Which choices do I have?]**
- **Set_payoff() [What is the payoff]**

```
class Player(BasePlayer):
    matrix_answer = models.IntegerField(label="Your answer:")
    answer_correct = models.BooleanField()
    total_correct = models.IntegerField()

    # Dynamic form field validation
    def matrix_answer_choices(self):
        choices = list(range(1, Constants.all_matrices[self.round_number]['number_of_answers'] + 1))
        return choices


    def check_answer(self):
        if self.matrix_answer == Constants.all_matrices[self.round_number]['correct_answer']:
            self.answer_correct = True
        else:
            self.answer_correct = False

    def set_payoff(self):
        self.total_correct = sum([p.answer_correct for p in self.in_all_rounds()])
        self.payoff = self.total_correct * Constants.payment_per_matrix
```

RPM choices in each round

```
# Dynamic form field validation
 def matrix_answer_choices(self):
    choices = list(range(1, Constants.all_matrices[self.round_number]['number_of_answers'] + 1))
    return choices
```

- Number of answer choices differs by matrix
  - Dynamic form field validation to change the choices for each round/matrix
- {formfield}_choices
  - Must return a list of choices
  - Could also vary it by treatment/role etc

Checking the answer in each round

```
def check_answer(self):
    if self.matrix_answer == Constants.all_matrices[self.round_number]['correct_answer']:
        self.answer_correct = True
    else:
        self.answer_correct = False
```

- Correct answer differs by round
- We want to verify the answer to be able to calculate the profit at the end
- Note:
  - The field answer_correct will never be seen by the participant but only manipulated by us in the background

Checking the answer in each round

```
def set_payoff(self):
    self.total_correct = sum([p.answer_correct for p in self.in_all_rounds()])
    self.payoff = self.total_correct * Constants.payment_per_matrix
```

- Function to be used at the end of the experiment to calculate the final payoff
- By convention this function is called set_payoff()
- Payoffs will be saved in the *payoff* variable
  - Note that this is pre-defined by oTree -> We do not need to define it

Pages

Pages

pages.py

Manages “backend” of the pages

Page class: Welcome page

- **Only show in the first round**
- **Show payment info**

Page class : Matrix page

- **Set up the form matrix_answer**
- **Check the answer**

Page class : Results page

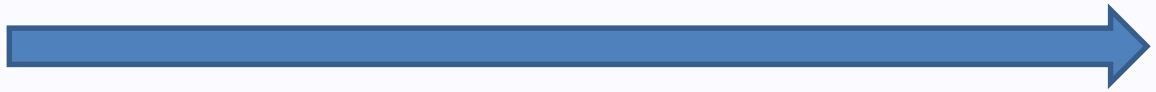
- **Only show in the last round**
- **Show final payoff**

```
class Welcome(Page):
    pass

class Matrix(Page):
    pass


class Results(Page):
    pass
```

Page sequence repeated in each round



```
page_sequence = [Welcome, Matrix, Results]
```

- Each round repeats the page_sequence
- Problem:
  - Welcome page should be only shown in the first round
  - Matrix page in every round
  - Results only in the last round
- Solution:
  - Use oTrees is_displayed() method
  - Governs if a page is shown or not
  - Can be used for rounds or also roles (e.g. sender/receiver in trust game)

Welcome Page in the first round

```
class Welcome(Page):

    # Welcome Page only shown in the first round
    def is_displayed(self):
        if self.round_number == 1:
            return True
        else:
            return False
```

- Use self.round_number to retrieve current round number and condition on it
- If is_displayed() returns True, the page is shown in this round

Showing additional variables

- Constants, player/group variables can be easily retrieved in templates by {{player.variable}}
- What if we would like to access other information in the template?
  - Show-up fee
  - Conversion rate
  - Intermediate values that are not saved to the DB
  - …
- Use the vars_for_template() method
  - Returns a dictionary with key-value pairs
  - Key: Variable name to be used in template
  - Value: Value of the variable we want to show

vars_for_template() method

```
class Welcome(Page):

    # Welcome Page only shown in the first round
    def is_displayed(self):
        if self.round_number == 1:
            return True
        else:
            return False

    def vars_for_template(self):
        return {
            'real_world_currency_per_point': self.session.config['real_world_currency_per_point'],
            'participation_fee': self.session.config['participation_fee']
        }
```

- Conversion rate and show-up fee stored in session-config
  - You can change it in settings.py or when you start an experiment
- {{self.session.config['real_world_currency_per_point']}} not possible in template
- After using vars_for_template() I can use {{participation_fee}} and {{real_world_currency_per_point}} in the template

Matrix Page

```
class Matrix(Page):
    form_model = 'player'
    form_fields = ['matrix_answer']

    def vars_for_template(self):
        # Compose the link to the files
        # oTree will look automatically in the *static* folder
        imgPath = './' + Constants.all_matrices[self.round_number]['file']

        return {
            'imgPath': imgPath
```

- Define form_model and form_fields:
  - form_model : tell which model you use from your models.py
  - form_fields : tell which fields you want the input from
- We have to tell oTree where to look for the matrix image

- Check **AFTER** the subject answered the matrix if the answer is correct
  - Where to do this?!

before_next_page()

```
class Matrix(Page):
    form_model = 'player'
    form_fields = ['matrix_answer']

    def vars_for_template(self):
        ...
    def before_next_page(self):
        self.player.check_answer()
        if self.round_number == Constants.num_rounds:
            self.player.set_payoff()
```

```
def check_answer(self):
    if self.matrix_answer == Constants.all_matrices[self.round_number]['correct_answer']:
        self.answer_correct = True
    else:
        self.answer_correct = False
```

- Executed AFTER the subject submits a page
- Useful for all calculations that have to be made conditionally on player choices on this page
- Call defined player methods (Round differ in the method)
- Also note:
  - before_next_page() is only executed once (upon submission of the page)
  - vars_for_template () each time I reload the page (when pressing F5)
  - Important when doing random draws etc

Results Page

```
class Results(Page):

    def vars_for_template(self):
        return {
            'money': self.player.payoff.to_real_world_currency(self.session)
        }

    def is_displayed(self):

        # Results page is only shown in the last round
        if self.round_number == Constants.num_rounds:
            return True
        else:
            return False
```

- Final profit can be calculated by self.player.payoff.to_real_world_currency(self.session)
  - Built-in oTree function
- Use again is_displayed() as we only want this page to appear in the last round

Templates

Templates

Manages “frontend” of the pages

- **Welcome.html**
  - **Show variables (show up fee etc)**
- **Matirx.html**
  - **Show matrix image**
  - **Formfield for matrix answer**
- **Results.html**
  - **Show final payoff**

Welcome.html template

```
{% extends "global/Page.html" %}
{% load otree static %}

{% block title %}
   Welcome
{% endblock %}

{% block content %}
   For each pattern that you complete correctly, you will receive
   <b>{{Constants.payment_per_matrix}}.</b><br>
   One point corresponds to {{real_world_currency_per_point}} €.<br>

   <div id='p_fee'>
   <p>
   You receive {{participation_fee}} for showing up today.
   </p>

   </div>
   {% next_button %}


<style>
#p_fee {
   color: #7FFFD4;
}

</style>
{% endblock %}
```

- Variables that we entered in vars_for_template() in pages.py can be used
- Different HTML elements can be used
- Examples here:
  - <b>-tag: Bold text
  -
-tag: Break line
  - <p>-tag: Paragraph
  - tag: Container in html which is styled by CSS or manipulated by JavaScript
  - <style>-tag:
    - Define style information with CSS
    - Lots of flexibility here

Template

Matrix.html

```
{% extends "global/Page.html" %}
{% load otree static %}

{% block title %}
   Please complete the picture below
{% endblock %}

{% block content %}
   <img width="50%" height="50%" src="{% static imgPath %}" />

   {% formfields %}

   {% next_button %}

{% endblock %}
```

Results.html

```
{% extends "global/Page.html" %}
{% load otree static %}

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

{% block content %}
   You answer {{player.total_correct}} matrices correctly.<br>
   Each correct matrix gives you {{ Constants.payment_per_matrix}}.<br>
   Your payoff is {{ player.payoff }}. <br>
   This corresponds to {{money}}.
{% endblock %}
```

Run the App

Look into DB

Improving the RPM app

- Matrix information in Constants class but not in the database
- By looking at the data alone we do not know which matrices have been answered by the participant
  - Problem for reproducibility & if we consider treatment variations
- New objective:
  - Extend the existing app
  - Save the information for each matrix for each round in the database

Extending the exisiting models.py

```
class Player(BasePlayer):
    # Save the input also in the database
    matrix_file = models.StringField()
    matrix_id = models.StringField()
    matrix_correct_answer = models.IntegerField()
    matrix_number_of_answers = models.IntegerField()

    # Fields for response related data
    matrix_answer = models.IntegerField(label="Your answer:")

    ...
```

- We need fields to save the data from Constants to the database
  - matrix_file: File name (Text)
  - matrix_id: ID of the matrix (Text)
  - matrix_correct_answer: Correct answer (Integer)
  - matrix_number_of_answers: Number of possible answers (Integer)
- Other model fields remain untouched

Creating_session()

```
class Subsession(BaseSubsession):
    def creating_session(self):
        pass
```

- Subsession model has a special method creating_session
- The code there will be executed just before the session starts.
- **It will be executed as many times as many rounds the game has.**
- It is used for assigning initial values, randomizing things and assign treatments
- Subsession model also has two other methods:
  - get_players()
  - get_groups()
- Helpful in our case to assign the matrix values to each player for each round and save it to the database

Creating_session()

```
class Subsession(BaseSubsession):
    def creating_session(self):
        all_players = self.get_players()

        # Python 3.7+:
        # Dictionary iteration order is guaranteed to be in order of insertion.
        for p in all_players:
            p.matrix_file = Constants.all_matrices[self.round_number]['file']
            p.matrix_id = Constants.all_matrices[self.round_number]['id']
            p.matrix_correct_answer = Constants.all_matrices[self.round_number]['correct_answer']
            p.matrix_number_of_answers = Constants.all_matrices[self.round_number]['number_of_answers']
```

- Self.get_players() returns a list of all players in the subsession
- In each round it saves the matrix information to each player
- Those variables can then also be used in the rest of the app in each round

“New” variables used in the app

```
# Dynamic form field validation
def matrix_answer_choices(self):
    choices = list(range(1, self.matrix_number_of_answers + 1))
    return choices

def check_answer(self):
    if self.matrix_answer == self.matrix_correct_answer:
        self.answer_correct = True
    else:
        self.answer_correct = False

def set_payoff(self):
    self.total_correct = sum([p.answer_correct for p in self.in_all_rounds()])
    self.payoff = self.total_correct * Constants.payment_per_matrix
```

- Self.varname always references the value of the variable in the given round (Subsession)
  - Remember Player object is owned by group which is owned by subsession

Other round related function we have not used

- Player, group, and subsession objects have the following methods:
  - in_previous_rounds()
  - in_all_rounds()
  - in_rounds()
  - in_round()
- Self.participant.vars
  - Dictionary that is persistent across apps and rounds
  - Useful to store any type of data that you want to access in a later stage
  - Possible to store other data types like list/tuples
  - Useful if you want to randomize something in the first round and then use this already randomizied object later (For your assignment)
- Self.session.vars
  - For global variables that are the same for all participants in the session

- IMPORTANT: Session.vars and participants.vars are NOT stored in the database

Run the App

Look into DB