Programming and conducting experiments

Università degli Studi di Torino
Dipartimento di Culture, Politica e Società

Matteo Ploner (UNITN)

30 October, 2020

Programming and managing experiments in oTree

1 Our course

1.1 Me

- I work at the department of Economics and Management of the University of Trento as Associate Profesor
- I am deputy manager of the Cognitive and Experimental Economic Laboratory (CEEL)
- My research interests are in **Experimental and Behavioral Economics**
- My home page
    - https://matteoploner.eco.unitn.it/
    - To contact me: matteo.ploner@unitn.it
- I like analyzing data 🗎, riding bikes 🚵 and snowboarding 🏂

1.2 Description 🗎

- Essential tools for setting up economic experiments on a web platform developed in the oTree environment.
    - The experiments can be conducted both in a controlled laboratory environment and remotely via the Web.
- Introduction to the oTree software for designing and conducting online experiments.
    - Participants will be guided in the experiment’s management, from the planning phase to the data collection and management phase.
- Hands-on approach
    - Assignments
    - Participants need to develop a project

1.3 Contents 🗃

- 6 interdependent modules
    - Practical examples will be provided, and participants will be asked to experience the experimental situations firsthand.

1. Programming and management of experiments in oTree

2. Fundamental components of an experiment
3. Questionnaires and individual decision making
4. Strategic interaction
5. Graphical elements and online implementation
6. Participants' projects

1.4 Materials

- Lecture material will be available on course's Moodle platform
- (Online) Lectures
    - Meeting link: https://unito.webex.com/unito-en/j.php?MTID=ma364c81cb21f96264fdbdd81515e709d
    - Meeting number: 121 216 6783
    - Password: f6DJbPyNQ22
- Slides
    - Available in .pdf shortly before the lecture on Moodle
- Code
    - Available after the lecture on Moodle
    - code available live in a github repository

2 Economic experiments

2.1 Think carefully

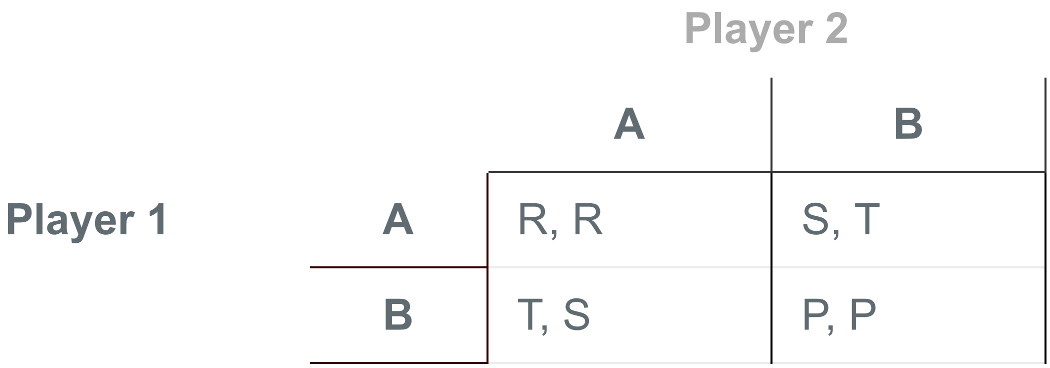
Enzo Mari (1957)

> Un giorno, dunque, mi faccio venire la folle idea di realizzare un puzzle con 16 animali, tutti diversi e riconoscibili a prima vista - elefante, ippopotamo, serpente, orso, giraffa, rinoceronte - che si incastrino perfettamente uno con l’altro. Un progetto, come mi capita sempre, molto lungo e complicato.

2.2 Planning an economic experiment

- Planning an experiment requires a careful definition its building blocks
- **Choice Task (stage game)**
  - What are participants required to do?
    - Which data types will I collect?
- **Repetitions**
  - Do participants repeat the task?
    - One-shot, Stationary repetition, conditional task, ...
- **Matching**
  - Are participants interacting?
    - Which kind of matching protocol?
- **Payment scheme**
  - Is the payment scheme adopted incentive-compatible?
    - Random lottery incentive, cumulative payment, strategy method ...
- All details should be clearly defined before the writing of the software
  - Of course, adjustments are possible but prone to mistakes

2.3 An example: Prisoner’s Dilemma

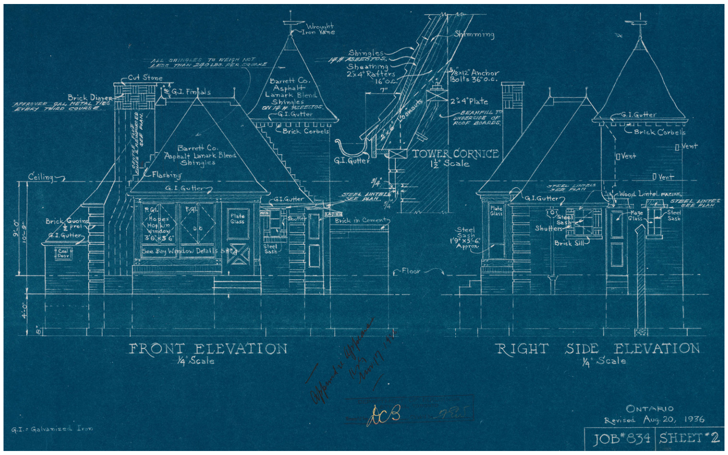
- **Choice Task (stage game)**
- 

		Player 2	
		A	**B**
Player 1	**A**	R, R	S, T
	B	T, S	P, P

- Participants make a dychotomous choice
  - A ($\neq A$)
- **Repetitions**
  - It may be played an indefinitely number of rounds → TIT-FOR-TAT

    - A finitely number of round → (sequential equilibrium à la Kreps et al. (1982))
    - One-shot
        - What theory are you testing?
- **Matching**
    - Partner or Stranger (Andreoni and Miller 1993)
- **Payment Scheme**
    - If repeated, all round cumulatively or one round random (RLI)
        - What are the assumptions of different payment methods?

2.4 Experimental instructions



- Experimental instructions are the blueprint for the experimental software
    - They contain details about
        - Payoff functions
        - Matching
        - Rounds
        - Format of the choice task
            - Cardinal, ordinal, categorical, …
    - Important to match the wording of instructions and of the software

2.5 Computerized experiments

2.6 Advantages of computerized experiments

- Computerized experiments present several advantages over paper&pencil experiments
    - Live interaction
    - Dynamic interfaces

    - Codified data
    - Wide audience
        - Web-based experiments
- Specialized softwares are available
    - z-Tree (Fischbacher 2007) is the de-facto standard for lab experiments
        - Runs locally on Win OS
        - See my online lecture notes if interested
    - oTree (Chen, Schonger, and Wickens 2016) is an emerging software
        - Platform-independent, web-based
- Here, we focus on oTree

2.7 About oTree

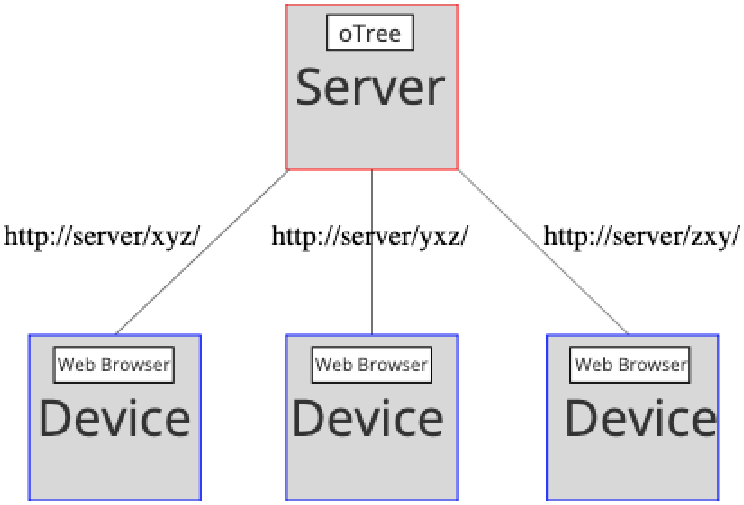
- oTree is a framework based on Python to run controlled experiments
    - Games
        - e.g., Public Goods Games (PGG)
    - Individual decision making
        - e.g., Risk elicitation tasks
    - Surveys and tests
        - e.g., Raven test
- Support by the community
    - Forum
    - Code developed by others
        - e.g. Holzmeister (2017)
        - A list of apps can be found here
- oTree is open-source,
    - Licensed under an adaptation of the MIT license.
        - Cite it in the paper when you use it

2.8 Code

- Programming language of oTree is **Python**
    - Popular object-oriented programming language
    - Developed in early 90's by Guido Van Rossum
- OTree's user interface is based on **HTML5**
    - Supported by modern browser and rich in functionalities
- OTree is based on **Django** web application framework
    - OTree applications are web applications
- All the components of oTree are free and open-source

2.9 Functioning

- The basic setup consists in
    1. An **app (experiment)** written within oTree
    2. A **server** computer
        - Cloud server, local PC …
    3. Subjects’ **devices** with a browser
        - PC, Laptop, Tablet, Mobile Phone …
- oTree creates a session on the server and generates links for all participants
- Participants click on the links and are sent to a personal page
    - They submit their answers, which are collected by the server
    - The experimenter can check the progress on the server



3 oTree app

3.1 Conceptual overview

<table>
<tr><th colspan="6">Session</th></tr>
<tr><td colspan="4">Subsession</td><td colspan="2">Subsession</td></tr>
<tr><td>Page</td><td>Page</td><td>Page</td><td>Page</td><td>Page</td><td>Page</td></tr>
</table>

- **Sessions**
    - Participants take part in a series of tasks
- **Subsessions**
    - Sections of a session
        - EXAMPLE: a PGG is subsession 1 and a questionnaire is subsession 2

    - Repetitions of the same task are performed over distinct subsessions (periods)
- **Page**
  - Sections of a subsession
    - EXAMPLE: the PGG is made of 4 pages (instructions, ...) and the questionnaire of 2 pages
- **Groups**
  - Each subsession can be divided into groups of players
    - Groups can be shuffled between subsessions

3.2 Object hierarchy

- oTree's entities are organized with the following hyerarchy
- Session
  - Subsession
    - Group
      - Player
        - Page
- A session is a series of subsessions
  - A subsession contains multiple groups
    - A group contains multiple players
      - Each player proceeds through multiple pages
- ⚠ player and participant have different meanings
  - A Player is a section of a Participant
    - Similar to Subsession and Session
      - e.g., a participnt can be player X in a subsession and player Y in another subsession

3.3 What is self? 😕

- In our journey we will often encounter `self`
  - It is the *class* a method *belongs* to!
- As an example, in *models.py* we will find

```
class Player(BasePlayer):
    def set_payoff(self):
        self.payoff=-99
```

- The `self` here is Player
  - The method `set_payoff` defines the payoff for the player

```
class Group(BaseGroup):
    payoff = models.CurrencyField()
    def set_payoff(self):
        self.payoff = -99
```

- The `self` here is Group
  - The method `set_payoff` defines the payoff for the group

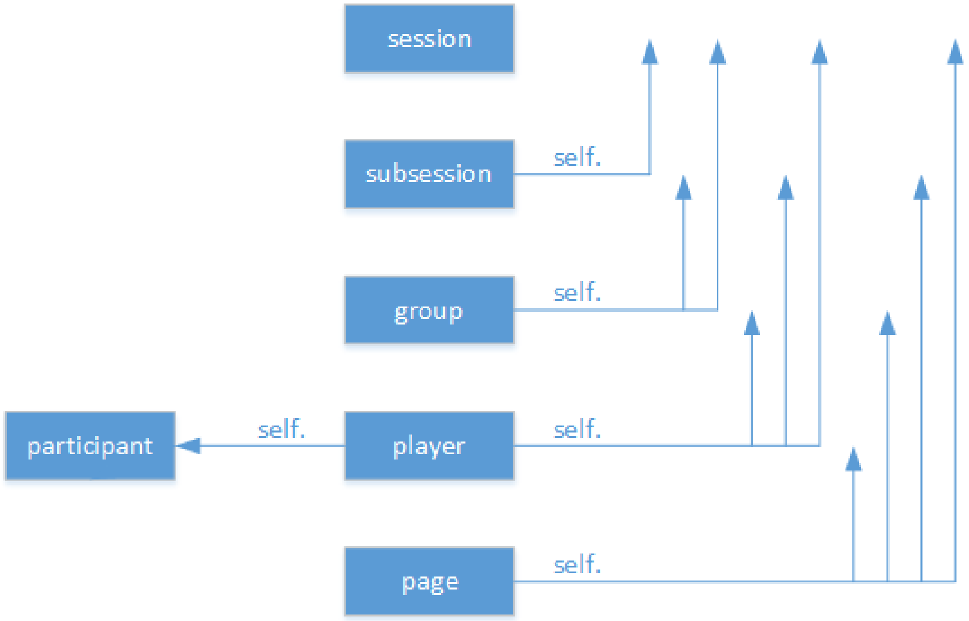
3.4 Using self to access information in other objects

- The diagram shows the properties you can access with `self`
- If you are in a method in class *Player*, you have access to the property `payoff` with `self.payoff`

```
class Player(BasePlayer):
  def set_payoff(self):
      self.payoff=-99
```

- If you are in a method in class *Page*, you need to “navigate” to the to the property `payoff` with `self.player.payoff`

```
class PayoffPage(BasePlayer):
  def vars_for_template(self):
      return{'payoff':self.player.payoff}
```



3.5 Illustrative example of self in *models.py*

- models.py is the file that controls the database of the app
    - Subsession
        - Group
            - Player

```
class Subsession(BaseSubsession):
    def example(self):

        # current subsession object
        self

        # parent objects
        self.session

        # child objects
        self.get_groups()
        self.get_players()

        # accessing previous Subsession objects
        self.in_previous_rounds()
        self.in_all_rounds()

class Group(BaseGroup):
    def example(self):

        # current group object
        self

        # parent objects
        self.session
        self.subsession

        # child objects
        self.get_players()

class Player(BasePlayer):

    def example(self):

        # current player object
        self

        # method you defined on the current object
        self.my_custom_method()
        # parent objects
        self.session
        self.subsession
        self.group
        self.participant

        self.session.config

        # accessing previous player objects
        self.in_previous_rounds()

        # equivalent to self.in_previous_rounds() + [self]
        self.in_all_rounds()
```

4 Your first app

4.1 Create an app

- To create an app named “*my_first_app*” move to the oTree folder

```
cd oTree
```

- and create the app

```
otree startapp my_first_app
```

- Move to the folder *my_first_app*
  - You will find the following files
    - **models.py**
    - **pages.py**
    - tests.py
  - And a subfolder
    - **templates/my_first_app**
      - **MyPage.html**, **Results.html**

4.2 models.py

- Here you define the structure of your data
  - 3 data models

        - Subsession
        - Group
        - Player
            - These are Python classes (see below)
    - A model is basically a **database**
        - Specify columns and their nature
            - Integers, strings, ...

```python
from otree.api import (
    models,
    widgets,
    BaseConstants,
    BaseSubsession,
    BaseGroup,
    BasePlayer,
    Currency as c,
    currency_range,
)

author = "Me"

doc = """
This is my first app
"""


class Constants(BaseConstants):
    name_in_url = 'my_first_app'
    players_per_group = None
    num_rounds = 1

class Subsession(BaseSubsession):
    pass

class Group(BaseGroup):
    pass

class Player(BasePlayer):
    pass
```

4.3 templates

- These are the pages that are displayed to participants
    - **html files** that contain informations and *forms*

        - *forms* are used to collect data
    - A default MyPage.html is created
        - {% formfields %} will display the forms of the page
            - see *pages.py*
        - {% next_button %} will display a button to continue
- The HTML can contain “fancy” stuff
    - Javascript
    - Bootstrap framework
    - ...

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

{% block title %}
    Page title
{% endblock %}

{% block content %}

    {% formfields %}

    {% next_button %}

{% endblock %}
```

4.4 pages.py

- Pages that the participants see are defined in *pages.py*
    - Logic for how to display the HTML templates
        - when, how, and what to display
- *page_sequence* gives the order of pages
    - if there are multiple rounds the sequence is repeated

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


class MyPage(Page):
    pass


class ResultsWaitPage(WaitPage):
    def after_all_players_arrive(self):
        pass


class Results(Page):
    pass


page_sequence = [MyPage, ResultsWaitPage, Results]
```

5 Implement a simple survey

5.1 Steps to implement the survey

- We implement a simple survey to collect the age of course participants

1. Develop the oTree code
    - models.py
    - pages.py
    - templates
2. Test our code locally
3. Transfer our code to a server (running oTree)
4. Send links to participants (you)
5. Fill in the survey
6. Collect outcomes and analyze them

5.2 Develop the oTree code: models.py

```
from otree.api import (
    models,
    widgets,
    BaseConstants,
    BaseSubsession,
    BaseGroup,
    BasePlayer,
    Currency as c,
    currency_range,
)


author = 'Your name here'

```

```
doc = """
Collect participants' age
"""


class Constants(BaseConstants):
    name_in_url = 'my_first_survey'
    players_per_group = None
    num_rounds = 1


class Subsession(BaseSubsession):
    pass


class Group(BaseGroup):
    pass


class Player(BasePlayer):
    age = models.IntegerField(choices=range(18, 99, 1))
```

- Create the field `age` in class player
  - The input will be an integer spanning 18-99

5.3 Develop the oTree code: pages.py

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


class CollectAge(Page):
    form_model = 'player'
    form_fields = ['age']

class Results(Page):
    def vars_for_template(self):
        return{
            'your_age':self.player.age
        }


page_sequence = [CollectAge, Results]
```

- In page `CollectAge` we insert a form for class *player*
    - The field is `age`
        - Must be the same name we used in models.py
            - `age` and not, as an example, `Age`
- In page `Results` we give a feedback
    - `your_age` will be printed on the screen
- `page_sequence` defines the sequence of your pages
    - All your pages are a class that must be defined here

5.4 Develop the oTree code: templates

- Templates are in the subfolder `templates/my_first_survey`

- **CollectAge**

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

 {% block title %}
   Insert your age here
 {% endblock %}

{% block content %}

{% formfield player.age %}

{% next_button %}

{{ form.age.errors }}

{% endblock %}
```

- `{% formfield player.age %}` is where the information is input
    - Variable `age` in the class player
- Aesthetics elements
    - Title
    - Body
    - Next button

5.5 Develop the oTree code: templates (ii)

- Templates are in the subfolder `templates/my_first_survey`

- **Results**

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

{% block title %}
    Your age
{% endblock %}

{% block content %}

Your age is {{ your_age }}. Thank you for answering!

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

- `{{ your_age }}` is passed by `vars_for_template` in pages.py
  - The right name must be put within `{{ }}` brackets

5.6 Test our code locally

- Move to the folder in which your oTree is installed
  - Usually `cd ~/oTree`
  - Add your new app to the `settings.py file

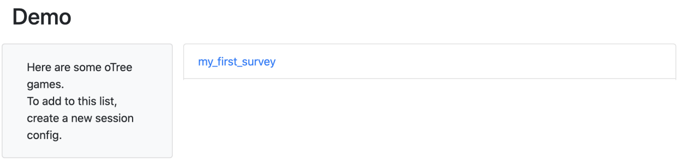
```
SESSION_CONFIGS = [
    dict(
        name='my_first_survey',
        display_name='my_first_survey',
        num_demo_participants=4,
        app_sequence=['my_first_survey'],
    )
]
```

- In `app_sequence` the exact name of the apps should be given
- Give command

```
otree devserver
```

5.7 Test our code locally (ii)

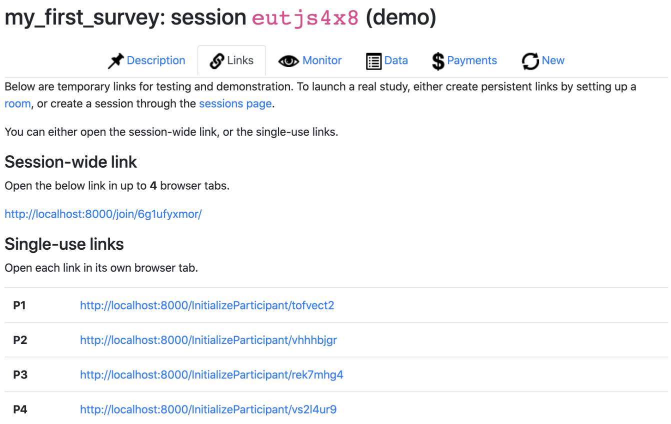
- Open a browser and insert `http://localhost:8000/`



Demo

Here are some oTree games.
To add to this list, create a new session config.

- Click on the name of your app



my_first_survey: session eutjs4x8 (demo)

Below are temporary links for testing and demonstration. To launch a real study, either create persistent links by setting up a room, or create a session through the sessions page.

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

Session-wide link

Open the below link in up to **4** browser tabs.

http://localhost:8000/join/6g1ufyxmor/

Single-use links

Open each link in its own browser tab.

P1	http://localhost:8000/InitializeParticipant/tofvect2
P2	http://localhost:8000/InitializeParticipant/vhhhbjgr
P3	http://localhost:8000/InitializeParticipant/rek7mhg4
P4	http://localhost:8000/InitializeParticipant/vs2l4ur9

- Click on the session-wide link and try it

5.8 Now push our app online …

via GIFER

6 Python

6.1 Basics

- A very short introduction to Python
  - Based on Learn X in Y minutes, where X=Python Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)
  - and on Ascher and Lutz (1999)

```
#####################################################
# Numbers and logical operators
####################################################

# Math is what you would expect
1 + 1   # => 2
8 - 1   # => 7
10 * 2  # => 20
35 / 5  # => 7.0

# Enforce precedence with parentheses
1 + 3 * 2  # => 7
(1 + 3) * 2  # => 8


# Boolean Operators
## Note "and" and "or" are case-sensitive
True and False  # => False
False or True   # => True

## True and False are actually 1 and 0 but with different keywords
True + True # => 2
True * 8    # => 8
False - 5   # => -5

# Comparisons

## Equality is ==
1 == 1  # => True
2 == 1  # => False

## Inequality is !=
1 != 1  # => False
2 != 1  # => True

## More comparisons
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True
```

6.2 Basics: Strings and variables

- Strings are an ordered collection of *characters*

```


# Strings are created with " or '

"This is a string."

'This is also a string.'

# Strings can be added too! But try not to do this.
"Hello " + "world!"  # => "Hello world!"

# You can also format using f-strings or formatted string literals (in
name = "Reiko"
f"{name} is {len(name)} characters long." # => "Reiko is 5 characters l

# Strings can be sliced and indexed

name[0] => "R"
name[-2]  => "k"


# There are no declarations, only assignments.
# Convention is to use lower_case_with_underscores
some_var = 5
some_var  # => 5
```

6.3 Basics: Lists and tuples

- Lists are ordered collections of arbitrary items
    - Can contain *numbers, strings, and other lists*
    - Accessed by offset
        - As an example, all players in a group

```
# empty list
li = []

# Prefilled list
other_li = [4, 5, 6]

# Add stuff to the end of a list with append
other_li.append(7)    # => li is now [4, 5, 6, 7]

# Remove from the end with pop
other_li.pop()        # => 7 li is now [4, 5, 6]

# Access a list like you would any array
other_li [0]   # => 4

# Look at the last element
other_li [-1]  # => 6

# You can look at ranges with slice syntax.
# The start index is included, the end index is not
# (It's a closed/open range for you mathy types.)
other_li [1:3]   # Return list from index 1 to 3 => [5, 6]

# Remove arbitrary elements from a list with "del"
del other_li[2]  # li is now [4, 5]

# Insert an element at a specific index
other_li.insert(1, 2)  # li is now [4, 2, 5] again

# Get the index of the first item found matching the argument
other_li.index(2)  # => 1

# Check for existence in a list with "in"
1 in other_li  # => False

# Examine the length with "len()"
len(other_li)  # => 3

# Tuples are like lists but are immutable.
tup = (1, 2, 3)
tup[0]      # => 1
```

6.4 Basics: Dictionaries

- Useful way to collect and organize information
  - As an example, payoffs of players

```
# Dictionaries store mappings from keys to values

empty_dict = {}

# Here is a prefilled dictionary
filled_dict = {"one": 1, "two": 2, "three": 3}

# Look up values with []
filled_dict["one"]  # => 1

# Get all keys as an iterable with "keys()". We need to wrap the call i
# to turn it into a list. Order of insertion is preserved (in Python >=

list(filled_dict.keys())  # => ["one", "two", "three"]


# Get all values as an iterable with "values()". Once again we need to
# in list() to get it out of the iterable.

list(filled_dict.values())  # => [1, 2, 3]

# Check for existence of keys in a dictionary with "in"
"one" in filled_dict  # => True
1 in filled_dict      # => False

# Adding to a dictionary
filled_dict.update({"four":4})  # => {"one": 1, "two": 2, "three": 3, "
filled_dict["four"] = 4         # another way to add to dict

# Remove keys from a dictionary with del
del filled_dict["one"]  # Removes the key "one" from filled dict
```

6.5 Basics: Control Flow and Iteration

- Many times you need to iterate through variables
  - As an example, collect all choice sof participants in a group

```
# Let's just make a variable
some_var = 5

# Here is an if statement. Indentation is significant in Python!
# Convention is to use four spaces, not tabs.
# This prints "some_var is smaller than 10"

# if conditions

if some_var > 10:
    print("some_var is totally bigger than 10.")
elif some_var < 10:    # This elif clause is optional.
    print("some_var is smaller than 10.")
else:                  # This is optional too.
    print("some_var is indeed 10.")

# loops

for i in range(4):
    print(i)
# =>0
# =>1
# =>2
# =>3

for animal in ["dog", "cat", "mouse"]:
    # You can use format() to interpolate formatted strings
    print("{} is a mammal".format(animal))
# =>dog is a mammal
# =>cat is a mammal
# =>mouse is a mammal

# We can use list comprehensions to loop or filter
numbers = [3, 4, 5, 6, 7]
[x for x in numbers if x > 5]   # => [6, 7]
```

- Indenting is important!

6.6 Basics: Functions

- Functions are a device that groups a bunch of statements

```

# Use "def" to create new functions
def add(x, y):
    print("x is {} and y is {}".format(x, y))
    return x + y  # Return values with a return statement

# Calling functions with parameters
add(5, 6)  # => prints out "x is 5 and y is 6" and returns 11
```

6.7 Basics: Classes

- Classes are main **object-oriented-programming (OOP)** in Python
    - **Class objects** provide default behavior
        - The *class* statement creates a class object and assigns it a name
        - Assignments inside class statements make class attributes
        - Class attributes export object state and behavior
            - *def* statements inside class generate a *method*
    - **Instance objects** are generated from classes
        - Calling a class object makes a new *instance object*
        - Each instance object inherits class attributes and gets its own namespace
        - Assignments to *self* in methods make per-instance attributes
            - *self* refers to the instance being processed

6.8 Classes: An Example

- Pokemon is a **class** with some properties
    - Height
    - Weight
    - Category
- Two **instances** of the class

Charmander:

Height: 2’ 00";

Weight 18.7 lbs;

Category: Lizard

Bulbasaur:

Height: 2’ 04";

Weight: 15.2 lbs;

Category: Seed

6.9 Classes: An Example (ii)

```
class Pokemon:
    nature = "Pokemon"# this property is shared by all instances
    def __init__(self, name, height, weight, category):
            # Assign the argument to the instance's name attribute
            self.name = name
            self.height = height
            self.weight = weight
            self.category = category
            self.comment=[]
    # to add a comment
    def add_comment(self, comment):
        self.comment.append(comment)
    # convert height and weight into metric
    def convert_metric(self):
        # conversion rates
        feet_conv_cm=30.5
        inch_conv_cm=2.54
        lbs_conv_kg=0.453592
        self.height_cm = int(self.height.split("'")[0])*feet_conv_cm+in
        self.weight_kg = self.weight*lbs_conv_kg
        # print the result
        print("Height (cm): %f, Weight (Kg): %f" %(self.height_cm, self
```

6.10 Classes: An Example (iii)

```
# Create two instances of the class
p_1 = Pokemon("Charmander","2' 5''", 18.7, "Lizard")
p_2 = Pokemon("Bulbasaur","2' 04''", 15.2, "Seed")

# Add a comment
p_1.add_comment("Its ability is Blaze")
p_2.add_comment("Its ability is Overgrow")

# Convert their measures in metric system
p_1.convert_metric()
p_2.convert_metric()

# Who is taller?
if p_1.height>p_2.height:
    print(p_1.name + " is taller")
elif p_1.height<p_2.height:
    print(p_2.name + " is taller")
else:
    print(p_2.name + " and" + p_1.name + "are equally taller")
```

6.11 Thank you

7 Assignment

7.1 Software required during the course

- Install the following software on a machine you have access to
- All the software is open source and freely downloadable
    - You find below the links to OS WIN installations, installations for other OS are available online
- **oTree**
    - oTree
- **Python**
    - Python 3
- Editor (suggested)

    - VS Code

- Without this software you will not be able to follow next lectures

7.2 Try this!

1. Consider the *class Pokemon* we illustrated above
    - Create a new instance of the class
    - **Charizard**: Height: 5’ 07"; Weight: 199.5 lbs; Category: Flame
    - Check whether Charizard is taller than Bulbasaur
2. Extend our survey to collect gender
    - Hint: the variable in class Player could be implemented like this

```
sex =  models.CharField(widget=widgets.RadioSelectHorizontal(),choices=
```

References

Andreoni, James, and John H Miller. 1993. “Rational Cooperation in the Finitely Repeated Prisoner’s Dilemma: Experimental Evidence.” *The Economic Journal* 103 (418): 570–85.

Ascher, David, and Mark Lutz. 1999. *Learning Python*. O’Reilly.

Chen, Daniel L, Martin Schonger, and Chris Wickens. 2016. “OTree—an Open-Source Platform for Laboratory, Online, and Field Experiments.” *Journal of Behavioral and Experimental Finance* 9: 88–97.

Fischbacher, Urs. 2007. “Z-Tree: Zurich Toolbox for Ready-Made Economic Experiments.” *Experimental Economics* 10 (2): 171–78.

Holzmeister, Felix. 2017. “OTree: Ready-Made Apps for Risk Preference Elicitation Methods.” *Journal of Behavioral and Experimental Finance* 16: 33–38.

Kreps, David M, Paul Milgrom, John Roberts, and Robert Wilson. 1982. “Rational Cooperation in the Finitely Repeated Prisoners’ Dilemma.” *Journal of Economic Theory* 27 (2): 245–52.