# Programming with PsychoPy, 1 Introduction

# An introductory programming guide for time-accurate experiments

Authors:
Daniel von Rhein (mailto:"Daniel.vonRhein@gmail.com")
Pascal de Water (mailto:"P.deWater@donders.ru.nl")
Wilbert van Ham (mailto:"W.vanHam@socsci.ru.nl")

This course was written by Daniel von Rhein, Pascal de Water and Wilbert van Ham for the Faculty of Social Sciences of the Radboud University Nijmegen. It is an adaptation of the course written by the first two authors, for Neurobs Presentation for the Donders Institute.

# Introduction (Aim of the course)

PsychoPy is a multi platform (Windows, Linux, Mac) based programming tool that allows experimenters to set up and program all sort of experiments. It is the recommended software for time-accurate experiments and therefore supported by the institutes (Faculty of Social Sciences, TSG) technical support group.

The TSG offer (PhD) students a couple of preprogrammed experiments (i.e. templates), which can be adjusted to build up own experiments. In this way, the PhD student can efficiently program experiments fitting an technical optimal environment.

This is what this course is all about. It aims at teaching programming skills, which are needed to modify the existing templates such that they meet your own demands. Because this can be quite complex, we start up with short assignments, which address one basic and simple problem at a time. They will all contribute to the final assignment in which you will work on an existing template.

# Installation

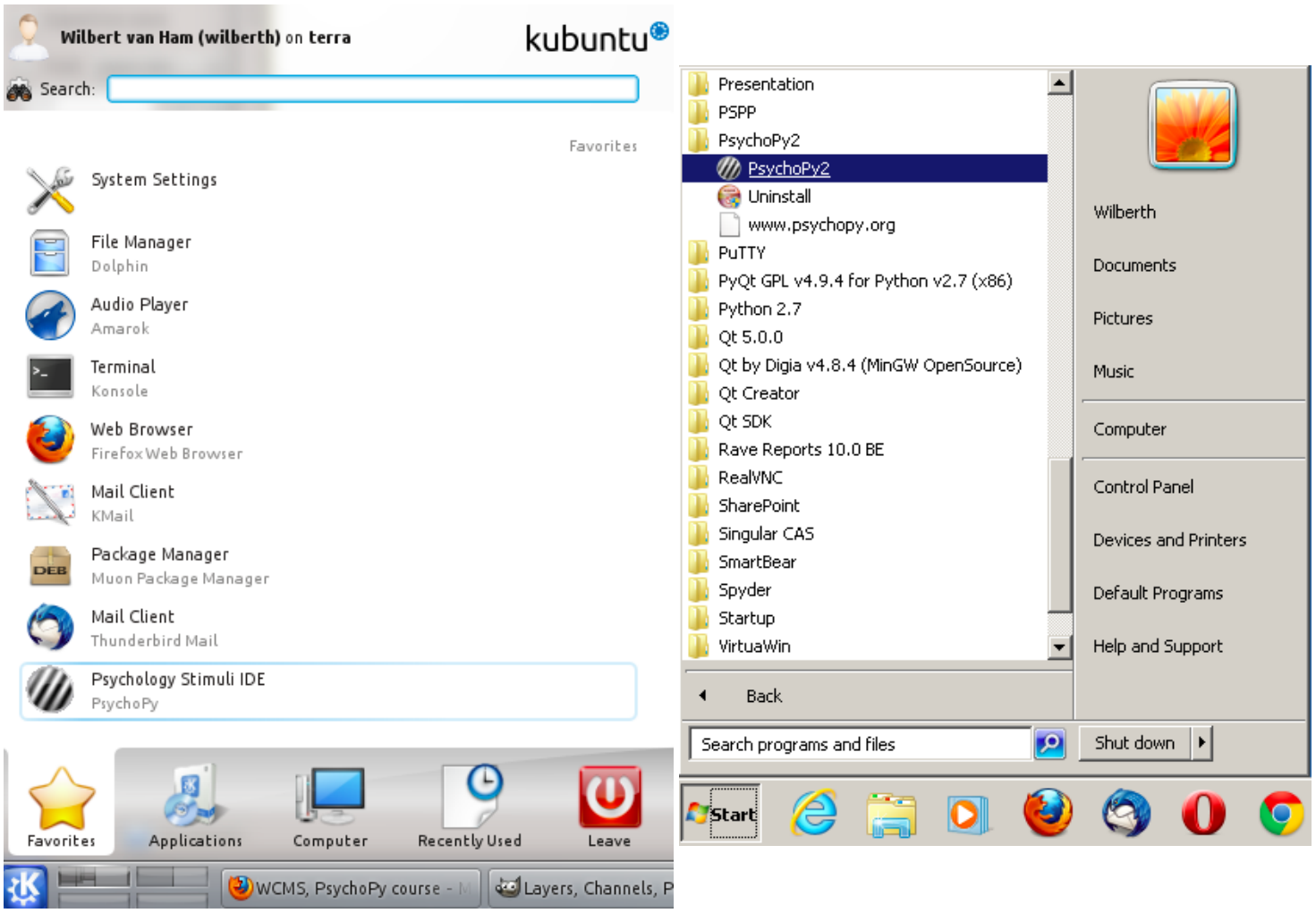
Installing PsychoPy on lab computers is done for you. Do not attempt to install it yourself or to make changes to an existing installation. Ask lab support for help if you need to do more than what is possible with the installed software. Lab support will make sure all monitor, sound card and interface settings are correct. Skip the *installation* section if you are using a lab computer.

Instructions on how to install psychopy on your own computer can be found here

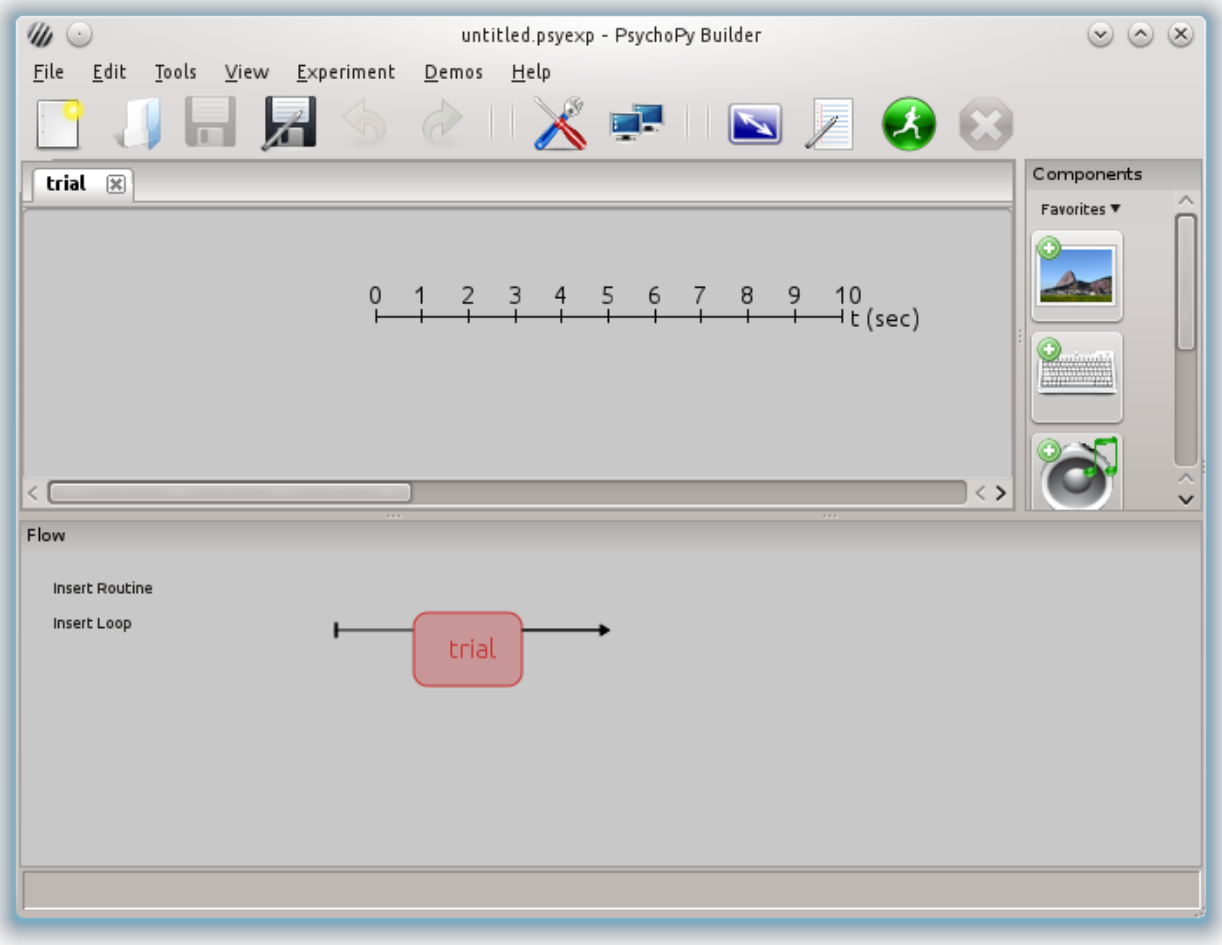
If you want to use the Buttonbox or the red Joystick for your experiment, you also need to install the RuSocSci package.

# Starting the software

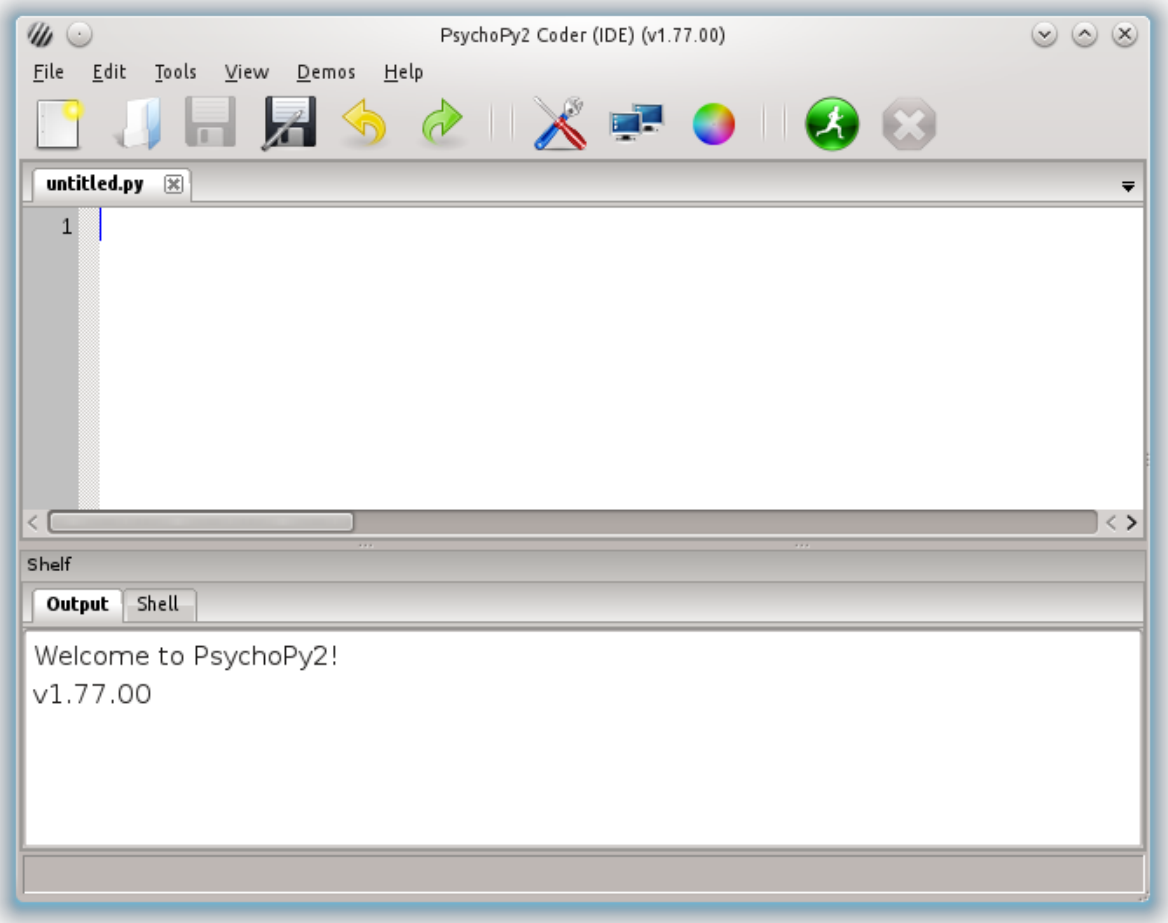
Start the PsychoPy program using whatever way you usually start programs on your system.




Now the PsychoPy program will start. The window that we see is called the *builder view*.




We do not use the builder. Switch to *coder view* with *ctrl-l* or with *View -> Open Coder view*.




The *coder* view is really an editor in which you can write your experiment. There is a *run* button on top (the button with the running man). Press it to start your experiment. You can also use your own preferred programming environment (IDE/editor). You do not need to start the PsychoPy program to start your experiment. After saving the experiment you can start it by double clicking it in Windows Explorer, Mac OS Finder, KDE Dolphin, ...

Continue with the next lesson

# Programming with PsychoPy, 2 Hello World

In this chapter we will explain your very first experiment line by line. But before we start I would like to share some knowledge of professor Donald E. Knuth with you. This may be the first time that you program a computer. The computer will often say that you are wrong. But do not worry. It is not angry with you.

# Experiment header, what will we use

Experiments in PsychoPy are really computer programs written in the Python programming language. To make sure that the computer understands this and understands which parts of PsychoPy to make available, we must start our experiment with a header. Note that all lines starting with a # , except for the very first two lines, are actually comments, which do not do anything.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# The first line is called the hash bang. It helps operating systems
# understand what to do with this script. The second line describes
# the encoding of this file. UTF-8 is the only one used nowadays.
# It supports all alphabeths, even runes and linear B.

# Import the PsychoPy libraries that you want to use
from psychopy import core, visual
```

# Stimulus Definition: What to present

For most experiments, we want to present visual or acoustic stimuli or a combination of both. It is also possible to present videos, although this topic is outside the scope of this course. Thus, what we need to do is to define visual and acoustic stimuli.

For a visual stimulus, we have a *Window* on which we present stimuli. This window can be a full screen, a window with borders around it, or a videowall the size of a football stadium. On this window we present some stimulus, for instance a text stimulus that says *Hello World!* In PsychoPy a visual stimulus definition would look like this:

```
# Create a window
win = visual.Window([400,300], monitor="testMonitor")

# Create a stimulus for a certain window
message = visual.TextStim(win, text='Hello World!')
```

As we can see, our window has the name *win*, it is 400 pixels wide and 300 pixels high. Our visual stimulus has the name *message*. it defines, what our stimulus consists of. In this case, it consists of the text *Hello World!*

# Control over stimuli: When to present

Two more steps are required to present our stimulus on the screen.

```
# Draw the stimulus to the window.
message.draw()

# Flip backside of the window.
win.flip()

# Pause 5 s, so you get a chance to see it!
core.wait(5.0)
```

First we draw message on its window back buffer, then we flip the back and front buffer. The flipping is the moment that the text is really presented.

The function *flip()* puts the picture immediately on the screen. We would see it appearing, however, it wouldn't stay there for long (just for a fraction of a second). We explicitly have to tell the program to keep on running without doing anything. Another built-in function doing exactly that is core.wait(). It keeps the program as it is and waits for the amount of seconds we command to wait.

# Cleaning up, closing window and experiment.

Finally we close the window and close the experiment.

```
# Close the window
win.close()

# Close PsychoPy
core.quit()
```

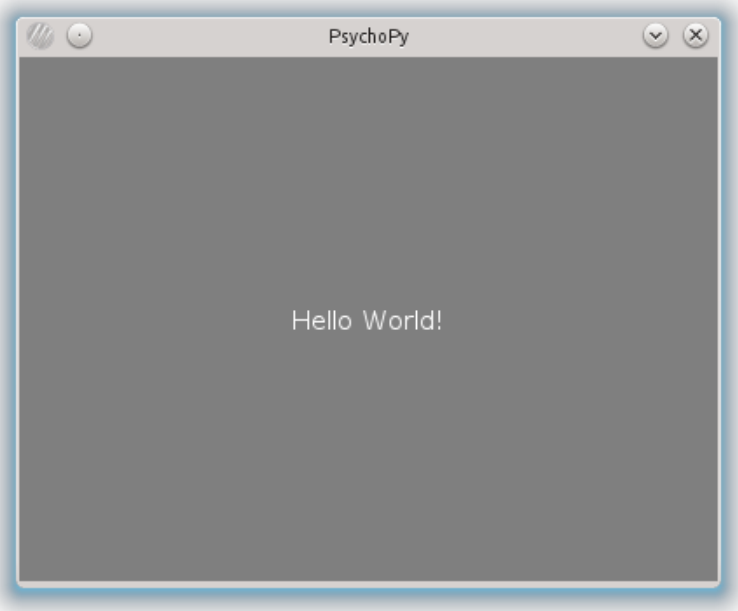
# Assignment 1: Putting it all together

Lets put it all together. Copy this experiment into PsychoPy and run it.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# The first line is called the hash bang. It helps operating systems
# understand what to do with this script. The second line describes
# the encoding of this file. UTF-8 is the only one used nowadays.
# It supports all alphabeths, even runes and linear B.

# Import the PsychoPy libraries that you want to use
from psychopy import core, visual

# Create a window
win = visual.Window([400,300], monitor="testMonitor")

# Create a stimulus for a certain window
message = visual.TextStim(win, text='Hello World!')
```

```
# Draw the stimulus to the window. We always draw at the back buffer of the
message.draw()

# Flip back buffer and front  buffer of the window.
win.flip()

# Pause 5 s, so you get a chance to see it!
core.wait(5.0)

# Close the window
win.close()

# Close PsychoPy
core.quit()
```

It should give you the following stimulus:




# Assignment

Can you change the text? Change the waiting time? More difficult: Can you first present the original text stimulus, then change the text and present the new stimulus for 3 seconds? You can look up how to change the text of an existing stimulus in the API documentation.

Continue with the next lesson

# Programming with PsychoPy, 3 Programming

That went pretty quick. We have written our first experiment in PsychoPy and it works. As is always the case, we will need some background knowledge before we can continue with more complex experiments. This lesson is about programming: variables, lists and control statements.

# Variables and lists

## Variables

Variables are place holders to store values. You can choose almost any name for a variable, although it is common to use names that clearly indicate what they actually do. And it makes the code more readable. Examples of variables are:

```
i = 17
peterSmith = 17.5
s = "My name is Anne"
answer = True
```

In the first line of code, the variable with the name *i* becomes the integer number 17. In the second line *peterSmith* becomes the real number 17.5. Real numbers are different from integer numbers, as we will see. In the third line the variable *s* becomes the text string *My name is Anne*. We have to enclose text strings in quotation marks. In the last line the variable named *answer* receives the value *True*. Boolean variables have only two values: True and False.

**Summary:**

integer number
: positive or negative whole number

real number
: number with decimals

text string
: character string variables (word or sentences)

boolean value
: boolean variable (True or False)

Note that we always write variable with a lowercase first letter. This is not something that PsychoPy demands. It is just our convention.

## Lists

An important object in PsychoPy is alist. A list is a collection of values, most often of the same type. The easiest way to visualize a list is to imagine a list as a table that contains only one row with one or (almost always) more columns. Each column contains a value, and a program can use the number of each column to access the value in it. The number of such a column is referred to as the index of that array. In PsychoPy, the index begins with the integer 0 and is always written inside squared brackets. A list is also a variable. It is made (or declared) the following way:

```
presentationTime = [3.5, 5.5, 7.5]
presentationText = ["Apple", "Spam", "Ham"]
```

In the first example, *presentationTime* consists of three elements of the type *real number*. To address an individual element in it, you have to write the elements index in square brackets straight after the variable name. For example, to obtain the first value, you would need to use the first of the following commands, for the last value, you would need the second one:

```
firstValue = presentationTime[0]
lastValue = presentationTime[2]
```

You do not need to know all values in your list when you start. You can even start with an empty list:

```
responseTime = []
responseTime.append(3.5)
responseTime.append(5.5)
responseTime.append(7.5)
```

The above example will put the same values in *responseTime* as there are in *presentationTime*.

List are not restricted to one dimension. For example, if you want to save values from a table in an array you could also add an extra dimension to the array:

```
trialResponseTime = []
trialResponseTime.append([2.5, 3.5])
trialResponseTime.append([12.5, 13.5])
trialResponseTime.append([22.5, 23.5])
print(trialResponseTime)
```

Note that what we really did was making an array of arrays. The last line shows us the result.

# Control Statements

In PsychoPy, you can use control statements to alternate the sequence of your program. This is necessary to make your program flexible. In the following paragraphs, three types of control statements will be discussed: *for*, *while* and *if else*.

## The for-statement

For looping over a list, it is easyest to use the *for* statement. It nicely iterates over the elements

```
stimuli = ['Apple', 'Banana', 'Orange', 'Dead Parrot']
for stimulus in stimuli:
    message = visual.TextStim(win, text=stimulus)
    message.draw()
    win.flip()
    core.wait(1.0)
```

Sometimes you need the index that points to the element as well as the element itself. That is possible too:

```
stimuli = ['spam', 'ham', 'more spam', 'even more spam']
for i in range(len(stimuli)):
    message = visual.TextStim(win, text=str(i)+": "+stimuli[i])
    message.draw()
    win.flip()
    core.wait(1.0)
```

You could have used the for statement without a list. Replace *len(stimuli)* with a number and whatever is inside the loop will simply be executed that number of times.

## The while-statement

Sometimes you do not know in advance the number of times that the loop must be executed. Use the *while-statement* then. Make sure you make it possible to exit the loop. Otherwise we speak of an infinite loop. In PsychoPy, while-statement contain two parts:

- The statement *condition* indicates what must be true for the loop to continue.
- The indented part is the *body* the loop. All the code that is indented will be repeated.

As a partial example, here is a loop counting every second for 1 minute:

```
i = 0
c = None
while c==None:   # loop until a key is pressed
    message = visual.TextStim(win, text=str(i)+" press a key")
    message.draw()
    win.flip()
    c = event.waitKeys(maxWait=1.0)
    i = i + 1
```

Note that we need *str()* to convert the integer number *i* to a text string.

## The if-else-statement

Often, it is important to let a program do things differently, depending on the value of a variable. For example, if a variable has a positive value, the program has to react differently than when it is negative.

```
if someVariable > 0:
    # do something
else:
    # do something different
```

In PsychoPy, the *if-else-statement* is used to manage choices whether to do something at a certain point, or to do something else. The choice is always made on basis of an evaluation within the statement. Possible evaluations are as follows:

- x == y (x is equal to y)
- x != y (x is not equal to y)
- x > y (x is greater than y)
- x <= y (x is less than or equal to y)

- x >= y (x is greater than or equal to y)

The following partial example code evaluates whether a student has passed an exam. If students receive a mark higher than or equal to 5.5 he has passed the test:

```
if result >= 5.5:
    message = visual.TextStim(win, text="pass")
else:
    message = visual.TextStim(win, text="fail")
```

It is also possible to use the if-part without a following else-part. Here an example:

```
if temperature < 0):
    message = visual.TextStim(win, text="Frozen\ncold")
```

In this example the \n refers to a next line or a *enter*.

Each if-statement is actually a logical test. If the test is true, then the following line of code is executed. If the test is false, then the statement following the 'else' is executed (if present). After this, the rest of the program continues as normal.

Sometimes, we wish to make a choice out of several conditions. The most general way of doing this is by using the *else if* variant of the if-statement. This works by cascading several comparisons. As soon as one of these tests gives the result *True*, the following code is executed, and no further test is performed. In the following example grades are awarded depending on the result of an exam:

```
if result >= 7.5:
    message = visual.TextStim(win, text="Pass: Grade A")
elif result >= 6.0:
    message = visual.TextStim(win, text="Pass: Grade B")
elif result >= 5.5:
    message = visual.TextStim(win, text="Pass: Grade C")
else:
    message = visual.TextStim(win, text="Fail")
```

# Assignment 2: Grades

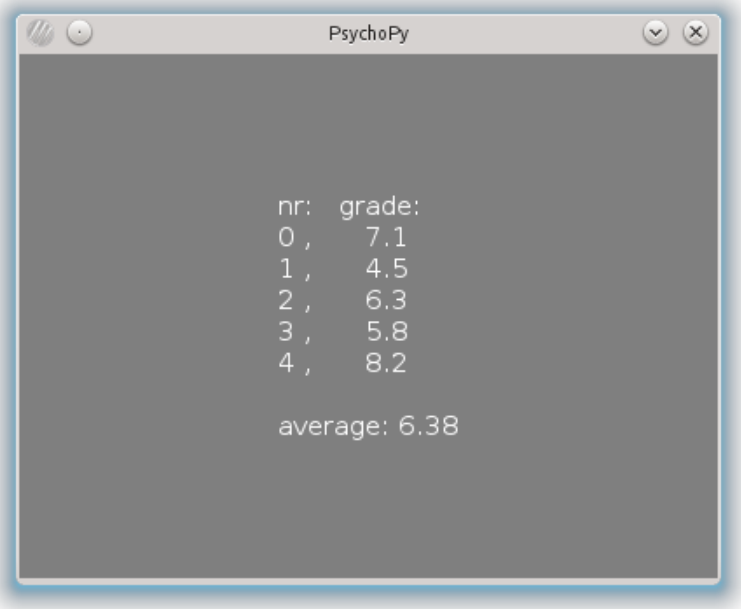
As you have just read, PsychoPy works with variables and control structures. Both things are commonly combined, for example a loop that runs through all elements of an array. Or think of visual feedback, which depends on the value of a certain variable. Now, we will make a short program consisting of an array.

- Run the following experiment.
- Present the average grade on the screen. For calculating the average grade you can use: *average = sum(grades)/len(grades)*. For adding a text to the myText string you can use *myText += "\naverage: " + str(average)*'. Where would this code go?
- Add a third column to the screen, indicating whether the student has passed or failed the test.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
# A PsychoPy experiment is now divided in three sections. In the
# Setup section we read files and open windows. These things are slow
# and therefore should not be done while the experiment is running.
# in the Experiment section we do the actual presenting of stimuli
# en in the Closing section we do what has to be done after the
# experiment if finished.

## Setup Section
from psychopy import core, visual

win = visual.Window([400,300], monitor="testMonitor")

# The following block of code prepares a text stimulus called message
# a numerical variable contains a number
studentNumber = 0
# a string variable contains text, it must be enclosed in quotation marks
myText = "nr:   grade:\n"
# a list variable contains a number of variables
grades = [7.1, 4.5, 6.3, 5.8, 8.2]
# do something for every item in the list
for grade in grades:
    # extend the text with one line, "\n" tells PsychoPy to start en new lin
    myText += str(studentNumber) + " ,      " + str(grade) + "\n"
    # increase the student number with one
    studentNumber += 1

# write myText to a TextStim object called message and draw this to the wind
message = visual.TextStim(win, text=myText)
message.draw()

## Experiment Section
win.flip()
core.wait(5.0)

## Closing Section
win.close()
core.quit()
```

It should give you the following screen:




Continue with the next lesson

# Programming with PsychoPy, 4 Built-in Functions

# Built-in functions

In assignment 2 we used a number of functions. For instance the *sum()* function was used to return the sum of the variables in a list and the *len()* function was used to return the length of the list. The values between parentheses are called the functions arguments. Start from the code given in assignment 2.

We also used a second type of function. This type of function works on an object or module. *message.draw()*for instance draws a message and *win.flip()* flips the window buffers. These objects or module must be made or imported before they are used. If We wanted to use the sine function we would have to import the math module with *import math* and call the sine function using *math.sin()*. PsychoPy has 17 modules which you can use. Look at the reference manual to see what they do.

# Assignment 3: Working with the online documentation

- Look up what the built-in function *TextStim* does exactly. Try several of its optional arguments.
- Find a built-in function that is suitable for showing a image from a jpg-file and write an experiment that uses it. You can start from the code given in assignment 1.

Continue with the next lesson

# Programming with PsychoPy, 5 Structure: Writing your own functions

With all knowledge from previous chapters, we can start programming our experiment. When programming PsychoPy, it is often the case that a certain sequence of operations, is put together, forming a little coherent part. Such a sequence can have a form like

- change the text of a stimulus
- refresh the memory where the stimulus is stored and
- put the new stimulus on the screen.

Such a part is called a function and it forms the core of PsychoPy programming. You can build your whole experiment from these little functions, because they can be accessed anywhere in your PsychoPy code, or within another function, via a function call, using the name of the function. This makes your code easily readable and very flexible. A function consists of:

- the name of the function;
- optionally: the variables that are sent to the function;
- the body of the function which contains the statements;
- optionally: the return statement of the function which send a variable back from the function.

Here is an example of a function:

```
def showText(window, myText):
    message = visual.TextStim(window, text=myText)
    message.draw()
    window.flip()
```

Let us examine the details of this function. The functions header is introduced by the reserved word *def* (for define) and followed by the function name. After that you can see parentheses containing variables you want to pass to the function, the so-called arguments. Each argument is separated from the next by a comma. A function can also be defined without argument, then the parentheses contain nothing. This functon does not contain a return statement.

The body of the function is the indented part and end when the indent stops. All statements between these words will be executed when the function is called. Any variable declared here will be treated as local to the function. The subroutine described above can be called in PsychoPy by the following code.

```
win = visual.Window([400,300], monitor="testMonitor")
showText (win, "Hello World")
```

It seems like a useless function, but when something has to be printed several times, for instance, when computing a list, it can become very useful. Let's take another function, which can calculate square values of integers (math) returning this value. It is important to ensure that your arguments are from the expected type. Otherwise the function will produce strange results or errors. The return statements causes the function to return the desired result.

```
def calculateSquare(number):
    return number * number;
```

# Assignment 4: getInput

- Try to run the next program in PsychoPy.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# There is little experiment if the experimentee cannot give any input.
# Here we changed our assignmet 1 a bit so that it waits for a keys, rather
# than waiting 5 s. Note that we need to import the event library from
# PsychoPy to make this work.
from psychopy import core, visual, event

## Setup Section
win = visual.Window([400,300], monitor="testMonitor")
textString = "Press any key to continue\n"
message = visual.TextStim(win, text=textString)

## Experiment Section
message.draw()
win.flip()
c = event.waitKeys() # read a character
message = visual.TextStim(win, text=textString + c[0])
message.draw()
win.flip()
event.waitKeys()

## Closing Section
win.close()
core.quit()
```

- Make a function that prints some text on the screen (for instance a question) records the one character result, and returns it.

Continue with the next lesson

# Programming with PsychoPy, 6 Structuring: separate files

In the last assignment you have learned to make a subroutine with a passed argument. The text string sent to the function was used to define the question it showed. This was done from the main program. But, subroutines can also be called from within another subroutine.

By this, all individual subroutines created for your experiment can be seen as building blocks forming your experiment. As we progress in programming our experiment and the code and number of functions increase, we need to organize the code. We can for example take all subroutines out of the main program file and make a separate subroutine file for them. Let's call this file *my.py*. By doing this, we get a nicely structured experiment, allowing us to keep overview.

Let's make a file that contains just one function. One that you can call at any moment and that prints the time to your *Output* window and the message that you give to it. Note how the format function can be used to string together the hours, the minutes and the seconds. The format function has many option. You can look them up in the manual. You can for instance do *"{}-{}-{}".format(day, month, year)* to format a date if you have three numbers or use *"{:.2f}".format(3.141592)* to limit the number of decimals of pi to two.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# My functions.

from psychopy import core, visual, event
import math, time

## Function section
def debugLog(text):
    tSinceMidnight = time.time() % 86400
    tSinceWholeHour = tSinceMidnight % 3600
    minutes = tSinceWholeHour / 60
    hours = tSinceMidnight / 3600
    seconds = tSinceMidnight % 60
    print("log {:02d}:{:02d}:{:f}: {}".format(int(hours), int(minutes), seco
```

Save this file as *my.py*. For sure, PsychoPy needs to know where to search for the variables and subroutines. Therefore, a link has to be made. The command import is the instruction we need to use the contents of another file. We use the import statement as follows:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from psychopy import core, visual, event
import my

## Setup Section
win = visual.Window([400,300], fullscr=True, monitor="testMonitor")

## Experiment Section
for i in range(10):
```

```
    core.wait(1.000)
    win.flip()
    my.debugLog(i)

## Closing Section
win.close()
core.quit()
```

We made a my.py for you with a lot more useful functions. Please import and use this one in the assignments from now on.

# Assignment 5: accuracy

- Run the code.
- Note that times are logged to microsecond accuracy. This doesn't make sense. Limit the logging of the time to milliseconds.
- The screens do not appear exactly 1 second apart. How much is the difference? You can calculate it yourself or change the code to do it for you. Change the waiting time. Let it increase in small steps (for instance 2 ms) from 1 s to 2 s. What happens to the difference between desired waiting time and actual waiting time? Can you explain the 'stepping' behaviour?

Continue with the next lesson

# Programming with PsychoPy, 7 Data Import and Export

So far, we have focused on stimuli presentation and responding to these stimuli. We let PsychoPy create the stimuli and put the results on the screen. But we need to record the data in external data files and sometimes we have already existing stimuli lists, we want to use for our experiment. For these functions we need data import and export.

# Export

For exporting data we can use the PsychoPy experiment handler or the underlying Python file handling system. We choose the second since it is simpler to use and offers more freedom.

### Example: saving your data

```
#!/usr/bin/env python
import csv

## Experiment Section, collect data here
data = [1, 0, 1, 1, 1, 0, 0, 1, 1]

## Closing Section
rowcount = 0
datafile = open('data.csv', 'wb')
writer = csv.writer(datafile, delimiter=";")
for row in range(len(data)):
    writer.writerow([row, data[row]])
datafile.close()
```

Saving data in Python works in two steps. What you have to do first is to define an output file where Python can direct output to. You do this by using the *open()* function. Second, you have to make a csv.writer that send your list of data to the output file every time you want to write to it. The term csv stand for comma separated value. It can really do more that just that, but we will see about that in the exercise.

For open(), you need the file name (i.e. how to call the file on the hard disk) as argument and a second argument that describes how to open the file. We use 'wb' for writing en 'rb' for reading. There are two more functions needed, writerow() and close(). With writerow() from csv.writer object you add a line of comma separated values to your output file on the hard disk, with close() from the file object, you close your file.

# Importing

Importing or reading in data is almost as easy as exporting and works in a comparable manner. Imagine you want to present a square of a certain size and color on the screen. The experiment consists of four trials, but this could be a different set of trials for different groups of testees. Let's says that you have a file for each experiment and these files have one line for each trial. The value in the first column is the

size of the square and the value in the second column is the time it should be on the screen. The columns are separated by a semicolon:

```
3.3;"red"
1.1;"green"
4.4;"red"
2.2;"green"
```

The experiment code code look like his:

```
#!/usr/bin/env python
import csv

## Setup section, read experiment variables size and color from file
size = []
color = []
datafile = open('stimuli.csv', 'rb')
reader = csv.reader(datafile, delimiter=';')
for row in reader:
    size.append(float(row[0]))
    color.append(row[1])
datafile.close()

## Experiment Section, use experiment variables here
for trial in range(len(size)):
    print("size = {} m, color = {} s".format(size[trial], color[trial]))
```

First we make the empty lists to store size and color of the stimulus. Then we make a file object and connect it to a *csvreader*. The reader wants to know the precise format of the file, such as the delimiter used. Then we read every line of the file. The first field is appended to the list of sizes. The second row to the list of colors. This example does not show a real experiment section. It just prints the values to the PsychoPy output in the bottom of your window.

You do not need the csv.reader directly in your experiment. *my.py* as given in the previous lesson contains functions that can do this for you.

# Assignment 6: Red-Green

- Save this experiment:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from psychopy import core, visual, event
import csv

## Setup section, read experiment variables from file
win = visual.Window([400,300], monitor="testMonitor", units="cm", fullscr
stimuli = []
datafile = open('redgreen_stimuli.csv', 'rb')
```

```
reader = csv.reader(datafile, delimiter=';')
for row in reader:
    size = float(row[0])  # the first element in the row converted to a f
    color = row[1]        # the second element in the row
    stimulus = visual.Rect(win, width=size, height=size)
    stimulus.setFillColor(color)
    stimuli.append(stimulus)
datafile.close()

## Experiment Section, use experiment variables here
for stimulus in stimuli:
    stimulus.draw()
    win.flip()
    core.wait(1.000)

## Closing Section
win.close()
core.quit()
```

- Run this experiment. You can make the stimuli file with the Psychopy editor or with Excel. Choose *Save as CSV (Comma delimited)* if you use the latter.
- Change the order of the presented stimuli in a random way. Use the shuffleArray function from the PsychoPy misc module for this.
- Change the experiment so that the block is shown until a key is pressed. Store the reaction time and write it to the screen.
- Make sure that the participant has to respond within 1 second. If there is no response within one second write to the screen 'Please respond faster'.
- Extend your program that it writes the results to a file that contains original trial number, box size, box color and reaction time for each trial.

# Warning for use with Microsoft Excel

Microsoft Excel uses the *List separator* from the *Region and Language* settings as delimiter when saving a csv-file, even though in the *Save File* dialog it says *CSV (Comma delimited) (*.csv)*. This is a long standing bug in Microsoft Excel that especially affects European users, since in European version of Microsoft Windows the standard list separator is set to semi-colon (;). Since the interpretation of the comma (and the period) as decimal marker (and therefore not as list separator) is considered standard in the ISO 80 000 *International System* (pdf 5.3.4) we assume in this tutorial that the list separator is set to semi-colon.

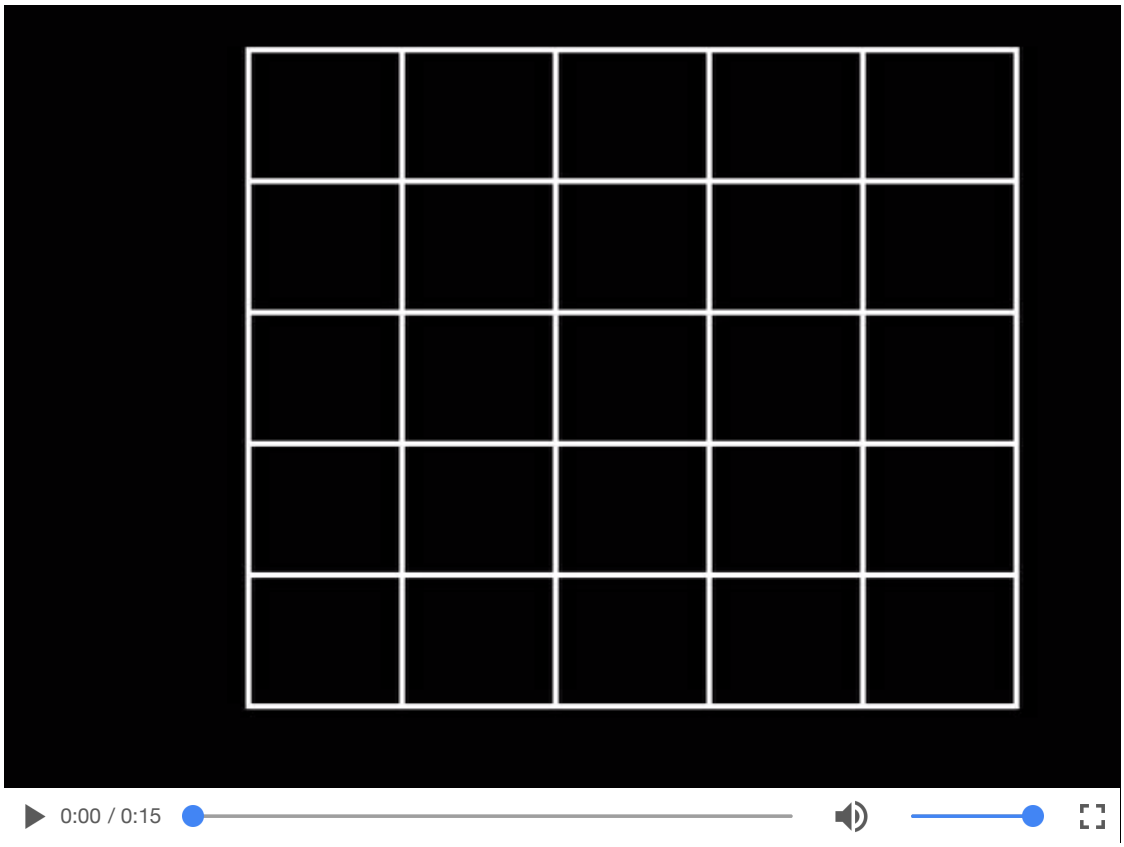
Unlike most other programs, Microsoft Excel does not support the UTF-8 character set in csv-files. Keep in mind that you cannot use non-ascii characters (χαρακτήρες) if you make your experiment file in Excel.

Continue with the next lesson

# Programming with PsychoPy, 8 Stimuli

# Visual

Computer screens are updated according to their refresh rate. What happens then is that the whole screen is redrawn line by line all the time. For example, with a refresh rate of 60 Hz, the screen is redrawn 60 times per second (1000 ms) and the duration it takes to redraw it line by line is 16.67 ms (1000/60). The following video shows what it looks like:




When attempting to redraw the screen while it is currently already being updated (the lines are drawn) the result might lead to artifacts, since the update occurs immediately, leading to parts of both, the new and and the old screen content, being visible. The following image shows what happens if an image is updated twice while it is drawn to the screen:


What is even worse is that you will never know in which phase of the redrawing the new redraw was started. Thus, you cannot be sure when exactly the new content is fully visible on screen. The first step towards getting around this problem is to synchronize the actual redraw to the vertical retrace of the screen. This means that a change in content will never happen immediately, but always only when the retrace is at the top left position. When synchronizing to the vertical retrace, the graphic card is told to update the screen the next time it starts redrawing the first line. While this will solve the problem of artifacts, you will still face the problem of not knowing when exactly something was visible on the screen, since the graphic card handles this synchronization itself in the background.

PsychoPy solves this problem with the .flip() function. It allows waiting for the vertical retrace to actually happen before proceeding with the code that tells the graphic card to update the screen (this is also known as blocking on the vertical retrace). This means that whenever something should be presented on screen, no matter in which line the redraw is at this point in time, the graphic card will wait for the redraw to be in the first line and then present the stimulus. Since the code is blocking, the time presentation reports the stimulus to be presented on screen will always be the time when the redraw is starting at the first line.

It is important to switch off power saving schemes of your graphic card's driver, and use a special graphical card provided by your technical support!

# Audio

To play back audio you have to create an sound object from the sound module:

```
#!/usr/bin/env python
from psychopy import core, sound
s = sound.Sound(value='C', secs=0.5)

s.play()
core.wait(4.0)

core.quit()
```

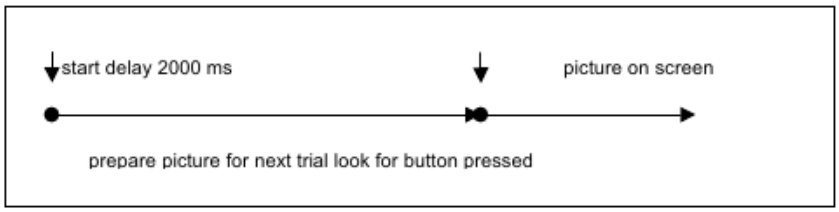
The play() function of auditory stimuli will return immediately. Therefore the experiment has to do a bit of waiting after calling the play function. You can also ask for input or do other things after calling the play function. Just make sure the experiment does not end there, because then you will not hear the sound.

Since the audio stream has to be sent to the hardware, there will still be a delay before the audio can be heard. Unfortunately, the latency of the sound onset is not 0 ms. However, it is assumed to be relatively stable over time. Tests results around 5 ms. And again use a special sound card provided by your technical support!

The sound object has many more options. It can play a tone, a complete stereo waveform or audio files.

# Visualizing your experiment: A time line

In order to keep track of the desired timing of your experiment, it might be useful to visualize it by drawing a time line. A time line gives a nice overview how your experiment is build up and when and what has to happen. It is advised to draw a timeline of the experiment before you start to program. In this manner you can decompose your program into event pieces and start programming these step by step (or bit by bit ;-).




In this example you start with creating a delay. Then you prepare your picture. When a button is pressed, the picture occurs on your screen. Programming done!

# Assignment: time line

- Study the following experiment and draw its timeline. Do not try to run this experiment. The world as we know it will cease to exist.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from psychopy import core, visual, event
import csv, random, time

## Setup Section
win = visual.Window([800,600], fullscr=False, monitor="testMonitor", units='

# turn the text strings into stimuli
textStimuli = []
for iTrial in range(10):
    # append this stimulus to the list of prepared stimuli
    text = "number: {}".format(iTrial)
```

```
    textStimuli.append(visual.TextStim(win, text=text))

# make a question stimulus
questionStimulus = visual.TextStim(win, text="Was this number larger than 4?

# fixation cross
fixation = visual.ShapeStim(win,
    vertices=((0, -0.5), (0, 0.5), (0,0), (-0.5,0), (0.5, 0)),
    lineWidth=5,
    closeShape=False,
    lineColor='white'
)

# open data output file
datafile = open("/tmp/datafile.csv", "wb")
# connect it with a csv writer
writer = csv.writer(datafile, delimiter=";")
# create output file header
writer.writerow([
    "number",
    "response",
    "responseTime",
    ])


## Experiment Section
n = len(textStimuli)
for i in random.sample(range(n), n):
    # present fixation
    fixation.draw()
    win.flip()
    core.wait(0.980) # note how the real time will be very close to a multip

    # present stimulus text and wait a maximum of 2 second for a response
    textStimuli[i].draw()
    win.flip()
    textTime = time.time()
    wait = random.uniform(0.100, 0.500)
    core.wait(wait)

    # ask wheter the number was larger than 4
    questionStimulus.draw()
    win.flip()
    key = event.waitKeys(2.000, ['y','n', 'escape'])
    responseTime = time.time()

    # write result to data file
    if key==None:
        key=[]
        key.append("no key")
```

```
    writer.writerow([
        i,
        key[0],
        "{:.3f}".format(responseTime - textTime),
        ])

    if key[0]=='escape':
        break
datafile.close()

## Closing Section
win.close()
core.quit()
```

Continue with the next lesson

# Programming with PsychoPy, 9 Buttonbox

# The problem

Please take a stopwatch (a real one). Start it, and stop it at 1.00 s. Start again and stop at 2.00 s. Try it a number of times. Estimate how far your deviation from the integer second is. The author of this tutorial can do it with a standard deviation of about 0.02 s, but many people are much better than that.

The fact that human reaction times (or reaction precisions, or whatever you wish to call them) are in the region of 20 ms means that in order for us to measure them we will need a computer that is an order of magnitude faster. If I want to measure the difference between a slow person who reacts in 20 ms and a quick person who reacts in 10 ms it is useless to have a computer that cannot see the difference between the two.

Unfortunately computers are not as fast as they seem. A typical computer keyboard will take about 30 ms to send a keypress to the computer. This means that if you measure times that are in the order of 1 second and have a σ of 0.1 s you will have a systematical error in your μ measurement of 3%, an additional random error in each measurement of 3% and a granularity (coarseness) that completely ruins the measurement of σ. Reaction times are often much faster than this, making the problem even larger.

# The solution

Fortunately the Faculty's Technical Support Group has a solution: the buttonbox.

The buttonbox has a reaction time of 1 ms, reducing your problem with a factor of 30 (and the square of that for σ). It also has a few LED's that are just as fast, solving all your problems with display timing. It

can play sounds and react to them, it can interface with other devices and it can write your PhD thesis for you. (Only one of the claims in the previous sentence is not true)

The buttonbox talks to your computer over a USB-cable of the same type that you use for connecting your printer or mobile phone. If you are using an old buttonbox that is connected with a serial cable, please stop reading and go to the next chapter, since this experiment will not works for you.

# The software

The buttonbox connects to the computer using the BITSI protocol. Older versions but there is no need for you to use this protocol directly. Use the functions from the RuSocSci package:

```
#!/usr/bin/env python
## Setup Section
from psychopy import core, visual, event
from rusocsci import buttonbox
win = visual.Window([400,300], monitor="testMonitor")
bb = buttonbox.Buttonbox()

## Experiment Section
bb.waitButtons(maxWait = 10.0, buttonList=['A'])

## Cleanup Section
core.quit()
```

Note that this is almost identical to the same experiment using a keyboard. All you have to do is replace the waitKeys() function from the event module with the waitButtons() function from the buttonbox module

It is even better than that. If there is no buttonbox connected the waitButtons() function will simply use the input from the keyboard. This way you can develop and test your experiment at home and run it in the lab without changing as much as one line of code.

The buttonbox sends a 'A' when the first key is pressed and a 'a' when the first key is released. Do not forget about the difference en make use of it. The time between the two is a measure for the way people are typing.

# Assignment: Simon

- Save the following experiment:

```
#!/usr/bin/env python
# -*- coding: utf8 -*-
from psychopy import visual, event, data, logging
import random, datetime, time

## Setup section
# experiment specific
#win = visual.Window([400,400], units='height', winType='pyglet', fullscr
```

```
win = visual.Window([400,400], units='height', winType='pyglet')
width = float(win.size[0])/win.size[1] # actual width in units of height
logging.console.setLevel(logging.WARNING) # default, only Errors and Warn
# run  specific
fileName = "data/simon"+datetime.datetime.now().strftime("%Y-%m-%dT%H_%M_
dataFile = open(fileName, 'w') # note that MS Excel has only ASCII .csv,
dataFile.write("#{}, {}, {}\n".format("position", "time", "response"))


instruction = visual.TextStim(win,
        text='Press ← if you see a rectangle left of the fixation cross.
        'Press → if you see one on the right. Press Escape to stop the '\
        'experiment, or continue to the end. It takes about a minute.'.de
    )
# fixation cross
fixation = visual.ShapeStim(win,
        vertices=((0, -3), (0, 3), (0,0), (-3,0), (3, 0)),
        lineWidth=10,
        size=.01,
        closeShape=False,
        lineColor='blue'
    )

# stimulus rectangle
stimulus = visual.Rect(win,
        width     = 0.1,
        height    = 0.1,
        fillColor = 'red',
        lineColor = 'red'
    )

## Experiment section
for i in range(50):
    # fixation
    fixation.draw()
    win.flip()
    presses = event.waitKeys(1.0)
    if presses and presses[0] == 'escape':
        break

    # stimulus
    x = -(width-0.1)/2 +(width-0.1)*random.random()
    stimulus.setPos((x, 0))
    stimulus.draw()
    win.flip()
    timeBefore = time.time()
    presses = event.waitKeys(3) # wait a maximum of 3 seconds for keyboar
    timeAfter = time.time()

    #handle keypress
    if not presses:
```

```
            # no keypress
            print "none"
            p = 0
        elif presses[0]=="left":
            p = -1
        elif presses[0]=="right":
            p = 1
        elif presses[0]=="escape":
            break
        else:
            # some other keypress
            print "other"
            p = 0
        dataFile.write("{}, {}, {}\n".format(x, timeAfter-timeBefore, p))
        #print x, timeAfter-timeBefore, p

    ## Cleanup section
    win.close()
    core.quit()
```

- Change this experiment to do the following with the buttonbox instead of with the keyboard. Show a square somewhere on the screen, vertically centered. Let people press the A button if it is at the left side, or the E button if it is on the right side. Record in a file whether the result is correct or not. Also record the reaction time. Do this for a hundred trials. Plot the results. In my results people are faster if the stimulus is more pronounced. Do you get the same result? How significant is it? Did you need the speed of the buttonbox?

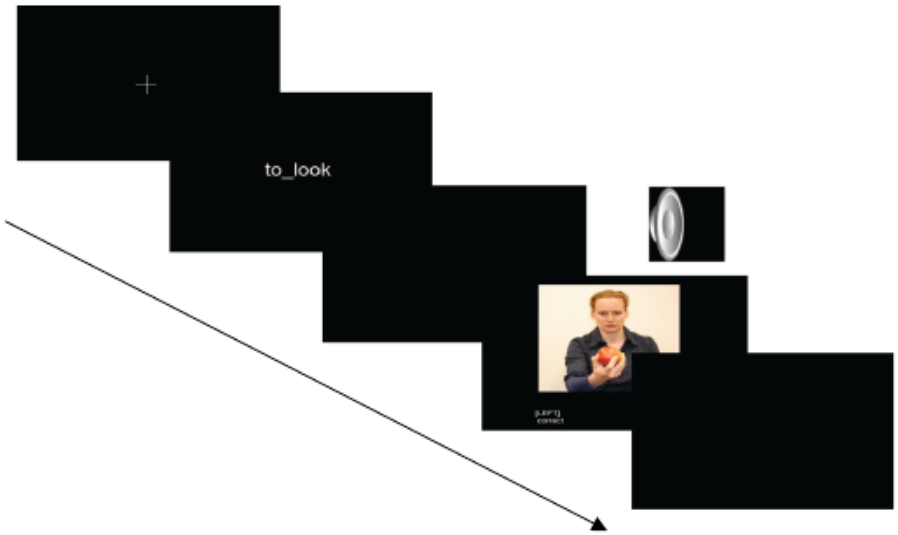
Continue with the next lesson

# Programming with PsychoPy, Final assignment

It is time for the grande finale: the final assignment. We prepared for you a template with example functions, code snippets and useful comments. It contains the examples from this course and much more. You can use these functions from the template directly or you can copy them into your own experiment and change them. It often makes sense to change the name of the function if you change it's behaviour. Look in lesson 6 to see how to use the functions directly.

# The experiment

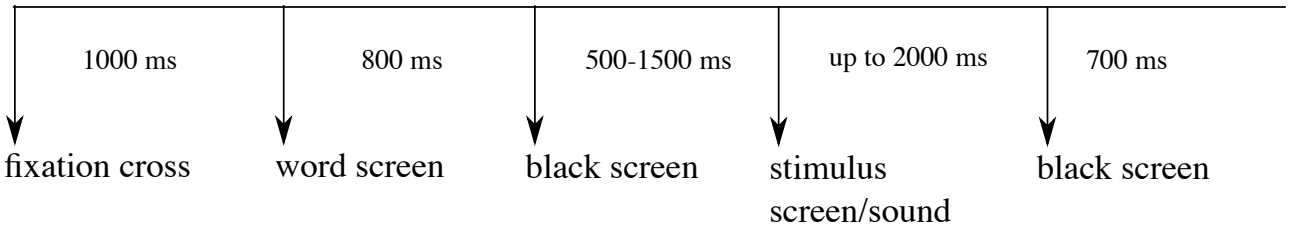
As final assignment you will write an experiment




Does the prime match the end goal of the action, irrespective of the object involved? You will need these files. The archives contains the image files, the sound files, the template with a well structured dummy experiment and the file my.py containing a number of useful functions.

# Specifications:

1. Present fixation, Wait (1000 ms)
2. Present prime for 800 ms ("to taste", "to smell", "to listen", "to look")
3. Black screen Wait (jitter 500 - 1500 ms),
4. Present action picture for (2000 ms) with response
5. Play correct or false sound
6. Black screen for 700 ms




- Use the template to make a well structured experiment.
- Use the timeline to build up the experiment bit by bit, start with presenting a fixation cross.
- Create a 1000 ms delay and use the frequency of the monitor to create the correct time slot
- Find the already made tab stimuli file (*stimuli.csv*) the trials are defined in there.
- Read in a semi column delimited file with the function *my.getStimulusInputFile("stimuli.csv")* use te file specified.
- Create all the word stimuli screens
- Create all the picture stimuli screens
- Use the bitmap stimuli screen and add text "correct" and "incorrect".
- Create the incorrect and correct sound
- present the word screen for 800 ms
- present the black screen with a jitter of 500 - 1500 ms. hint; random.uniform()
- present the image stimuli
- Give the participant 2000 ms to react.
- Only except the responses "Y" and "N".
- play a sound at the time the button was pressed
- Present a black screen for 200 ms
- Check with the input files last column (y or y) if the answers given to the picture are correct and give feedback with a correct or incorrect sound
- Log which button is pressed and the time the button is pressed.
- Log all the times when pictures are presented
- Write all the logged data to the hard disk use writer.writerow()
- Check if the output file is correct