Frankenstein's Python

PYTHON & SCIENCE

Moritz Nipshagen mnipshagen@uos.de
Antonia Hain ahain@uos.de

Structure

- Week 1: Introduction
- Week 2: Syntax, Variables & Functions
- Week 3: Control Structures
- Week 4: Lists & Collections
- Week 5: RegEx & Strings
- Week 6: Sorting & I/O
- Week 7:Debugging, Errors & Strategies
- Week 8: 4P: Packages, Practices and Patterns
- Week 9: Object Oriented Programming
- Week 10: Time, Space & Documentation
- Week 11: Numpy & Matplotlib
- **Week 12: Python & Science**
- Week 13: Neural Nets
- Week 14: Honorable Mentions & Wrap Up

Python & Neuroscience

- Many experiments you will conduct in neuroscience these days are run with the help of a script
- The script will most often be responsible for both leading the subject through the experiment and collecting data
- Working with Python to create our experiment scripts is a good idea as there are many libraries that can help you build an experiment and handle your data
- In this lecture, we are going to introduce you to some of them

Experiment Frameworks: PsychoPy & Expyriment

Let's Build an Experiment

- Let's compile a list of things we (usually) need to build our own psychology experiments:

1. Some sort of **graphic surface**
   - Like a window
2. **Stimuli**
   - Like fixation points, shapes, user instructions
   - We also need to know how to display them on the graphic surface
3. **User input**
   - Like key presses
4. The actual **experiment procedure**
5. **Data logging**

- Plus, potentially communicating with other devices like eye trackers, EEG, ...
- That's a lot of things to consider….

PsychoPy and Expyriment

- Both PsychoPy and Expyriment are libraries that offer tools for the creation of psychological experiments
- So they give us the possibility to implement the little building blocks we need for our experiments
- In the following, we will mostly consider the main things mentioned on the previously slide
  - But there are more functionalities one might need in an experiment that are supported as well

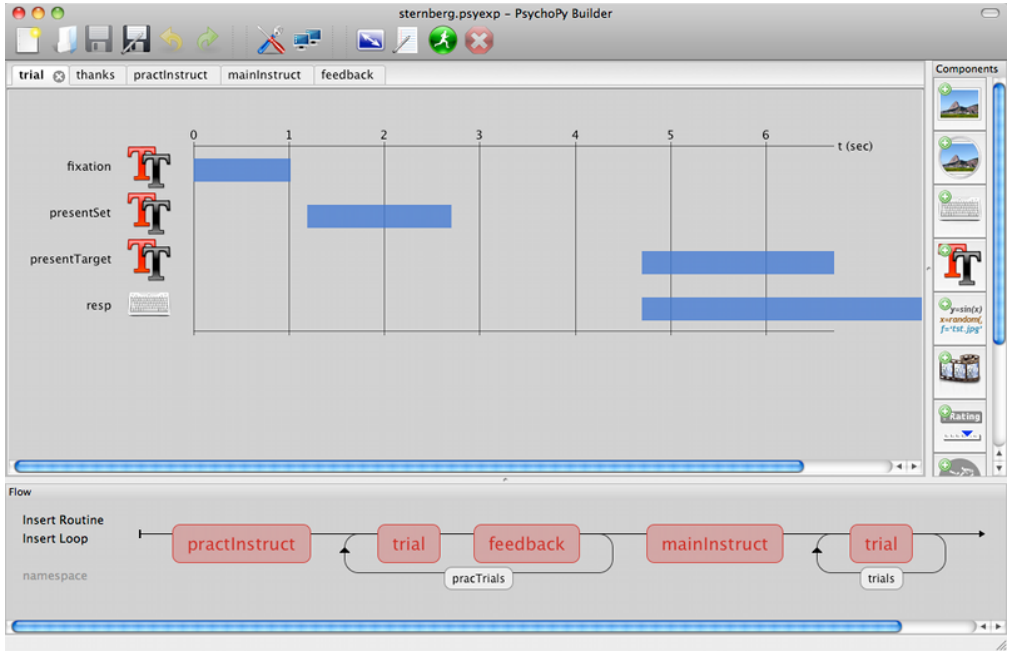
PsychoPy

PsychoPy

- PsychoPy is an open-source library specifically designed for psychology and neuroscience experiments
  - http://www.psychopy.org/about/overview.html
- It is the most commonly used experimental psychology library as of now
  - (probably)

PsychoPy - Builder and Coder

- PsychoPy has two interfaces: the **Builder** and the **Coder**
- The *Builder* offers a **GUI** in which users can plug together their experiments
- The *Coder* offers **packages** we can incorporate into our Python scripts to code the experiments
- Here, we're going to focus on the *Coder*:
  - It offers more control over the experiments
  - We're all pro-coders now that love writing our own code ;)



Builder image taken from official website

PsychoPy - The Coder

Reference Manual (API)

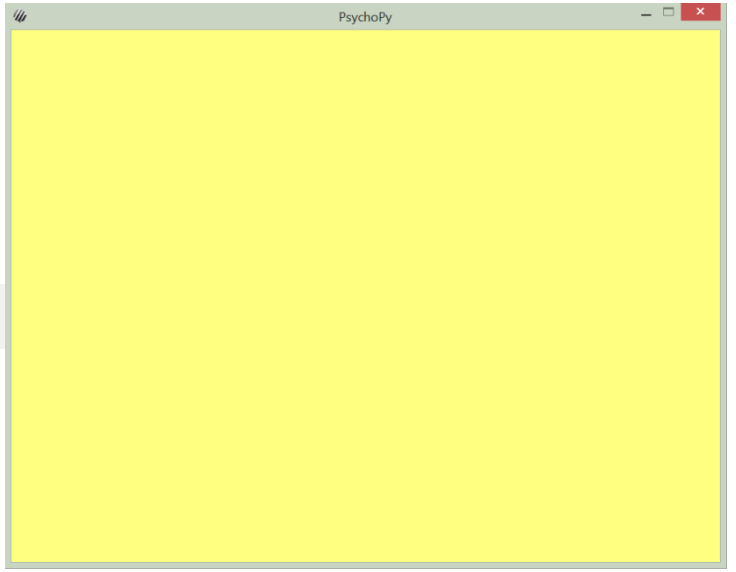
Contents:

- psychopy.core – basic functions (clocks etc.)
- psychopy.visual – many visual stimuli
- psychopy.clock – Clocks and timers
- psychopy.data – functions for storing/saving/analysing data
- Encryption
- psychopy.event – for keypresses and mouse clicks
- psychopy.filters – helper functions for creating filters
- psychopy.gui – create dialogue boxes
- psychopy.hardware – hardware interfaces
- psychopy.info – functions for getting information about the system
- psychopy.iohub – ioHub event monitoring framework
- psychopy.logging – control what gets logged
- psychopy.microphone – Capture and analyze sound
- psychopy.misc – miscellaneous routines for converting units etc
- psychopy.monitors – for those that don't like Monitor Center
- psychopy.parallel – functions for interacting with the parallel port
- psychopy.preferences – getting and setting preferences
- psychopy.serial – functions for interacting with the serial port
- psychopy.sound – play various forms of sound
- psychopy.tools – miscellaneous tools
- psychopy.voicekey – Real-time sound processing
- psychopy.web – Web methods

- This is an overview of the packages included in the PsychoPy coder as listed in the documentation
- The ones we need today are `core`, `visual`, `data` and `event`
- You can find the manual here
- We will now walk through our list of experiment building blocks step by step

1. PsychoPy – Graphic Surface

- In PsychoPy, our graphic surface will be an object of class `Window`
  - that shouldn't be too hard to remember
- `Window` objects are part of the package `visual`
  - So let's import that first




```
from psychopy import visual
```

- Now we can create a window:

```
# create window of size 800x600px on a monitor object we'll call testMonitor and with yellow color
# the scaling unit for object sizes and locations is normalized between -1 and 1
mywin = visual.Window([800,600], monitor="testMonitor", units="norm", color=[255,255,0])
```

- Looks pretty boring, eh?

2. PsychoPy - Stimuli

- `visual` also contains classes for several visual stimuli
  - Like text stimuli, shapes (e.g. circles and rects), patterns (like grating stimuli), …
- We'll start out by adding a fixation point in the middle of the screen

```
from psychopy import visual
mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

# create grating simulus on our window, size 0.015, center position, spatial frequency 0
# (grating stimulus is otherwise striped), pitch black (-1) color
fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)
```

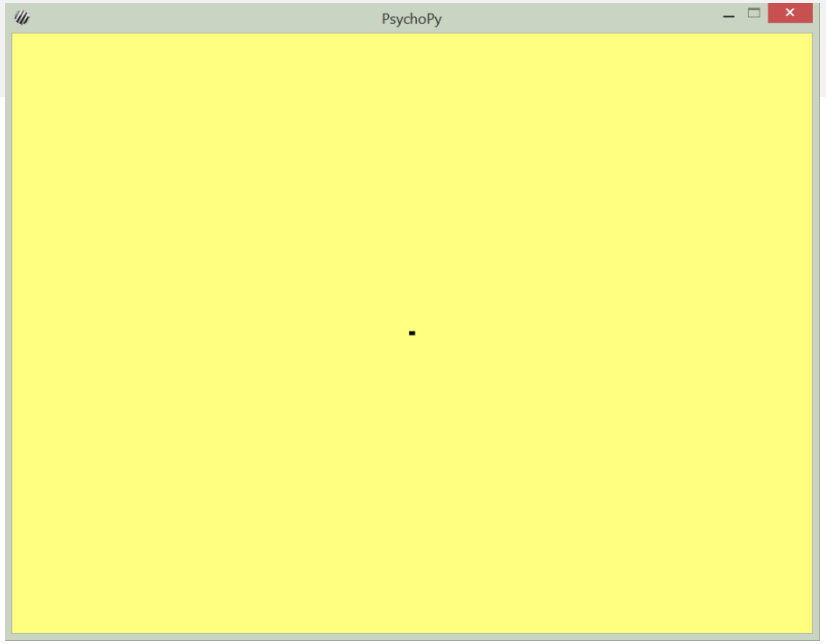
2. PsychoPy - Stimuli

- However, for our stimulus to show, we need to do two more things:
- First is `fixation.draw()`
  - for the stimulus to actually be drawn
- Then we need to do `mywin.flip()`
  - For our drawing to show
  - The reason for this is that the fixation is drawn on the *back buffer* where all objects are drawn before they are shown
  - Meanwhile, the *front buffer* is in place and holds everything we currently see
  - *Flipping* will swap the front and back buffers such that after drawing is finished, the drawing can be shown on the new front buffer while the next drawing can be prepared on the new back buffer

2. PsychoPy - Stimuli

- So our code now looks like this:

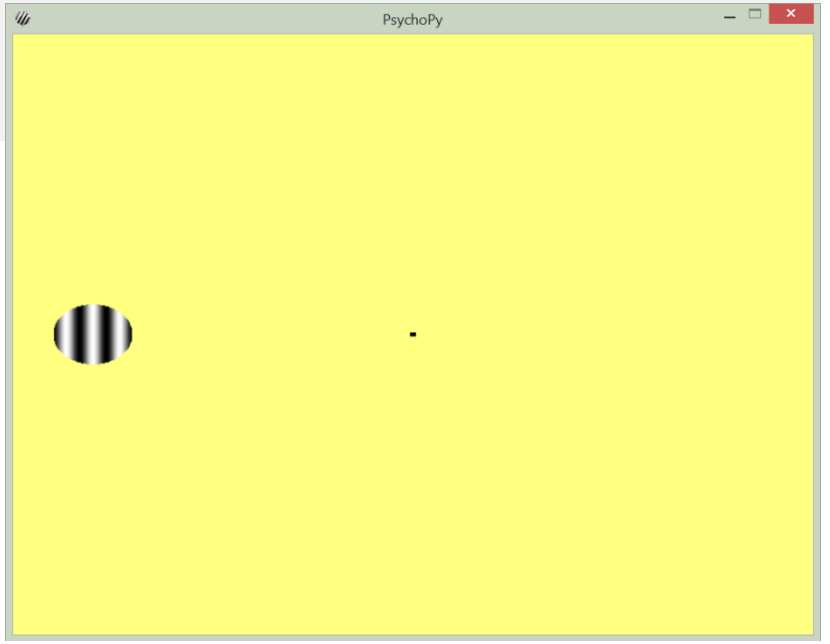
```
from psychopy import visual
mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)

fixation.draw()
mywin.flip()
```



2. PsychoPy - Stimuli

- Next, we'll add another (actual) grating stimulus to the family

```
from psychopy import visual
mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)
grating = visual.GratingStim(win=mywin, mask="circle", size=0.2, pos=[-0.8,0], sf=3)

fixation.draw()
grating.draw()
mywin.flip()
```



- Beautiful.

3. PsychoPy - User Input: waitKeys()

- With our current code, we don't have much time to look at the window though – it closes right away
- To prevent this from happening, we can wait for a key press from the user with `event.waitKeys()`
  - This function has some extra parameters as well, for example, we can specify which keys to look for – this is arguably the most common usage though

import `event` to be able to do this

```
from psychopy import visual, event
mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)
grating = visual.GratingStim(win=mywin, mask="circle", size=0.2, pos=[-0.8,0], sf=3)

fixation.draw()
grating.draw()
mywin.flip()
event.waitKeys()
```

and add `waitKeys()` here

3. PsychoPy - User Input: Global Keys

- However, it would be pretty inconvenient if keys presses could only be processed whenever the program is currently waiting for some key to be pressed
- To avoid this, we can define global keys that execute a function whenever they are pressed
- Lets make a key event for the escape button: Quit the program when it is pressed
  - The `core.quit()` function can be used for this, so we'll also need to import the module core

```
# add 'escape' to global keys. core.quit() function is executed when escape button is pressed
event.globalKeys.add(key='escape', func=core.quit)
```

- **For this to work, you have to deactivate Num Lock!**
  - Don't ask us why. We don't know. Seriously.

3. PsychoPy - User Input: Putting it Together

```
from psychopy import visual, event, core

mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)
grating = visual.GratingStim(win=mywin, mask="circle", size=0.2, pos=[-0.8,0], sf=3)

event.globalKeys.add(key='escape', func=core.quit)

fixation.draw()
grating.draw()
mywin.flip()
event.waitKeys()
mywin.flip()
event.waitKeys()
```

- So now we have a nice little program with a fixation and a stimulus that closes itself on the escape key and for any other key press wipes the screen
  - And then closes itself after pressing another key
- Now we want to include some sort of experiment condition

4. PsychoPy – Experiment Procedures

- Usually, experiments consist of various *trials* with different conditions
- For this, we can use the `TrialHandler` and the `StairHandler`
  - Both can handle the selection of the next trial and data storage
  - The `TrialHandler` can choose a trial (sequentially or randomly) out of a list of pre-defined conditions
  - The `StairHandler` uses an *adaptive staircase*, which means that each trial, the next condition is generated based on the participant's response
- In the following, we will implement a simple `TrialHandler`

4. PsychoPy – Experiment Procedures

- The `TrialHandler` is part of the module `data`
  - So remember to first import this module as well

```
psychopy.data.TrialHandler(trialList, nReps, method='random', dataTypes=None, extraInfo=None,
seed=None, originPath=None, name='', autoLog=True)
```

- Now, what we want to do is draw the grating stimuli sequentially at different horizontal positions
- Since we are using the `norm` unit for our window, horizontal positions within the screen are between -1 and 1
- So let's choose 5 horizontal positions:
  - These are now our trials during the experiment

```
positions = [-1, -0.5, 0, 0.5, 1]
```

4. PsychoPy – Experiment Procedures

- Now we can make a `TrialHandler` and feed the trials into it:

```
positions = [-1, -0.5, 0, 0.5, 1]

# 1 repetition of all trials, in sequential order
handler = data.TrialHandler(positions, 1, method='sequential')
```

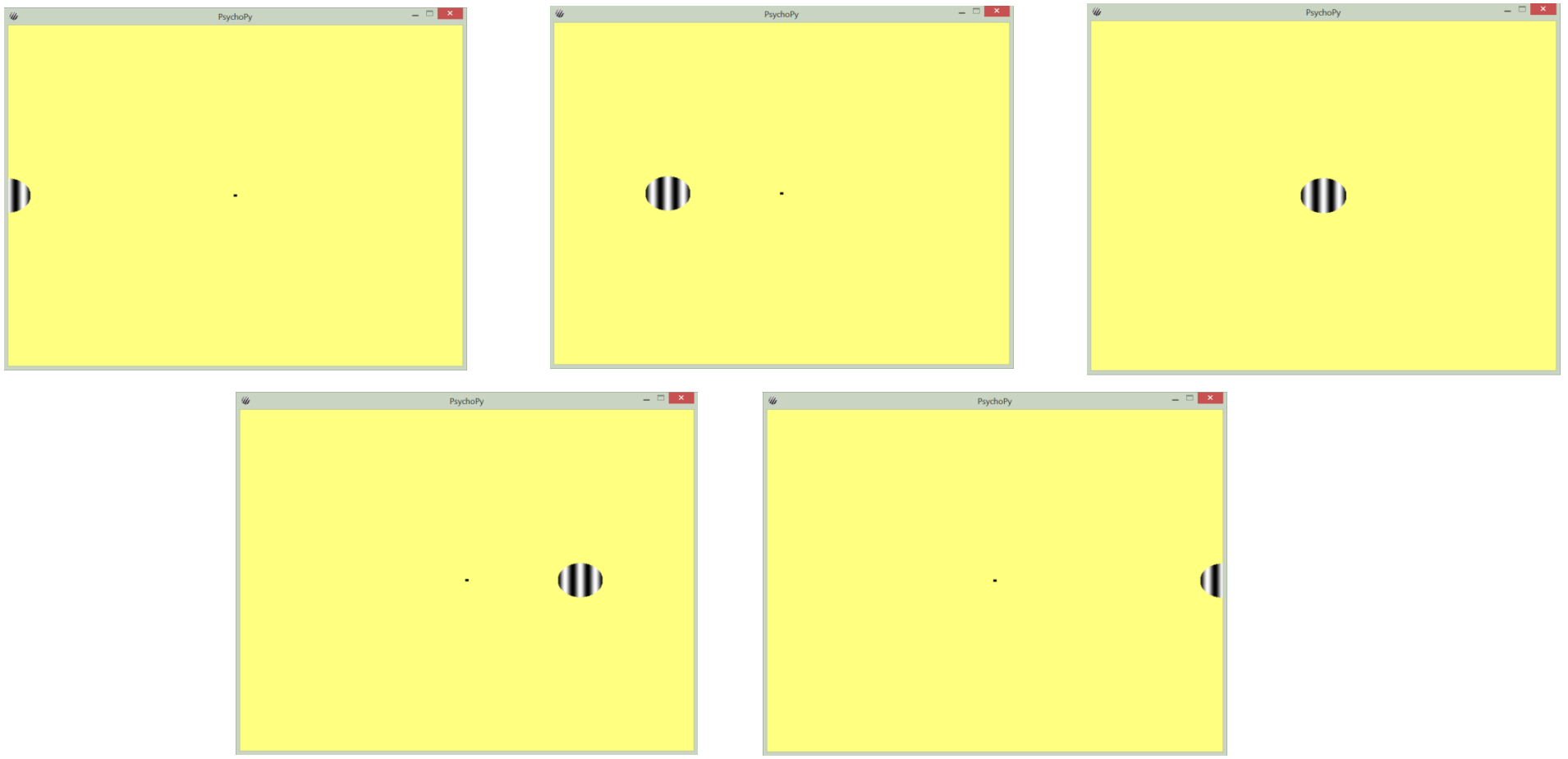
- What's left to do is to make a loop that executes all the conditions

```
# go through all trials as given by the TrialHandler
for trial in handler:
    # set grating stimulus to new position
    grating.setPos([trial,0])

    # draw all stimuli and wait for input
    fixation.draw()
    grating.draw()
    mywin.flip()
    event.waitKeys()
```

4. PsychoPy – Complete Code

```
from psychopy import visual, event, core, data

mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

positions = [-1, -0.5, 0, 0.5, 1]
handler = data.TrialHandler(positions, 1, method='sequential')

fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)
grating = visual.GratingStim(win=mywin, mask="circle", size=0.2, pos=[0,0], sf=3)

event.globalKeys.add(key='escape', func=core.quit)

for trial in handler:
     grating.setPos([trial,0])

     fixation.draw()
     grating.draw()
     mywin.flip()
     event.waitKeys()
```

4. PsychoPy - The Program



5. PsychoPy - Data Logging

- There are multiple ways to do data logging which would be out of scope for today's lecture
- Basically, you can just do this the way we're used to (Python built-in file handling)
- Or you can take a look at the documentation for details
  - Especially here and here

PsychoPy – Code Structure

- Following these certain guidelines when structuring your code is recommended:
  - This is not the order in which you necessarily have to *write* your code – just how it is best structured in the end

```
from psychopy import visual, event, core, data

mywin = visual.Window(size=[800,600], monitor="testMonitor", units="norm", color=[255,255,0])

positions = [-1, -0.5, 0, 0.5, 1]
handler = data.TrialHandler(positions, 1, method='sequential')

fixation = visual.GratingStim(win=mywin, size=0.015, pos=[0,0], sf=0, color=-1)
grating = visual.GratingStim(win=mywin, mask="circle", size=0.2, pos=[0,0], sf=3)

event.globalKeys.add(key='escape', func=core.quit)

for trial in handler:
    grating.setPos([trial,0])

    fixation.draw()
    grating.draw()
    mywin.flip()
    event.waitKeys()
```

first create the window

then the handler

then prepare all your stimuli before you use them

then your global key events

then your actual procedure

PsychoPy – The Catch

- We have a confession to make …
  - we have *a strong aversion* to PsychoPy …

…and this was us during multiple occasions of setting it up and creating some scripts:

[insert Angry Panda Trashing PCs GIF]

  - It is barely runnable on Linux
  - Not entirely compatible with Python 3
  - The documentation is confusing and incomplete

(ノ☉益☉)ノ彡┻━┻

Expyriment

Expyriment

- Expyriment is an alternative to PsychoPy
  - http://docs.expyriment.org/index.html
- It is better compatible with Python 3 and pretty easy to install and follow
  - At least that's our impression
  - In the end it's up to you what you want to use

┬─┬ノ(^_^ノ)

Expyriment - General Structuring

- Experiments in Expyriment are structured like this:
  - There is an experiment
    - The experiment consists of blocks
      - The blocks consists of trials

- So it’s good practice to follow this structure

0. Expyriment – Building the Frame

- Before we can get into any of our experiment creation steps, we need to build the frame of our experiment

```
from expyriment import design, control

# create experiment object
exp = design.Experiment(name="Cool Experiment")

# initialize experiment object and make it active experiment
# this will show a startup screen
# it will also initialize exp.screen, exp.mouse, exp.keyboard, exp.event and exp.clock
control.initialize(exp)

# this will present a subject number screen and a ready screen after initialization
# is completely finished
control.start()

# this will show an "ending experiment" screen and save data
control.end()
```

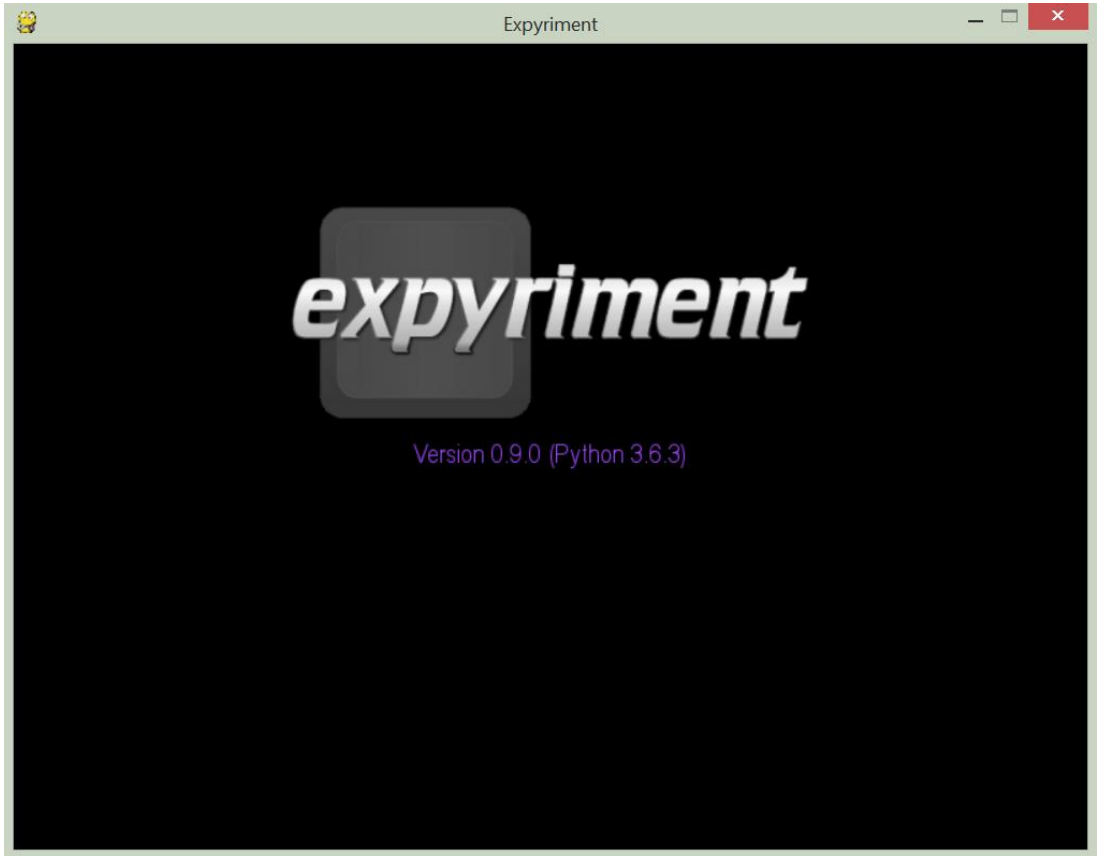
1. Expyriment– Graphic Surface

- Through our experiment initialization, the window is already there
- By default, however, it is a full screen window with a black background
  - Let's change that!

```
from expyriment import design, control

control.defaults.window_mode = True # True corresponds to windowed
control.defaults.window_size = [800,600] # 800x600 resolution
# we are going to change the default background color for this experiment
# however this can also be changed later after initialization using exp.screen.colour()
design.defaults.experiment_background_colour = (230,230,70)

exp = design.Experiment(name="Cool Experiment")

control.initialize(exp)
control.start()
control.end()
```

1. Expyriment– Graphic Surface

- Now everything is in a neat little window, but we won't see much of our newly set background color as the **experiment** itself **is still empty**
  - The startup screen is unaffected by our color change
- So we need to define and show some stimuli!



2. Expyriment - Stimuli

- We will, again, start with a point for fixation
  - But first, we update our import statement

```
from expyriment import design, control, stimuli
```

- To keep a clear structure, we want to attach this stimulus to a trial in a block
  - Therefore, we also need a block and a trial

```
# put this after initializing the experiment
block_one = design.Block(name="Our only block")
tmp_trial = design.Trial()
```

2. Expyriment - Stimuli

- Now we can create our stimulus!
  - Here, we even have a special object for fixation crosses

```
cross = stimuli.FixCross(colour=(0,0,0)) # set to black
cross.preload() # loads the stimulus to memory
```

- Then we attach this stimulus to the trial, the trial to the block and the block to the experiment

```
tmp_trial.add_stimulus(cross)
block_one.add_trial(tmp_trial)
exp.add_block(block_one)
```

- And finally, after starting the experiment, we can go through our blocks and trials and present the stimulus

```
# since right now we only have one block and one trial, this will only do
# one iteration each. but it's good to keep the loops because they can easily
# generalize for multiple trials or multiple blocks
for b in exp.blocks:
    for t in b.trials:
        # the cross is the first (and only) stimulus in this trial
        t.stimuli[0].present()
```

2. Expyriment - Stimuli

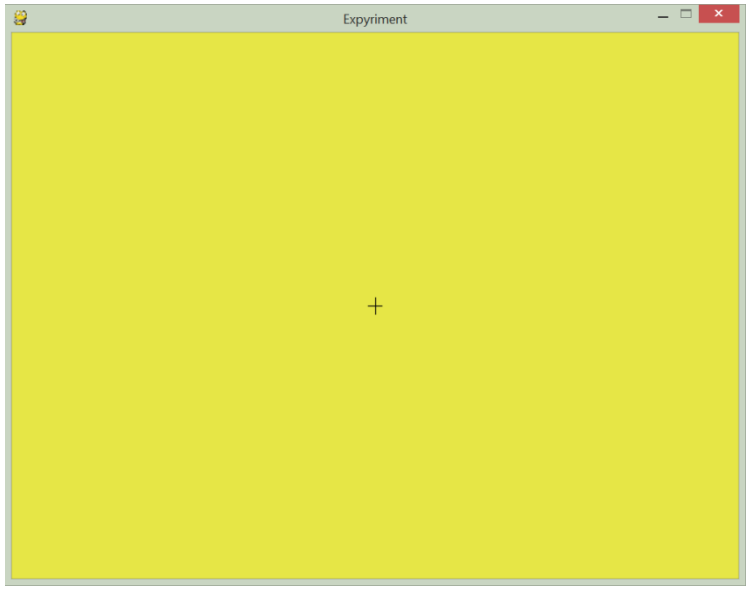
- For now, we don't need to flip the buffers for the stimulus to be shown
- The reason for this lies in the function definition for `present()`:

```
present(clear=True, update=True, log_event_tag=None)
```

- The function has two attributes that will be important very soon:
  - `clear` will, if set to `True`, clear the buffer before drawing
  - `update` will, if set to `True`, flip the buffers after drawing
- So the buffers are already implicitly flipped for us!

2. Expyriment - Stimuli

- So right now, our program looks like this
  - Theoretically. We wouldn't see it because it, again, immediately closes after opening
  - But we will fix this in a second
  - For now, believe us it's there




```
from expyriment import design, control, stimuli

control.defaults.window_mode = True
control.defaults.window_size = [800,600]
design.defaults.experiment_background_colour = (230,230,70)

exp = design.Experiment(name="Cool Experiment")
control.initialize(exp)

block_one = design.Block(name="Our only block")
tmp_trial = design.Trial()

cross = stimuli.FixCross(colour=(0,0,0))
cross.preload()

tmp_trial.add_stimulus(cross)
block_one.add_trial(tmp_trial)
exp.add_block(block_one)
control.start()

for b in exp.blocks:
     for t in b.trials:
          t.stimuli[0].present()
          exp.keyboard.wait()

control.end()
```

2. Expyriment - Stimuli

- We decide to add another stimulus that will be moving around again
- As there is is no template for a grating stimulus in Expyriment, a circle will have to do

```
stim = stimuli.Circle(radius=25, colour=(0,0,0),position=[-100,0])
tmp_trial.add_stimulus(stim)
```

- For both stimuli to be displayed, we will have to change up the presenting part a bit

```
for b in exp.blocks:
    for t in b.trials:
        t.stimuli[0].present()
```

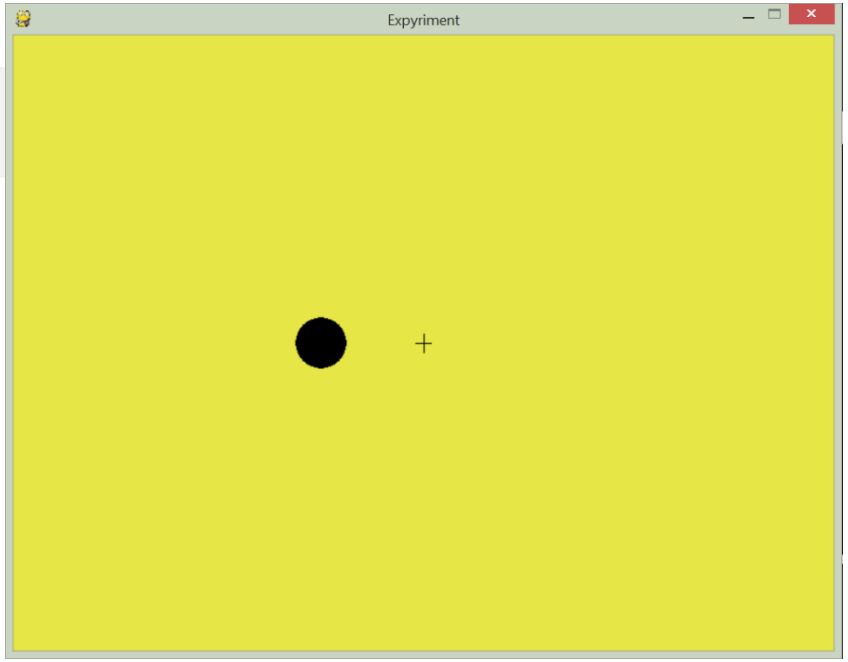
- As of now, not only will only the first of two stimuli in the trial be displayed, but also the buffer will be flipped directly after
  - Which means that the next object would be drawn on the other buffer
  - But we want both stimuli to be drawn on the same buffer

2. Expyriment - Stimuli

- Therefore, we add a line that presents the second stimulus and also change up the `present()` functions to
  - Clear the screen, then present the first stimulus, but do NOT flip the buffers
  - Do NOT clear the screen (we've just drawn the first stimulus on it), but draw the second stimulus, then flip the buffers

```
t.stimuli[0].present(clear=True, update=False)
t.stimuli[1].present(clear=False, update=True)
```

  - And voilà!



3. Expyriment - User Input - keyboard.wait()

- Let's take care of our window closing problem
- This is easy as the Expyriment equivalent for `waitKey()` works really similarly
  - It is named `keyboard.wait()` and can be called on the experiment object
  - Again, we can specify things like which keys to look for
  - Even fancier, this function will return the time it took the user to press the key in milliseconds!

```
for b in exp.blocks:
    for t in b.trials:
        t.stimuli[0].present(clear=True, update=False)
        t.stimuli[1].present(clear=False, update=True)

        exp.keyboard.wait()
```

3. Expyriment - User Input: Global Keys

- Again, we want to define a key with which we can quit the program
- There is actually an extra method called `set_quit_key()` for this
- 27 is the ASCII equivalent for the escape button, so we hand it over as a parameter

```
exp.keyboard.set_quit_key(27)
```

3. Expyriment – Code Overview

- So you have a reference of what the code looks like so far:

```
from expyriment import design, control, stimuli

control.defaults.window_mode = True
control.defaults.window_size = [800,600]
design.defaults.experiment_background_colour = (230,230,70)

exp = design.Experiment(name="Cool Experiment")
control.initialize(exp)

exp.keyboard.set_quit_key(27)

block_one = design.Block(name="Our only block")
tmp_trial = design.Trial()

cross = stimuli.FixCross(colour=(0,0,0))
cross.preload()

stim = stimuli.Circle(radius=25, colour=(0,0,0),position=[-100,0])
stim.preload()

tmp_trial.add_stimulus(cross)
tmp_trial.add_stimulus(stim)
block_one.add_trial(tmp_trial)
exp.add_block(block_one)
control.start()

for b in exp.blocks:
     for t in b.trials:
                t.stimuli[0].present(clear=True, update=False)
                t.stimuli[1].present(clear=False, update=True)

                exp.keyboard.wait()

control.end()
```

4. Expyriment – Experiment Procedures

- There are no handlers in Expyriment, so we will have to come up with the experiment procedures ourselves
  - The only thing left to do is make our stimulus move around
  - So we change up the code just a bit to allow for more trials
  - Note that we create a new `Circle` object each trial and make a deep copy of `tmp_trial()` before clearing it - we need to do this as Expyriment will only hand over references when adding stimuli to trials or trials to blocks
    - So if we didn't reinitialize the stimuli or trials, changing one of them would change all of them

```
cross = stimuli.FixCross(colour=(0,0,0))
cross.preload()

stim = stimuli.Circle(radius=25,
     colour=(0,0,0),position=[-100,0])
stim.preload()

tmp_trial.add_stimulus(cross)
tmp_trial.add_stimulus(stim)
block_one.add_trial(tmp_trial)
exp.add_block(block_one)
```

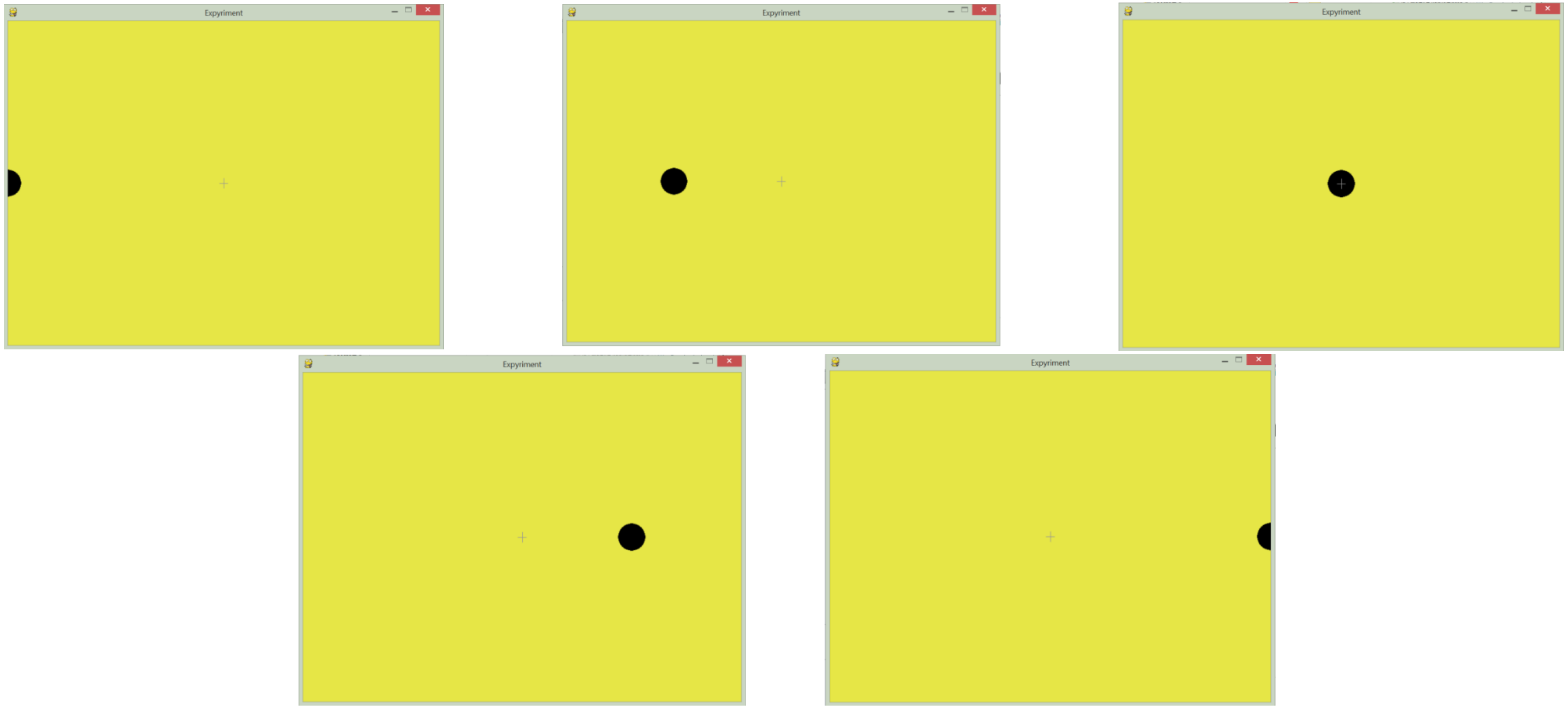
```
cross = stimuli.FixCross()
cross.preload()

# define stimulus positions
positions = [-400, -200, 0, 200, 400]

# go through all positions
for xpos in positions:
      # create circle accordingly
      stim = stimuli.Circle(radius=25, colour=(0,0,0),position=[xpos,0])
      stim.preload()

      # add both stimuli to the trial
      tmp_trial.add_stimulus(stim)
      tmp_trial.add_stimulus(cross)

      # add the trial to the block
      block_one.add_trial(tmp_trial)
      # make a deep copy of the trial and clear it for the next iteration
      tmp_trial = tmp_trial.copy()
      tmp_trial.clear_stimuli()
```

4. Expyriment – Complete Code

```
from expyriment import design, control, stimuli

control.defaults.window_mode = True
control.defaults.window_size = [800,600]
design.defaults.experiment_background_colour = (230,230,70)

exp = design.Experiment(name="Cool Experiment")
control.initialize(exp)

exp.keyboard.set_quit_key(27)

block_one = design.Block(name="Our only block")
tmp_trial = design.Trial()

cross = stimuli.FixCross(colour=(0,0,0))
cross.preload()

cross = stimuli.FixCross()
cross.preload()
```

```
positions = [-400, -200, 0, 200, 400]

for xpos in positions:
      stim = stimuli.Circle(radius=25,
      colour=(0,0,0),position=[xpos,0])
      stim.preload()
      tmp_trial.add_stimulus(stim)
      tmp_trial.add_stimulus(cross)
      block_one.add_trial(tmp_trial)
      tmp_trial = tmp_trial.copy()
      tmp_trial.clear_stimuli()

exp.add_block(block_one)
control.start()

for b in exp.blocks:
     for t in b.trials:
            t.stimuli[0].present(clear=True, update=False)
            t.stimuli[1].present(clear=False, update=True)

            exp.keyboard.wait()

control.end()
```

4. Expyriment - The Program



5. Expyriment - Data Logging

- Like with PsychoPy, we will leave it to you if you want to use the usual Python file handling methods or the ones specific to Expyriment
- You can, of course, find more information in the documentation
  - Especially http://docs.expyriment.org/expyriment.io.html provides multiple classes working with files

Expyriment – Code Structure

```
from expyriment import design, control, stimuli

control.defaults.window_mode = True
control.defaults.window_size = [800,600]
design.defaults.experiment_background_colour = (230,230,70)

exp = design.Experiment(name="Cool Experiment")
control.initialize(exp)

exp.keyboard.set_quit_key(27)

block_one = design.Block(name="Our only block")
tmp_trial = design.Trial()

cross = stimuli.FixCross(colour=(0,0,0))
cross.preload()

cross = stimuli.FixCross()
cross.preload()
```

First, change defaults (if needed)

Then, create and initialize experiment

Here, we have room for experiment-specific configurations

Then, define Block and Trial objects

Then, construct your blocks, trials and stimuli

```
positions = [-400, -200, 0, 200, 400]

for xpos in positions:
      stim = stimuli.Circle(radius=25,
      colour=(0,0,0),position=[xpos,0])
      stim.preload()
      tmp_trial.add_stimulus(stim)
      tmp_trial.add_stimulus(cross)
      block_one.add_trial(tmp_trial)
      tmp_trial = tmp_trial.copy()
      tmp_trial.clear_stimuli()

exp.add_block(block_one)

control.start()

for b in exp.blocks:
     for t in b.trials:
            t.stimuli[0].present(clear=True, update=False)
            t.stimuli[1].present(clear=False, update=True)

            exp.keyboard.wait()

control.end()
```

Then, start the experiment

Then, define your procedures

Finally, end the experiment

Structured Data

Data So Far

- So far we have mostly considered *simple text* data
- Just a collection of strings that were saved in the same file
  - In the hangman example we structured data, by having one word per line
- But this is often not enough to display more complex data structures
  - E.g. the data of several trials of one subject, or image data with labels, etc.
- One option would be to store each entry in a single file
  - But this is hard to keep track off, slow to process and a mess to organise
- This is why there are several **structured data types**
  - In the following we will have a look at some of those
  - Even though it will still be just strings, they are in *certain order* now

Why Structure Data?

- It provides a director way of handling data structures as present in programming, often at the cost of human readability
- It makes reading and writing data streamlined
- This makes data *transferable* and *(re)?usable* (with 3rd party tools|)
- Standards make collaboration and handling easier
- There are *many* standards with different advantages and disadvantages
- We will look at two of the (arguably) more prominent ones: CSV and JSON

CSV – What It Is

- **CSV** stands for **C**omma **S**eparated **V**alues
- It is a *plain text representation* of *tabular data*
  - Since a table is a graphical object, we need some way of putting it into a text representation
- The first row contains the names of the fields
- Each *row* represents *one record* in the table
- All records **must** have the same number of fields
- The *values* of the fields are *separated* by a *comma*

data_log.csv

```
timestamp, level, err_id, message
20180619T160252, ERROR, 404, Site not found
20180619T160334, WARNING, 37, div overflow
20180619T162732, ERROR, 408, timeout
```

timestamp	level	err_id	message
20180619T160252	ERROR	404	Site not found
20180619T160334	WARNING	37	div overflow
20180619T162732	ERROR	408	timeout

CSV – What It Is

- If a comma is in a field value, that value is often enclosed in quotation marks like a string
- If we then need to use quotation marks in the value, we need to escape those
  - You can get around this by using different quotation marks

data_log.csv

```
timestamp, level, err_id, message
20180619T160252, ERROR, 404, "Site not found,
http://localhost:8088/db"
20180619T160334, WARNING, 37, div overflow
20180619T162732, ERROR, 408, "timeout, 10
\"sec\""
```

timestamp	level	err_id	message
20180619T160252	ERROR	404	Site not found, http://localhost:8088/db
20180619T160334	WARNING	37	div overflow
20180619T162732	ERROR	408	Timeout, 10 “sec”

CSV - Advantages

- It is a very lightweight format
  - The file size of a csv file is often smaller than the same data in a comparable format
- It is easily streamable / separable
  - Since the content can be examined row by row, there is no need to load the whole file to process it
  - This makes it useful if you are limited on working memory or need to transmit it via a network
- It is straightforward to process and still fairly readable
- Is used commonly in (data) science or financing, thus there exists an abundance of software for it
- It follows a certain schema, so that each record is similar

CSV - Disadvantages

- There are many files, that call themselves csv, but are not obliging to the rules
  - Some use tabs, spaces or semicolons as separators instead of commas
  - Some use different quotes or escape characters
- That makes dealing with csv files unreliable, at times as files have to be checked manually first so that parameters can be set accordingly
- While many things *can* be displayed in a tabular format, it is not optimal for all data
  - Especially hierarchical or dictionary structures are not well represented

experiment_data.csv

```
subject_id, stim_id, stim_onset, subj_react
0, 0, 5123.21, 5127.19
0, 1, 5352.91, 5362.01
1, 0, 3213.22, 3214.10
1, 1, 4244.76, 4246.82
```

trees.csv

```
id, parent_id, children_ids, content
0, na, "[1,2,3]", "I. Am. Root."
1, 0, "[4]", "content_id: 2"
2, 0, "[]", "content_id: 23"
3, 0, "[]", na
4, 1, "[]", "content_id: 32"
```

JSON – What It Is

- **JSON** stands for **J**ava**S**cript **O**bject **N**otation
- It is a *plain text representation* of *objects* or *object data*
- It is very similar to the Python dictionary
- A JSON file always consists of a single **JavaScript object**
  - This means it **starts** with a **curly brace** and **ends** with **one**. {} is a valid object
- This object consists of unordered **key : value** pairs, with a **colon** between key and value
  - The key is a unique string in double-quotes
  - The value can be one of many things
- The key-value pairs are separated by **commas**

```
{
  "colors": [
    {
      "color": "black",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,255,255,1],
        "hex": "#000"
      }
    },
    {
      "color": "white",
      "category": "value",
      "code": {
        "rgba": [0,0,0,1],
        "hex": "#FFF"
      }
    }
  ]
}
```

JSON – What It Is

- The value can be
  - A string (in double-quotes)
  - A number
    - int or float works, as well as scientific notation
  - Another JavaScript object
  - An array
    - Values separated by commas in square brackets akin to a Python list
  - The Boolean values true and false
    - Not capitalised unlike Python
    - They do not need quotation marks
  - null
    - The same as None in Python

```
{
  "colors": [
    {
      "color": "black",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,255,255,1],
        "hex": "#000"
      }
    },
    {
      "color": "white",
      "category": "value",
      "code": {
        "rgba": [0,0,0,1],
        "hex": "#FFF"
      }
    }
  ]
}
```

JSON – Advantages

- It is also a fairly lightweight format
  - Not as lightweight as csv, but more powerful
- It can represent hierarchical data structures (to a certain extent), as well as different data types
- It has code counterparts (Python dict, JS object, C/C++/Java HashMap, …), and thus straightforward to process and still readable
- It is schema free, meaning you can expand it more dynamically
- It is used commonly throughout the web, (data) science and especially computer science
- There exist databases built around/with JSON, so called document based db

JSON – Disadvantages

- Can get convoluted quickly and thus hard to read by a human
- Always has to be loaded in full to be parsed
- Still not expressive enough for complex data structures
- It is schema free, meaning that each record inside can vary
  - This can make processing between systems difficult or lead to wrong processing
  - Or at the very least makes verifying harder

Image Data – A Short Word

- Digital images are just a series of numbers
- They are a grid structure, where each cell is one pixel of the image
  - A ndarray from numpy for example
- Each pixel then has its colour information
  - As seen in the JSON example colour can be represented in different values
    - The ones to know are probably RGBA and hexcodes
- Therefore an image is a 3d array
  - The first two dimensions are the pixel grid
  - The third dimension is the colour information
- Hence, we can perform all kinds of mathematical operations on images, like we would on any array

A cute yawning cat
taken from pixnio.com

Honourable Mentions

XML	One of the most used formats with static types, namespaces and hierarchy.
Database formats (SQL, Documents, Graphs, etc.)	The most complex, but also the most powerful of the bunch. They allow complex data structures and operations with good speed.
HTML	HyperText Markup Language. One of the backbones of the www
Office Document Formats (docx, odt, xlsx, pptx, rtf, etc.)	Often a form of xml, they can be used to store all kinds of data, like formatted text, tabular data, …
PDF	Made to be portable, but a pain to work with
Audio (mp3, wav, etc.)	Digital audio, just like pictures, is a bunch of numbers. Fairly nice to work with.
Video (mpeg4, ogg, etc.)	Videos are like images, but many of them. Still nice to work with.
MAT	Matlab formatted data. A binary proprietary format that can contain pretty much anything. Not humanly readable.
BLOB	Binary Large OBjects are binarized data files. Often they are compressed versions of the file formats above.

Working on Data: Pandas

Moritz Nipshagen mnipshagen@uos.de
Antonia Hain ahain@uos.de

Pandas

"pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language." pandas.pydata.org

- Pandas is a library built on top of numpy to further increase the capability of working with data
- It provides functionality to work with labelled data, time series and is robust against heterogeneity and missing data
- Pandas implements the data structures that are also present in R, which is great for inspecting, cleaning and analysing data
- If you did not already you can install Pandas with conda or pip
- In Python you can use pandas as `import pandas as pd`
  - Code in the following will presume imports of `pandas as pd`, `numpy as np`, and `pyplot as plt`

Pandas Datatypes: Series

- A Pandas Series[1] is a one-dimensional array of indexed data
- At first, this might sound like a simple numpy ndarray, but the key difference is that *indexed* part
- We can create a series from any object that can be a ndarray, and we can optionally provide an index list of same length
- We can then index the series not just by numbers, like we are used to, but using the index values as well

```
class pandas.Series(data=None, index=None, dtype=None,
    name=None, copy=False, fastpath=False)
```

```
data = pd.Series([0.5, 1, 2, 2.5])
print(data, data[1], data[1:3], "", sep="\n")

index = ['a', 'b', 'c', 'z']
data.index = index
print(data, data['a'], sep="\n")

Output:
0    0.5
1    1.0
2    2.0
3    2.5
dtype: float64
1.0
1    1.0
2    2.0
dtype: float64

a    0.5
b    1.0
c    2.0
z    2.5
dtype: float64
0.5
```

[1]https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

Pandas Datatypes: Series

- So instead of thinking of Series as an array, we might think of it more akin to a dictionary
  - It is a generalisation of an array, and a specialisation of a dict
- With the index being any kind of value and referring to one value, it is reminiscent
- But differently to a dict, order is preserved
- We can slice the series even using index values

```
population_dict = {'Germany': 80716000,
  'France': 67210000, 'UK': 64100000, 'Sweden': 10004962}

population = pd.Series(population_dict)
print(population, population['Germany'],
  population['France':'UK'], sep="\n")
```

Output:

```
Germany    80716000
France     67210000
UK         64100000
Sweden     10004962
dtype: int64
80716000
France    67210000
UK        64100000
dtype: int64
```

Pandas Datatypes: Dataframes

- A **dataframe**[1] (*df*) is a collection of series, creating a tabular data structure
- Each column is one series and each row is one record with one value for each column
- A *df* also has an index, and all series / columns of the df share this same index
- Like series were generalisations of 1d arrays, *dfs* are generalisations of 2d arrays
- And like series were specialised dictionaries, *dfs* are specialised dictionaries with a column name as a key and a series as a value

```
area_dict = {'Germany': 357168, 'France': 643801,
  'UK': 243610, 'Sweden': 449964}
area = pd.Series(area_dict)

df = pd.DataFrame({'population': population,
  'area': area})
print(df, df['area'], df.loc['France'], sep="\n\n")
```

Output:

```
            population    area
Germany       80716000  357168
France        67210000  643801
UK            64100000  243610
Sweden        10004962  449964

Germany     357168
France      643801
UK          243610
Sweden      449964
Name: area, dtype: int64

population    67210000
area            643801
Name: France, dtype: int64
```

```
class pandas.DataFrame(data=None, index=None,
  columns=None, dtype=None, copy=False)
```

[1] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

Pandas Datatypes: Dataframes

- Accessing a column returns a Series object
- You can also use a list of dictionaries to create a dataframe, where each dict is one record
  - Remember how we used dicts to represent object data like albums? Now we can make those into one database
- The dataframe will always use all present keys, even if the keys are not present in all supplied column data
- Any 2d numpy array can be made into a dataframe

```
df = pd.DataFrame({'population': population,
  'area': area})
print(type(df['population']))

peter = {'name': 'Peter', 'fav_colour': 'blue'}
susan = {'name': 'Susan', 'fav_colour': 'green', 'pet': 'Oscar'}
df = pd.DataFrame([peter, susan])
print(df)

df = pd.DataFrame(np.random.rand(3, 2),
  columns=['foo', 'bar'], index=['a', 42, True])
print(df, df.columns, df.values, df.index, sep="\n")


Output:
<class 'pandas.core.series.Series'>
  fav_colour   name    pet
0       blue  Peter    NaN
1      green  Susan  Oscar
           foo       bar
a     0.335329  0.181496
42    0.632545  0.201658
True  0.113261  0.744242
Index(['foo', 'bar'], dtype='object')
[[0.33532876 0.18149577]
 [0.63254465 0.20165753]
 [0.1132611  0.74424223]]
Index(['a', 42, True], dtype='object')
```

Pandas Datatypes: Index

- Series and DataFrame both have an index
- This index[1] is another pandas object, which is a one-dimensional array
- They share many attributes with numpy's ndarrays like size, shape, ndim and dtype
- But index objects are immutable and they are ordered sets
  - You cannot assign single values of an index
  - All entries are unique
  - The order of the entries is preserved
- It is a good idea to only have indices of the same type

```
idx = pd.Index(['a', 'b', 'c']) # stick to one
print(idx[1], idx[0:1:-1], idx.shape, idx.dtype)
idx = pd.Index([1, 2, 'dog']) # don't mix ints and strings
idx[2] = 3 # oh no index is immutable


Output:
b Index([], dtype='object') (3,) object
Traceback (most recent call last):
  File
"e:/Libraries/OneDrive/Uni_encrypted/Orga/Course/SS2018/BP
P/2018/12/code.py", line 29, in <module>
    idx[2] = 3 # oh no index is immutable
  File "C:\Users\nipsh\Anaconda3\lib\site-
packages\pandas\core\indexes\base.py", line 2051, in
__setitem__
    raise TypeError("Index does not support mutable
operations")
TypeError: Index does not support mutable operations
```

```
class pandas.Index(data=None, dtype=object,
copy=False, name=None, tupleize_cols=True)
```

[1]https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Index.html

Loading Data

- Pandas implements functionality to load from a lot of different data sources
- From pickle, csv, your clipboard, excel, json, html, sql, Google BigQuery, and some more
- You can easily handle these formats using pd.read_<file_format>(filepath_or_buffer)
  - df = pd.read_csv(“my_csv_file.csv”)
  - df = pd.read_json(open(“my_file.json”))
- Pandas automatically uses the header row as column names, and then creates a dataframe

Working With Data

- We have our data loaded and are all setup, so now we can start working
- The procedure would be
  - Collect the data
    - Either by an experiment, or from other sources, depending on what you are doing
  - Extracting & inspecting the data
    - see if there were loading issues or if the dataframe looks good, and if the data looks homogeneous
  - Visualising the data
    - Plotting the data can really help to spot regularities and get a general idea about your data (e.g. whether there is a correlation, how the data is spread, etc.)
  - Analysing the data
    - Fitting a statistical model, checking variances, means, and other measures as well, to understand the metric of your data and maybe draw conclusions, as well as…
  - Update the data collection
    - Change the experiment setup to control for more variables, exclude certain data sets when scraping, etc…. And repeat.

Inspecting Data

- We have now extraced our data from files or other objects and are ready to look at it
- To peek into a dataframe or series we can use `data.head(n=5)`
  - It returns the first n rows, so you can make sure the format is correct
- We can use the same attributes as in numpy to check the shape of the data
  - size, shape, the transpose, etc.

```
df = pd.DataFrame({'population': population,
    'area': area})
print(df.head())
print(df.size, df.shape, df.T)


Output:
 population    area
Germany    80716000  357168
France     67210000  643801
UK         64100000  243610
Sweden     10004962  449964
8 (4, 2)              Germany    France        UK
Sweden
population  80716000  67210000  64100000  10004962
area          357168    643801    243610    449964
```

Inspecting Data

- A handy method on dataframes is info()
  - It displays a summary of column names, data types and missing data

```
DataFrame.info(verbose=None, buf=None, max_cols=None,
    memory_usage=None, null_counts=None)
```

- There exists another powerful function: `describe()`
  - It outputs even some statistical meausrements

```
Series.describe(percentiles=None,
    include=None, exclude=None)
```

```
print(df.info(), "\n")
print(df.area.describe())

Output:
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, Germany to Sweden
Data columns (total 2 columns):
population    4 non-null int64
area          4 non-null int64
dtypes: int64(2)
memory usage: 256.0+ bytes
None

count         4.000000
mean     423635.750000
std      169305.588123
min      243610.000000
25%      328778.500000
50%      403566.000000
75%      498423.250000
max      643801.000000
Name: area, dtype: float64
```

Locating Data

- When the data looks good, we might want to extract certain apsects or data fields
- We already know how to index lists and arrays
- We will now look at how flexible Pandas is when indexing a Series or a DataFrame
  - And what to look out for

```
a = np.arange(16).reshape(4, 4)
print(a)
# Takes those values of the second and
# fourth column that are divisible by 3
print(a[:, [1, 3]][a[:, [1, 3]] % 3 == 0])

Output:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[ 3  9 15]
```

Locating Data: Series

- As said before, a Series object behaves much like an overlap between a numpy ndarray and a Python dictionary
- Like a dictionary a Series provides a mapping from key to value
- We can access any item in the Series by indexing it as we would with a dict
- In fact we can even call the dict method `keys()` on it

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
    index=['a', 'b', 'c', 'd'])
print(data)
print('b' in data)
print(data.keys() == data.index)
print(list(data.items()))
data['e'] = 1.25
print(data)

Output:
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
True
[ True  True  True  True]
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

Locating Data: Series

- But since Series also are like arrays, we can index and slice them as such
- For this we can use the ***explicit* index**
  - So the values inside the Series.index
- Or the **implicit integer index**
  - How we would index a “normal” array
- Remember, that when indexing lists and arrays the *end* of the slice is *exclusive*
- For Series this only applies if using the *implicit integer index*, but when we use the *explicit index* the end is **inclusive**
- We can also use a list of indices to extract several specific values
- If the index of a Series already is an integer list, the *slice* will use *implicit indexing*, but a direct access will use *explicit indexing*

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
    index=['a', 'b', 'c', 'd’])
print(data['a':'c']) # explicit: inclusive slice
print(data[0:2]) # implicit: exclusive slice
print(data[['a','e']) # fancy!
data.index = [1, 2, 3, 4, 5]
print(data[1])
print(data[0:2])

Output:
a    0.25
b    0.50
c    0.75
dtype: float64
a    0.25
b    0.50
dtype: float64
a    0.25
e    1.25
dtype: float64
0.25
1    0.25
2    0.50
dtype: float64
```

Locating Data: loc & iloc

- Since that behaviour is somewhat counterintuitive, there is a workaround
- Every Series has the attributes `loc` and `iloc`
- **`iloc`** makes it possible to **always** use the **implicit integer index** just like we are used to
- **`loc`** makes it possible to **always** use the **explicit index** which means inclusive slices and fancy indexing
- **Always use loc and iloc!**
  - Making explicit what you want to do, is always the preferred (and readable) way

```
print(data, "\n")
print(data.loc[1],
data.loc[1:3], "\n",
data.iloc[1],
data.iloc[1:3])

Output:
1    0.25
2    0.50
3    0.75
4    1.00
5    1.25
dtype: float64

0.25 1    0.25
2    0.50
3    0.75
dtype: float64
 0.5 2    0.50
3    0.75
dtype: float64
```

Locating Data: DataFrame

- Most of what we said about Series applies to DataFrames as well
- Since we now have two dimensions, indexing is ambiguous
- **Accessing** a DataFrame by index refers to the **columns**
- **Slicing** a DataFrame refers to the **rows**
- DataFrames also supply the `loc` and `iloc` attributes

```
population_dict = {'Germany': 80716000,
  'France': 67210000, 'UK': 64100000, 'Sweden': 10004962}
population = pd.Series(population_dict)
area_dict = {'Germany': 357168, 'France': 643801,
  'UK': 243610, 'Sweden': 449964}
area = pd.Series(area_dict)

df = pd.DataFrame({'population': population,
  'area': area})
print(df['area'])
print(df['Germany':'UK'])


Output:
Germany    357168
France     643801
UK         243610
Sweden     449964
Name: area, dtype: int64

         population    area
Germany    80716000  357168
France     67210000  643801
UK         64100000  243610
```

Locating Data: loc & iloc

- When dealing with DataFrames, loc and iloc take two indices instead of one
- The *first* indexes the *rows* and the *second* indexes the *columns* of the DataFrame
- Again `iloc` is the implicit integer indexing and the end of slicing is exclusive
- And `loc` is the explicit indexing and the end of slicing is inclusive
- This way we can also access single rows
- And again: **Always use loc and iloc!**

```
print(df.iloc[1:3, :2])
print(df.loc['France':'Sweden', :'area'], "\n")
print(df.loc['UK', :])


Output:
        population    area
France    67210000  643801
UK        64100000  243610
        population    area
France    67210000  643801
UK        64100000  243610
Sweden    10004962  449964

population    64100000
area            243610
Name: UK, dtype: int64
```

Boolean Indexing

- You can create a mask to only select cells that fit a certain criteria
- This way you can easily and controlled update or delete values
- This is also useful to extract a subset of your Data

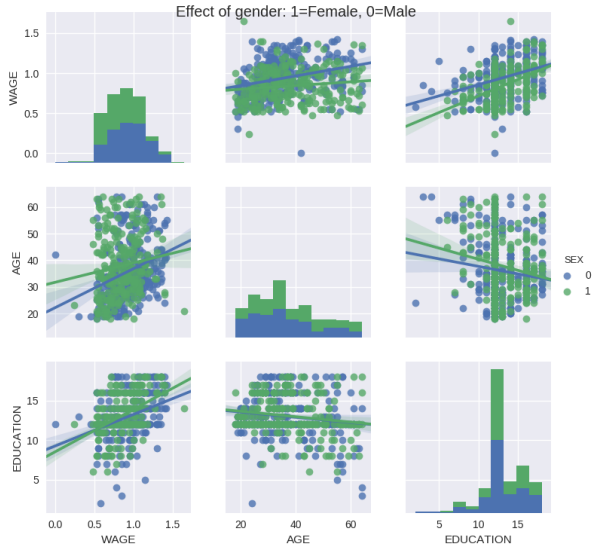
```
print(df['area'] > 400000)
print(df[df['area'] > 400000])


Output:
Germany    False
France      True
UK         False
Sweden      True
Name: area, dtype: bool
        population    area
France    67210000  643801
Sweden    10004962  449964
```

Resources

- If you want to read up more on Data Science in Python I strongly recommend having a look at “Python Data Science Handbook”
- It is a great collection of resources to understand and work with data
  - It covers some topics we talked about like NumPy, Matplotlib and Pandas, but in greater detail
- And they provide it for free! In Jupyter Notebooks on GitHub
- github.com/jakevdp/PythonDataScienceHandbook/tree/master/notebooks
  - Have a look around, and read upon some stuff if you are interested

Statistics

- Together with the statsmodels library, Pandas gets near full R functionality
- Including
  - Model fitting
    - Linear and multiple regression
  - Statistical analysis
  - Plotting and visual analysis
  - And much more



- There is nice scipy tutorial blog on this: www.scipy-lectures.org/packages/statistics/index.html#linear-models-multiple-factors-and-analysis-of-variance
  - It is a bit fast paced but still a good reference and/or starting point

Homework

Experiment!

- This week you will be tasked with creating a small experiment
  - Collecting some data
  - And plotting / visualising this data
  - No worries, there will be more guidelines than usual

See you all next week!