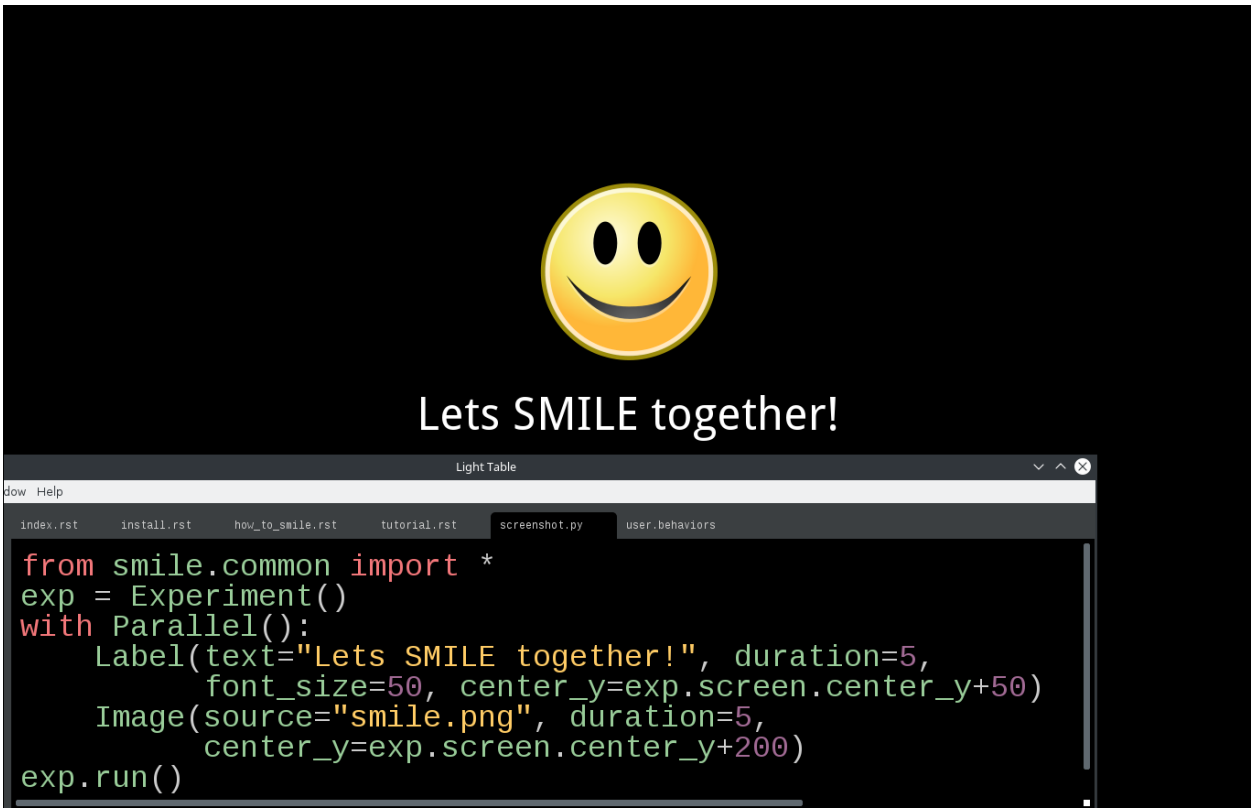
SMILE Documentation

Release 0.1.0

Per B. Sederberg

February 27, 2017

Contents



SMILE is the State Machine Interface Library for Experiments. The goal when developing SMILE was to create an east-to-use experiment building library that allows for millisecond-accurate timing. SMILE was written so that the end user doesn't have to worry about the intricacies of timing, event handling, or logging data.

Inspired by the concept of a state machine, SMILE works by storing the current status of releveant input events, then initiating an action depending on a predetermined set of rules. With the support of the versatile **Python** programming language and **Kivy**, a module create for video game development, SMILE is powerful and flexible while still being simple to use.

CHAPTER 1

What does a SMILE experiment look like?

Below is hello.py, an example of what the simplest SMILE experiment looks like:

```
from smile.common import *

exp = Experiment()

Label(text="Hello, World!", duration=5)

exp.run()
```

In order to run this experiment from a computer that has SMILE installed, you would use your favorite OS's command prompt and run the following line:

```
>> python hello.py -s SubjectID
```

This program creates a full-screen window with a black background and the words **Hello, World!** in white text in the center–just like that, we are SMILEing!

Now let us go through our SMILE experiment line by line and see what each of them does.

First is the line *exp = Experiment()*. This line is the initialization line for SMILE. This tells SMILE that it should prepare to see states being declared.

Second is the line *Label(text="Hello, World!", duration=5)*. **Label** is a SMILE visual state that displays text onto the screen. Certain SMILE states take a *duration*, and we are setting this state's duration to **5**. This means the state will remain active on the screen for 5 seconds.

Third is the line *exp.run()*. This line signals to SMILE that you have finished building your experiment and that it is ready to run. SMILE will then run your experiment from start to finish and exit the experiment window when it has finished.

CHAPTER 2

Whats Next?

To help you get ready to SMILE, the first section of this documentation is the SMILE installation and the installation of its dependencies. After that is a section that delves deeper into SMILE and how to write more complicated experiments.

Installation of SMILE!

Getting ready to SMILE? Then you are in the right place. This guide will tell you how to install SMILE and the package that SMILE is dependent upon, Kivy! Scroll down to the appropriate operating system and follow the directions provided to install Kivy, SMILE, and any extra needed packages.

Installing SMILE on Windows

Before installing anything, make sure that you have python installed and that you can run python through your command prompt.

Also, it is important to have pip installed to your python. Without pip you will not be able to run the commands needed to install SMILE. To install *pip*, click the link below and follow the instructions.

-Get pip

The next thing you need to install after *pip* is *kivy*. *Kivy* is the display backend for SMILE. Note that you do not need to know anything about how to use kivy to figure out how to use SMILE.

To install kivy on your windows machine, run the following line in your command prompt.

```
> python -m pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew
> python -m pip install kivy.deps.gstreamer --extra-index-url https://kivy.org/downloads/packages/sir
```

Then run this line in your command prompt.

```
> python -m pip install kivy
```

Note: If you run into any trouble installing kivy onto your windows machine, please check the kivy website for more detailed instructions.

After running the last command, it is now time to download SMILE. Download SMILE from the github link provided and then extract it.

-SMILE Download

Now, in your command prompt, navigate to the newly extracted smile download folder that contains setup.py and run the following line.

```
> python -m pip install .
```

The final thing you need to install to gain access to all of SMILE's functionality is PYO. PYO is used to play and record sound with SMILE. Download and install the windows version of PYO from their website. The link is provided below.

-PYO Download

Note: When PYO asks for a directory to install to, choose *C:Python27*. If that folder doesn't already exist, create it and then attempt to install PYO into that folder

---

With that, you are finished installing SMILE. Congrats! Head over to *The SMILE Tutorial* to start SMILING. This will cover a more advance look into how SMILE works.

Sync Pulsing on Windows

To use sync pulsing on windows via the parallel port, you must install **Inpout32**, or include *inpout32.dll* in the same folder as your experiment.

Windows Troubleshooting

If you are trying to replace an older version of SMILE, or if you just need to upgrade your current version, you must run the following command while the Anaconda command prompt is in the SMILE download folder.

```
> pip install . --upgrade
```

If you would like to use any of the audio options of SMILE, pyo is required. If you find that you can't install pyo, it is because you are not using the 32 bit version of Python. You can install SMILE on 64 versions of Python, but you will lose the ability to play sound files. Your ability to play sound while presenting a **video** file, however, will not be inhibited.

If you are trying to install SMILE to an Anaconda distribution of python, you must use 64 bit. We have found that the 32 bit version of GStreamer that Anaconda provides will not work well with Kivy, and will error out. Please use the 64 bit version of Anaconda if you choose to install SMILE to it.

If you are installing SMILE to a Python separate from Anaconda, but still have Anaconda installed on that machine, you may encounter a weird pathing error. We are still looking into what causes it, and it doesn't happen to everyone, but we would still like you to be aware that you may run into some problems.

Installing SMILE on Mac

The first step is to download and install Kivy. The following link will take you to the Mac-Kivy install guide.

-Mac-Kivy Install Guide

After you install Kivy, you must download and install SMILE. The following is a link to the SMILE download page, where you will download the zip, and extract it to an easy to find place.

-SMILE Download

Now, all you have to do is open the terminal and navigate to the newly extracted smile download folder. This folder should contain setup.py. Run the following line to install SMILE to your special Kivy distribution of python.

```
$ kivy -m pip install .
```

Easy. SMILE should have installed without any issue.

The final thing you need to install to gain access to all of SMILE's functionality is PYO. Download and install the Mac version of PYO from their website. The link is provided below.

-PYO Download

With that, you are finished installing SMILE. Congrats! Head over to *The SMILE Tutorial* to start SMILING. This will cover a more advance look into how SMILE works.

Mac Troubleshooting

If you are trying to replace an older version of SMILE, or if you just need to upgrade your current version, you must run the following command while the Anaconda command prompt is in the SMILE download folder.

```
$ kivy -m pip install . --upgrade
```

If you require any additional packages to run your experiment, you must use **kivy** to install them. Like above, you use the *kivy -m pip install* line to install any additional packages to the python that is linked to kivy.

Installing SMILE with Linux

SMILE requires Kivy to run properly, but if you would like to use the smile.sound functionality, you need to download and install PYO as well. Run the following in your command line to install both Kivy and PYO at the same time.

```
$ sudo aptitude install python-pyo python-kivy
```

If you are running something besides a Debian based linux system, the above line will look different. It depends on your prefered package manager.

Then, download SMILE from github and extract it to a place you can find later. The download link is the following:

-SMILE Download

Next, navigate to the newly extracted smile folder that contains setup.py, and run the following line in your terminal window.

```
$ python -m pip install .
```

This will add SMILE to your python distribution.

With that, you are finished installing SMILE. Congrats! Head over to *The SMILE Tutorial* to start SMILING. This will cover a more advance look into how SMILE works.

Sync pulsing on Linux

To use sync pulsing on linux over a parallel port, you must install PyParallel. Install it via *pip* or your favorite package manager.

Linux Troubleshooting

To be added when problems are found.

SMILE Tutorial Basics!

Hello SMILErs! This tutorial is for people just starting out in the world of SMILE. Further in this documentation, there is a more advanced tutorial. If you are brand new to SMILE and want to learn the basics line by line, you are in the right place.

Running a SMILE Experiment

After installing SMILE, there is only one thing needed to run a SMILE experiment, and that is a fully coded experiment file. SMILE uses python to run its experiments, so to run SMILE you must run the *.py* file with python.

If you followed our instructions for installing SMILE, Linux and Windows users would use the following line in a command prompt to run their SMILE experiments:

```
>> python filename.py -s SubjectID
```

If you are an OSX user, you just replace the **python** in the previous line with **kivy**:

```
$ kivy filename.py -s SubjectID
```

Notice the *-s* in the commands above. This is a command line argument for SMILE. SMILE has 3 command line arguments.

- *-s* : Subject ID, whatever identifier you would like to use for a particular run of the experiment. The next argument passed *-s* will be the subject ID for the purposes of where to save data on your system.
- *-f* : Fullscreen, if *-f* is present in the command line, SMILE will run in windowed mode.
- *-c* : CSV, if -c is present, SMILE will save out all of its *.slog* data files as *.csv* data files as well. **Not Recommended**

Before you learn how to code SMILE experiments, it is important to understand a few things about how SMILE works. The next section goes over how SMILE first *builds* then *runs* experiments.

Build Time V.S. Run Time

The difference between **Build Time** and **Run Time** is the most important concept to understand when learning to use SMILE. There are 2 lines of code that designate the start of **BT** and then the start of **RT**. Those lines are *exp = Experiment()* and *exp.run()* respectively.

exp = Experiment() initializes the instance of an `Experiment`. All calls to a state must take place after this line! Once this line is run, **BT** starts. **BT**, or Experimental Build Time, is the section of the code that sets up how the experiment will run.

During Experimental Build Time, all calls to the different states of SMILE define how your experiment will run to SMILE. SMILE sees each of those states and uses them to setup the rules of how your state machine will flow from one state to another. When SMILE see the *with Parallel():* state, it will know that all of the states that are defined within should run at the same time. When SMILE sees one **Label** following another **Label**, SMILE will know that the second **Label** should not show up on the screen until the first one has finished running.

During Experimental Run Time, all of the timing and intricacies of SMILE's backend are run. Once *exp.run()* is called, SMILE will start whatever the first state you defined in the experiment is and continue with the rest of your experiment afterwards.

Note: During **RT**, SMILE will not run any non-SMILE code. SMILE will only run the prebuilt state-machine. If you need to run any kind of python during your experiment, use the `Func` state.

Another thing to look out for when programming the experiment how variables are set and used in **BT**. A local variable in between *exp = Experiment()* and *exp.run()* cannot be set and expected to actually set during **RT**. In order to *set* and *get* local variables during **RT**, *set* and *get* must be used through the local `Experiment` variable. To set this kind of variable, *exp.variable_name* must be added to the beginning of the variable name. Doing this creates a `Set` state in SMILE that will run during **RT**. An example is as follows.

```
exp.variableName = lbl.appear_time['time']
```

For more information about setting in **RT** see the *Setting a Variable in RT* section of **Advanced SMILEing**

What are References?

Since SMILE will build the experiment before it runs it, we needed to think of a way to reference variables before the variables were created. That is why we developed the `Ref`. The **Ref**, very basically, is a delayed function call. Using **Ref**s, SMILE is able to hold onto a reference to data that hasn't been created yet in your experiment. **Ref**s are powerful in that they are recursive. That means that if you apply a basic operation to a **Ref** (i.e. +, -, , or /) it will create a new **Ref* that contains both sides of the operation, and the operation function itself.

```
from smile.ref import Ref
a = Ref.object(5)
b = Ref.object(6)
c = a + b
print c.eval()
```

In the above example, *a* and *b* are refs that are created to contain only an object. **Ref.object()** will return a **Ref** that will, when being evaluated later, check to see what the value of the object is at that moment and return that value. The above example creates 2 *integer* references. The third line *c = a + b* is an example of creating a recursive reference. When *c* tries to evaluate itself, it will attempt to evaluate *a* and *b*, then add them together and return the result. The above example will print out the number *11* when it finishes running.

Note: You should not have to ever call *.eval()* for a reference. This was just an example to demonstrate how we use references in SMILE's backend. SMILE calls *.eval()* automatically.

References can also create a Reference object that contains a conditional expression to be evaluated later. These are important when building SMILE `If` states. Say for instance the experimenter would like to present "CONGRATS" on screen if the participant responded in less than three seconds, and "FAILURE" if the participant took longer than three seconds to respond. The experimenter would need to rely on a Referenced conditional statement, where *Ref.cond(cond, true_val, false_val)* can return any kind of object if true or false. Say you want to display "jubba" if a participant presses "J" and "bubba" if the participant presses "K". SMILE allows you to use *cond* to do this in 1 line rather than use an **If** state. For the above example, please see the `Ref` docstring.

A `Done` state is a unique state that will wait until the value of a reference is made available. A reference is made available the first time something calls *.eval()*

Warning: This state is not for regular use. It should only be used when encountering the NotAvailableError. Misuse of the *Done* state, the experiment will have hang-ups in the framerate or running of the experiment.

For more information about `Ref` and `Func` please see *Preforming Functions and Operations in RT*

The next section of the doc will go over some simple SMILE tutorials and introduce you to the states you can add to a SMILE experiment.

Looping over Lists! In Style

The following example will walkthrough the basics of looping over a list. This walkthrough is divided into sections of code and explanation with the combined code sections given at the end of the example.

Before we start, create a new directory called *exp* and create a file called *randWord1.py*. In this file, the stimulus can be defined.

```
words=['plank', 'dear', 'adopter',
       'initial', 'pull', 'complicated',
       'ascertain', 'biggest']

random.shuffle(words)
```

The file has created a list of words that will be randomly sorted when compiled. From here, `Loop` is used to loop over the list of words. Before that, however, the preliminary variables must be established. After, *exp = Experiment()* begins the building process.

```
#Needed Preliminary Parameters of the Experiment
interStimulusDuration=1
stimulusDuration=2

#We are ready to start building the Experiment!
exp = Experiment()
```

The default state in which `Experiment` runs in is the `Serial` state. `Serial` just means that every other state defined inside of it runs in order, first in first out. So every state defined after *exp = Experiment()* will be executed fifo style. Next, a staple of every SMILE experiment, the `Loop` state is needed to be defined.

```
with Loop(words) as trial:
    Label(text=trial.current, duration=stimulusDuration)
    Wait(interStimulusDuration)

exp.run()
```

The list of words that are to be looped act as a parameter in *Loop*. This tells SMILE to loop over *words*. *Loop* also creates a reference variable. In this instance, the reference variable is called *trial*. *trial* acts as a link between the experiment's building and running states. Until *exp.run()* is called, *trial* will not have a value. The next line defines a `Label` state that displays text for a duration. By default, it displays in the middle of the experiment window. Notice *trial.current*: In order to access the numbers from the random list, *trial.current* is used instead of *words[x]*. *trial.current* is a way to tell SMILE to access the *current* member of the *words* list while looping.

> **Warning:** Do not try to access or test the value of trial.current. trial.current is a reference variable, so you will not be able to test its value outside of a SMILE state.

Finished rand_word_1.py

```
1 from smile.common import *
2 import random
3 words = ['plank', 'dear', 'adopter',
4          'initial', 'pull', 'complicated',
5          'ascertain', 'biggest']
```

```
6
7  random.shuffle(words)
8
9  #Needed Parameters of the Experiment
10 interStimulusDuration=1
11 stimulusDuration=2
12
13 #We are ready to start building the Experiment!
14 exp = Experiment()
15
16 with Loop(words) as trial:
17     Label(text=trial.current, duration=stimulusDuration)
18     Wait(interStimulusDuration)
19
20 exp.run()
```

And Now, With User Input!

The final step in the SMILE tutorial is to add user input and logging. In this experiment example, a participant is presented with words, one a time. The participant is told to press the J key if the presented word contains an even number of letters, or press K the number of letters is odd. The participant has 4 seconds to make a response.

This tutorial will also teach how to compare **trial.current** comparisons. First, create a directory called *WordRemember* and create a file within the directory called *randWord2.py*. Now, the word list must migrate to our new file from the previous file in the tutorial. This file will be slightly edited to make sure that the experiment will be able to tell which key is the correct key for each trial.

```
key_dic = ['J', 'K']
words = ['plank', 'dear', 'thopter',
         'initial', 'pull', 'complicated',
         'ascertain', 'biggest']

temp = []

for i in range(len(words)):
    condition = len(words[i])%2
    temp.append({'stimulus':words[i], 'condition':key_dic[condition]})

words = temp
random.shuffle(words)
```

The list of words is now a list of dictionaries, in which *words[x]['stimulus']* will provide the word and *words[x]['condition']* will provide whether the word has an even or an odd length. Similar to the last example, the next step is to initialize all of our experiment parameters. **key_list** is which keys the participant will be pressing later.

```
#Needed Parameters of the Experiment
interStimulusDuration=1
maxResponseTime=4

#We are ready to start building the Experiment!
exp = Experiment()
```

Notice the line change from *stimulusDuration=2* to *maxResponseTime=4*. Now, the basic loop can be set up. The first thing needed to be added to this loop is the *UntilDone():* state. A `UntilDone` state will run its children in `Serial` until the parent state has finished.

The following is an example before the loop was edited:

```
###########EXAMPLE, NOT PART OF EXPERIMENT#########
Label(text='Im on the screen for at most 5 seconds')
with UntilDone():
    Label(text='Im On the screen for 3 seconds!', duration=3)
    Wait(2)
```

As you can see, The first `Label` is on the screen for 5 seconds because the `UntilDone` state does not end until the second `Label` runs for 3 seconds and the `Wait` runs for 2 seconds.

Now to implement this state into the loop:

```
with Loop(words) as trial:

    Label(text=trial.current['stimulus'])
    with UntilDone():
        kp = KeyPress(keys=key_dic)

    Wait(interStimulusDuration)

exp.run()
```

This displays the number of the current trial until a key is pressed, after which the loop waits for the inter-stimulus duration that was predefined earlier. The next step entails editing *kp = KeyPress(keys=keys)* to include the response time duration. Also needed is the ability to add a check to see if the participant answered correctly. This will require the use of *trial.current['condition']*, which is a listgen value set earlier.

```
with Loop(words) as trial:
    Label(text=trial.current['stimulus'])
    with UntilDone():
        kp = KeyPress(keys=key_dic, duration=maxResponseTime,
                      correct_resp=trial.current['condition'])
    Wait(interStimulusDuration)
exp.run()
```

The last thing needed to complete the experiment is to add, at the end of the *Loop()*, the `Log`. Wherever a `Log` state is placed in the experiment, it will save out a **.slog** file to a folder called *data* in the experiment directory under a predetermined name put in the *name* field.

```
Log(name='Loop',
    correct=kp.correct,
    time_to_respond=kp.rt)
```

With this line, each iteration of the loop in the experiment will save a line into *Loop.slog* containing all of the values defined in the *Log()* call.

The loop will look as follows:

```
with Loop(words) as trial:
    Label(text=trial.current['stimulus'])
    with UntilDone():
        kp = KeyPress(keys=key_dic, duration=maxResponseTime,
                      correct_resp=trial.current['condition'])

    Wait(interStimulusDuration)
    Log(name='Loop',
        correct=kp.correct,
        time_to_respond=kp.rt)
```

Finished rand_word_2.py

```
from smile.common import *
import random

words = ['plank', 'dear', 'thopter',
         'initial', 'pull', 'complicated',
         'ascertain', 'biggest']

temp = []

for i in range(len(words)):
    condition = len(words[i])%2
    temp.append({'stimulus':words[i], 'condition':key_dic[condition]})

words = temp
random.shuffle(words)

#Needed Parameters of the Experiment
interStimulusDuration=1
maxResponseTime = 4
key_dic = ['J', 'K']

#We are ready to start building the Experiment!
exp = Experiment()

with Loop(words) as trial:
    Label(text=trial.current['stimulus'])
    with UntilDone():
        kp = KeyPress(keys=key_dic, duration=maxResponseTime,
                      correct_resp=trial.current['condition'])
    Wait(interStimulusDuration)
    Log(name='Loop',
        correct=kp.correct,
        time_to_respond=kp.rt)

exp.run()
```

Now you are ready to get SMILEing! The next section of this documentation goes over every state that SMILE has to offer!

SMILE States

The States of SMILE

Below is the list of most of the SMILE states you will ever need when running an experiment. Each state has a rudimentary tutorial on how to use them. If you need more information about what a specific state does, then checkout each state's docstring.

The Flow States of SMILE

One of the basic types of SMILE states are the **Flow** states. **Flow** states are states that control the flow of the experiment.

Serial State

A `Serial` state is a state that has children and runs its children one after another. All states defined between the lines *exp = Experiment()* and *exp.run()* in an experiment will exist as children of a *Serial* state. Once one state ends, the `Serial` state will call the next state's start method. Like any flow state, the use of the *with* pythonic keyword is required and makes the source code look clean and readable. Below is an example of the *Serial* state.

Note: For many examples, Action State *Label* will be used. This state merely displays text on the screen, similar to the "print" python command. For more details on *Label*, click `Label`.

The following two experiments are equivalent.

```
from smile.common import *

exp = Experiment()

Label(text="First state", duration=2)
Label(text="Second state", duration=2)
Label(text="Third state", duration=2)

exp.run()
```

```
from smile.common import *

exp = Experiment()

with Serial():
    Label(text="First state", duration=2)
    Label(text="Second state", duration=2)
    Label(text="Third state", duration=2)

exp.run()
```

As shown above, the default state of an experiment is a `Serial` state in which all of the states initialized between *exp = Experiment()* and *exp.run()* are children.

For more details, see the `Serial` docstring.

Parallel State

A `Parallel` state is a state that has children and runs those children simultaneously with each other, which we call parallel. The key to a *Parallel* state is that it will not end unless all of its children end. Once it has no more children running, the current state will schedule its own end call, allowing the next state to run.

The exception to this rule is a parameter called *blocking*. It is a Boolean property of every state. If set to False and the state exists as a child of a *Parallel* state, it will not prevent the *Parallel* state from calling its own end method. This means a *Parallel* will end when all of its *blocking* states have called their end method. All remaining, non-blocking states will have their cancel method called to allow the *Parallel* state to end.

An example below has 3 `Label` states that will disappear from the screen at the same time, despite having 3 different durations.

```
from smile.common import *

exp = Experiment()
```

```
with Parallel():
    Label(text='This one is in the middle', duration=3)
    Label(text='This is on top', duration=5, blocking=False,
          center_y=exp.screen.center_y+100)
    Label(text='This is on the bottom', duration=10, blocking=False,
          center_y=exp.screen.center_y-100)

exp.run()
```

Because the second and third *Label* in the above example are *non-blocking*, the *Parallel* state will end after the first *Label*'s duration of 3 seconds instead of the third *Label*'s duration which was 10 seconds.

For more details, see the `Parallel` docstring.

Meanwhile State

A `Meanwhile` state is one of two parallel with previous states. A *Meanwhile* will run all of its children in a `Serial` state and then run that in `Parallel` with the previous state in the stack. A *Meanwhile* state will end when either all of its children have left, or if the previous state has left. In simpler terms, a *Meanwhile* state runs while the previous state is still running. If the previous state ends before the *Meanwhile* has ended, then the *Meanwhile* will cancel all of its remaining children.

If a *Meanwhile* is created and there is no previous state, then all of the children of the *Meanwhile* will run until they end or until the experiment is over. An example of this would be if *Meanwhile* were inserted right after the line *exp = Experiment()*.

The following example shows how to use a *Meanwhile* to create an instructions screen that waits for a keypress to continue.

```
from smile.common import *

exp = Experiment()

KeyPress()
with Meanwhile():
    Label(text="THESE ARE YOUR INSTRUCTIONS, PRESS ENTER")

exp.run()
```

As soon as the `KeyPress` state ends, the `Label` will disappear off the screen because the *Meanwhile* will have canceled it.

For more details, see the `Meanwhile` docstring.

UntilDone State

An `UntilDone` state is one of two parallel with previous states. An *UntilDone* state will run all of its children in a `Serial` state and then run them in a `Parallel` with the previous state. An *UntilDone* state will end when all of its children are finished. Once the *UntilDone* ends, it will cancel the previous state if still running.

If an *UntilDone* is created and there is no previous state (right after the *exp = Experiment()* line), all of the children of the *UntilDone* will run until they end. The experiment will then end.

The following example shows how to use an *UntilDone* to create an instructions screen that waits for a keypress to continue.

```
from smile.common import *

exp = Experiment()

Label(text="THESE ARE YOUR INSTRUCTIONS, PRESS ENTER")
with UntilDone():
    KeyPress()

exp.run()
```

For more details, see the `UntilDone` docstring.

Wait State

A `Wait` state is a very simple state that has a lot of power behind it. This is particularly useful when coordinating the timings different action states. There are other options which can add to the wait to make it more complicated. The *jitter* parameter allows for the *Wait* to pause an experiment for the *duration* plus a random number between 0 and *jitter* seconds.

The unique characteristic a *Wait* state has is the ability to wait until a conditional is evaluated to True. The *Wait* will create a `Ref` that will *call_back Wait* to alert it to a change in value. Once that change evaluates to True, the *Wait* state will stop waiting and call its own end method.

An example below outlines how to use all the functionality of *Wait*. This example wants a `Label` to appear on the screen right after another *Label* does. Since the first *Wait* has a *jitter*, it is impossible to detect how long that would be, so there resides a second *Wait* wait until lb1 has an *appear_time*.

```
from smile.common import *

exp = Experiment()

with Parallel():
    with Serial():
        Wait(duration=3, jitter=2)
        lb16 = Label(text="Im on the screen now", duration=2)
    with Serial():
        Wait(until=lb1.appear_time['time']!=None)
        lb2 = Label(text="Me Too!", duration=2,
                    center_y=exp.screen.center_y-100)

exp.run()
```

For more details, see the `Wait` docstring.

If, Elif, and Else States

These 3 states are how SMILE handles branching in an experiment. Only a `If` state is needed to create a branch. Through the use of the `Elif` and the `Else` state, much more complex experiments can be created.

An *If* state runs all of its children in serial only if its conditional statement is considered True. Below is a simple of an *If* state.

```
from smile.common import *
exp = Experiment()
exp.a = 1
exp.b = 1
with If exp.a == exp.b:
```

```
        Label(text="CORRECT")
exp.run()
```

Here, *exp.a == exp.b* is the conditional statement. This *If* state expresses that if the conditional *exp.a == exp.b* is True, then the experiment will display the Label "CORRECT". In this case, if the conditional was False (say exp.b = 2 instead of 1), then the experiment will not display the Label.

An *Elif* statement, short for "Else if", is another conditional statement. It functions the same as the pythonic "elif". An *Else* statement is identical to the pythonic "else". The following is a 4 option if test.

```
from smile.common import *

exp = Experiment()

Label(text='PRESS A KEY')
with UntilDone():
    kp = KeyPress()

with If(kp.pressed == "SPACEBAR"):
    Label(text="YOU PRESSED SPACE", duration=3)

with Elif(kp.pressed == "J"):
    Label(text="YOU PRESSED THE J KEY", duration=3)

with Elif(kp.pressed == "F"):
    Label(text="YOU PRESSED THE K KEY", duration=3)

with Else():
    Label(text="I DONT KNOW WHAT YOU PRESSED", duration=3)

exp.run()
```

For more details, see the:py:class:~*smile.state.If*, `Elif`, or `Else` docstrings.

Loop State

A `Loop` state can handle any kind of looping needed. The main use for a *Loop* state is to loop over a list of dictionaries that contain stimuli. Loops can also be created by passing in a *conditional* parameter. Lastly, instead of looping over a list of dictionaries, *Loop* states can be used to loop an exact number of times by passing in a number as a parameter.

A *Loop* state requires a variable to be defined to access all of the information about the loop. This can be performed by utilizing the pythonic *as* keyword. *with Loop(list_of_dic) as trial:* is the line that defines the loop. If access to the current iteration of a loop is needed, 'trial.current' can be utilized.

Refer to the :py:class:'~smile.state.Loop'* docstring for information on how to access the different properties of a *Loop*.

Below is an example of all 3 loops.

List of Dictionaries

```
from smile.common import *

#List Gen
list_of_dic = [{'stim':"STIM 1", 'dur':3},
               {'stim':"STIM 2", 'dur':2},
               {'stim':"STIM 3", 'dur':5},
               {'stim':"STIM 4", 'dur':1}]
```

```
# Initialize the Experiment
exp = Experiment()

# The *as* operator allows one to gain access
# to the data inside the *Loop* state
with Loop(list_of_dic) as trial:
    Label(text=trial.current['stim'], duration=trial.current['dur'])

exp.run()
```

Loop a number of times:

```
from smile.common import *

exp = Experiment()

with Loop(10):
    Label(text='This will show up 10 times!', duration=1)
    Wait(1)

exp.run()
```

Loop while something is true:

```
from smile.common import *

exp = Experiment()

exp.test = 0

# Never use *and* or *or*. Always use *&* and *|* when dealing
# with references. Conditional References only work with
# absolute operators, not *and* or *or*
with Loop(conditional = (exp.test < 10)):
    Label(text='This will show up 10 times!', duration=1)
    Wait(1)
    exp.test = exp.test + 1

exp.run()
```

For more details, see the `Loop` docstrings.

The Action States of SMILE

The other basic type of SMILE states are the **Action** states. The Action states handle both the input and output in experiments. The following are subclasses of WidgetState.

Note: Heads up: All visual states that are wrapped by WidgetState are Kivy Widgets. That means all of their individual sets of parameters are located on Kivy's website. For all of the parameters that every single WidgetState shares, refer to the WidgetState Doctring.

Debug

`Debug` is a `State` that is primarily used to print out the values of references to the command line. This **State** should not be used as a replacement for **print** during experimental runtime. It should only be used to print the current values

of references during the experimental runtime.

You can give a **Debug** state a *name* to distinguish it from other **Debug** states that you might be running. **Debug** work with keyword arguments. Below is an example for how to properly use the **Debug** state and the sample output that it produces.

```
from smile.common import *

exp = Experiment()

lbl = Label(text="Hello, World", duration=2)
Wait(until=lbl.disappear_time)
Debug(name="Label appear debug", appear=lbl.appear_time['time'],
      disappear=lbl.disappear_time['time'])

exp.run()
```

And it would output:

```
DEBUG (file: 'debug_example.py', line: 7, name: Label appear debug) - lag=0.012901s
    appear: 1468255447.360574
    disappear: 1468255449.359951
```

For more details, see the `Debug` docstring.

Func

`Func` is a `State` that can run a function during Experimental Runtime. The first argument is always the name of the function and the rest of the arguments are sent to the function. You can pass in parameters to the **Func** state the same way you would pass them into the function you are wanting to run during experimental runtime. In order to access the return value of the function passed in, you need to access the *.result* attribute of the **Func** state.

The following is an example on how to run a predefined function during experimental runtime.

```
from smile.common import *

def pre_func(i):
    return i * 50.7777

exp = Experiment()

with Loop(100) as lp:
    rtrn = Func(pre_func,lp.i)
    Debug(i = rtrn.result)

exp.run()
```

For more details click `Func`.

Label

`Label` is a `WidgetState` that displays text on the screen for a *duration*. The parameter to interface with its output is called *text*. The label will display any string that is passed into *text*. *Text_size* can also be set, which is a tuple that contains (width, height) of the area the text resides in. If a goal in an experiment is to display multiple lines of text on the screen, this parameter is helpful through passing in (width_of_text, None) so the amount of text is not restricted in the vertical direction.

The following is a Label displaying the word "BabaBooie":

```
from smile.common import *

exp = Experiment()

Label(text="Hello, World", duration=2, text_size=(500,None))

exp.run()
```

For more details, see the `Label` docstring.

Image

`Image` is a `WidgetState` that displays an image on the screen for a *duration*. The parameter to interface with its output is called *source*. A string path-name is passed into the desired image to be presented onto the screen. The *allow_stretch* parameter can be set to True if the original image needs to be presented at a different size. The *allow_stretch* parameter will stretch the image to the size of the widget without changing the original ratio of width to height.

By setting *allow_stretch* to True and *keep_ratio* to False the image will stretch to fill the entirety of the widget.

Below is an example of an image at the path "test_image.png" to be presented to the center of the screen:

```
from smile.common import *

exp = Experiment()

Image(source="test_image.png", duration=3)

exp.run()
```

For more details, see the `Image` docstring.

Video

`Video` is a `WidgetState` that shows a video on the screen for a *duration*. The parameter to interface with its output is called *source*. A string path-name to the video can be passed in to present the video on the screen. The video will play from the beginning for the *duration* of the video. The *allow_stretch* parameter can be set to True if changing the video size from the original size is desired. Afterwards, the video will attempt to fill the size of the *Video* Widget without changing the aspect ratio. Setting the *keep_ratio* parameter to False will completely fill the *Video* Widget with the video. There is also the *position* parameter, which has to be between 0 and the *duration* parameter, telling the video where to start.

Below is an example of playing a video at the path "test_video.mp4" that starts 4 seconds into the video and plays for the entire duration (duration=None):

```
from smile.common import *

exp = Experiment()

Video(source="test_video.mp4", position=4)

exp.run()
```

For more details, see the `Video` docstring.

Vertex Instructions

Each **Vertex Instruction** outlined in *video.py* displays a predefined shape on the screen for a *duration*. The following are all of the basic Vertex Instructions that SMILE implements:

- `Bezier`
- `Mesh`
- `Point`
- `Triangle`
- `Quad`
- `Rectangle`
- `BorderImage`
- `Ellipse`

The parameters for each of these vary, but just like any other SMILE state, they take the same parameters as the default *State* class. They are Kivy widgets wrapped in our *WidgetState* class. Kivy documentation can be referred to for understanding how to use them or what parameters they take.

Beep

`Beep` is a state that plays a beep noise at a set frequency and volume for a *duration*. The four parameters needed to set the output of this **Beep** are *freq*, *volume*, *fadein*, and *fadeout*. *freq* and *volume* are used to set the frequency and the volume of the **Beep**. *freq* defaults to 400 Hz and *volume* defaults to .5 the max system volume. *fadein* and *fadeout* are in seconds, and they represent the time it takes to get from 0 to *volume* and *volume* to 0 respectively.

Below is an example of a beep at 555hz for 2 seconds with no fade in or out while at 50% volume:

```
from smile.common import *

exp = Experiment()

Beep(freq=555, volume=0.5, duration=2)

exp.run()
```

For more details, see the `Beep` docstring.

SoundFile

`SoundFile` is a state that plays a sound file - such as an mp3 - for a *duration* that defaults to the duration of the file. The parameter used to interface with the output of this state is *filename*. *filename* is the path name to the sound file saved on the computer. *volume* is a float from 1 to 0 where 1 is the max system volume.

The *start* parameter allows for sound files to begin at the desired point in the audio file. By using the *start* parameter, the audio will begin however many seconds into the audio file as desired.

The *end* parameter allows for sound files to end before the original end of the audio. The *end* parameter must be set to however many seconds from the beginning of the sound file it is desired to end at. The parameter must be greater than the value of *start*.

If the *loop* parameter is set to True, the sound file will run on a loop for the *duration* of the **State**.

Below is an example of playing a sound file at path "test_sound.mp3" at 50% volume for the full duration of the sound file:

```
from smile.common import *

exp = Experiment()

SoundFile(source="test_sound.mp3", volume=0.5)

exp.run()
```

For more details, see the `SoundFile` docstring.

RecordSoundFile

`RecordSoundFile` will record any sound coming into a microphone for the *duration* of the state. The audio recording will be saved to an audio file named after what is passed into the *filename* parameter.

Below is an example of recording sound for 10 seconds while looking at a Label that says "PLEASE TALK TO YOUR COMPUTER". It then saves the recording as "new_sound.mp3":

```
from smile.common import *

exp = Experiment()

Label(text="PLEASE TALK TO YOUR COMPUTER")
# UntilDone to cancel the label after the sound file
# is done recording.
with UntilDone():
    RecordSoundFile(filename="new_sound.mp3", duration = 10)

exp.run()
```

For more details, see the `RecordSoundFile` docstring.

Button

`Button` is a visual and an input state that draws a button on the screen with optional text in the button for a specified *duration*. Every button can be set to have a *name* that can be referenced by `ButtonPress` states to determine if the *correct* button was pressed. See the SMILE tutorial example for *ButtonPress* for more information.

Below is an example of a Form, where a `Label` state will ask someone to type in an answer to a `TextInput`. Then they will press the button when they are finished typing:

```
from smile.common import *

from smile.video import TextInput

exp = Experiment()

# Show both the Label and the TextInput at the same time
# during the experiment
with Parallel():
    # Required to show the mouse on the screen during the experiment!
    MouseCursor()
    Label(text="Hello, Tell me about your day!", center_y=exp.screen.center_y+50)
    TextInput(text="", width=500, height=200)

# When the button is pressed, the Button state ends, causing
# the parallel to cancel all of its children, the Label and the
```

```
# TextInput
with UntilDone():
    # A ButtonPress will end whenever one of its child buttons
    # is pressed.
    with ButtonPress():
        Button(text="Enter")

exp.run()
```

For more details, see the `Button` docstring.

ButtonPress

`ButtonPress` is a parent state, much like `Parallel`, that will run until a button inside of it is pressed. When defining a **ButtonPress** state, The name of a button inside of the parent state can be designated as the correct button to press by passing the string *name* of the correct **Button** or **Buttons** into the *correct_resp* parameter. Refer to the **ButtonPress** example in the SMILE tutorial document.

The following is an example of choosing between 3 buttons where only one of the buttons is the correct button to click:

```
from smile.common import *

exp = Experiment()

# A ButtonPress will end whenever one of its child buttons
# is pressed.
with ButtonPress(correct_resp=['First_Choice']) as bp:
    # Required to do anything with buttons.
    MouseCursor()
    Label(text="Choose WISELY")
    # Define both buttons, giving both unique names
    Button(name="First_Choice",text="LEFT CHOICE", center_x=exp.screen.center_x-200)
    Button(name="Second_Choice",text="RIGHT CHOICE", center_x=exp.screen.center_x+200)
Label(text=bp.pressed, duration=2)

exp.run()
```

For more details, see the `ButtonPress` docstring.

KeyPress

`KeyPress` is an input state that waits for a keyboard press during its *duration*. A list of strings can be passed in as parameters that are acceptable keyboard buttons into *keys*. A correct key can be selected by passing in its string name as a parameter to *correct_resp*.

Access to the information about the **KeyPress** state by can be achieved by using the following attributes:

> -pressed : a string that is the name of the key that was pressed. -press_time : a float value of the time when the key was pressed. -correct : a boolean that is whether or not they pressed the correct_resp -rt : a float that is the reaction time of the keypress. It is *press_time* - *base_time*.

The following is a keypress example that will identify what keys were pressed.

```
from smile.common import *

exp = Experiment()
```

```
with Loop(10):
    # Wait until any key is pressed
    kp = KeyPress()
    # Even though kp.pressed is a reference, you are able
    # to concatenate strings together
    Label(text="You Pressed :" + kp.pressed, duration = 2)

exp.run()
```

For more details, see the `KeyPress` docstring.

KeyRecord

`KeyRecord` is an input state that records all of the keyboard inputs for its *duration*. This state will write out each keypress during its *duration* to a *.slog* file.

The following example will save out a *.slog* file into log_bob.slog after recording all of the keypresses during a 10 second period:

```
from smile.common import *

exp = Experiment()

KeyRecord(name="Bob", duration = 10)

exp.run()
```

For more details, see the `KeyRecord` docstring.

MouseCursor

`MouseCursor` is a visual state that shows the mouse for its *duration*. In order to effectively use **ButtonPress** and **Button** states, **MouseCursor** in parallel must be used. Refer to the **ButtonPress** example in the SMILE tutorial page for more information.

The cursor image and the offset of the image can also be set as parameters to this state. Any image passed in filename will be presented on the screen, replacing the default mouse cursor.

The following example is of a mouse cursor that needs to be presented with an imaginary image to be displayed as the cursor. Since the imaginary image is 100 by 100 pixels, and it points to the center of the image, we want the offset of the cursor to be (50,50) so that the actual *click* of the mouse is in the correct location:

```
from smile.common import *

exp = experiment()

MouseCursor(duration = 10, filename="mouse_test_pointer.png", offset=(50,50))

exp.run()
```

For more details, see the `MouseCursor` docstring.

For more useful mouse tutorials, see the **Mouse Stuff** section of the Tutorial document.

Special Examples

This section is designed to develop techniques on how to use more advanced states and advanced interactions with other states in SMILE. For more detailed real life examples of experiments, reference Full Experiments page!

Subroutine

A subroutine is a stand-alone state that performs a specific action; an action that is often called multiple times in an experiment. The experiment will provide parameters for the subroutine, but not alter the subroutine itself.

Note: It's the experiment's responsibility to save logging information that comes from the subroutine. The coder should make sure that the subroutine is providing the desired information to the experiment, which the experiment may save.

This tutorial covers how to write custom `subroutine` states.In SMILE, a `subroutine` state is used. to compartmentalize a block of states that a researcher reuses in different experiments. The following example is an overview of a list presentation subroutine.

First, create a new directory called *ListPresentTest* and then create a new file in that directory called *list_present.py*. Next, import the necessary packages and define the subroutine for the list presentation.

```
from smile.common import *

@Subroutine
def ListPresent(self,
                listOfWords=[],
                interStimDur=.5,
                onStimDur=1,
                fixation=True,
                fixDur=1,
                interOrientDur=.2):
```

By placing *@Subroutine* above the subroutine definition, the compiler is told to treat this as a SMILE `subroutine`. The subroutine will eventually present a fixation cross, wait, present the stimulus, wait again, and then repeat for all of the list items it is passed. Just like calling a function or declaring a state, call `subroutine` in the body of the experiment and pass in the variables into *main_list_present.py*, which will be created later.

Warning: Always have *self* as the first argument when defining a subroutine. If you don't, your code will not work as intended.

A powerful feature of `subroutine` is that any variable declared into 'self' can be accessed outside of the subroutine. So, add a few of the following to the subroutine:

```
@Subroutine
def ListPresent(self,
                listOfWords=[],
                interStimDur=.5,
                onStimDur=1,
                fixDur=1,
                interOrientDur=.2):

    self.timing = []
```

The only variable needed for later testing is an element to hold all of the timing information to pass out into the experiment.

Next, add the stimulus loop:

```
@Subroutine
def ListPresent(self,
                listOfWords=[],
                interStimDur=.5,
                onStimDur=1,
                fixDur=1,
                interOrientDur=.2):
    self.timing = []

    with Loop(listOfWords) as trial:
        fix = Label(text='+', duration=fixDur)
        oriWait = Wait(interOrientDur)
        stim = Label(text=trial.current, duration=onStimDur)
        stimWait = Wait(interStimDur)
        self.timing += [Ref(dict,
                            fix_dur=fix.duration,
                            oriWait_dur=oriWait.duration,
                            stim_dur=stim.duration,
                            stimWait_dur=stimWait.duration)]
```

At this point the subroutine is finished. The *mainListPresent.py* needs to be written next. All that is needed is generation of a list of words to be passed into the new subroutine.

Finished main_list_present.py

```
from smile.common import *
from list_present import ListPresent
import random

WORDS_TO_DISPLAY = ['The', 'Boredom', 'Is', 'The', 'Reason', 'I',
                    'started', 'Swimming', 'It\'s', 'Also', 'The',
                    'Reason', 'I','Started', 'Sinking','Questions',
                    'Dodge','Dip','Around','Breath','Hold']
INTER_STIM_DUR = .5
STIM_DUR = 1
INTER_ORIENT_DUR = .2
ORIENT_DUR = 1
random.shuffle(WORDS_TO_DISPLAY)
exp = Experiment()

lp = ListPresent(listOfWords=WORDS_TO_DISPLAY, interStimDur=INTER_STIM_DUR,
                 onStimDur=STIM_DUR, fixDur=ORIENT_DUR,
                 nterOrientDur=INTER_ORIENT_DUR)
Log(name='LISTPRESENTLOG',
    timing=lp.timing)
exp.run()
```

Finished list_present.py

```
from smile.common import *

@Subroutine
def ListPresent(self,
                listOfWords=[],
```

```
                interStimDur=.5,
                onStimDur=1,
                fixDur=1,
                interOrientDur=.2):
    self.timing = []
    with Loop(listOfWords) as trial:
        fix = Label(text='+', duration=fixDur)
        oriWait = Wait(interOrientDur)
        stim = Label(text=trial.current, duration=onStimDur)
        stimWait = Wait(interStimDur)
        self.timing += [Ref(dict,
                            fix_dur=fix.duration,
                            oriWait_dur=oriWait.duration,
                            stim_dur=stim.duration,
                            stimWait_dur=stimWait.duration)]
```

ButtonPress

In this section, the `ButtonPress` state and the `MouseCursor` state will be examined. The following is a simple experient that allows a participant to click a button on the screen and then reports if the correct button was chosen.

Notice that this code, `ButtonPress`, acts as a `Parallel` state. This means that all of the states defined within `ButtonPress` become its children. The field *correct* that is passed into `ButtonPress` takes the *name* of the correct button for the participant as a string.

When defining **Buttons** within button press, the *name* attribute of each should be set to something different. That way, when reviewing post-experiment data, it is easy to distinguish which button the participant pressed.

When making an experiment with buttons, the cursor used to make the selections (such as a mouse cursor) is a necessesary consideration. The `MouseCursor` state handles this. By default, the experiment hides the mouse cursor. In order to allow the participant to see where they are clicking, a `MouseCursor` state must be included in the `ButtonPress` state. If the participant needs to use the mouse for the duration of an experiment, call the `MouseCursor` state just after assignment of the `Experiment` variable.

Finished button_press_example.py

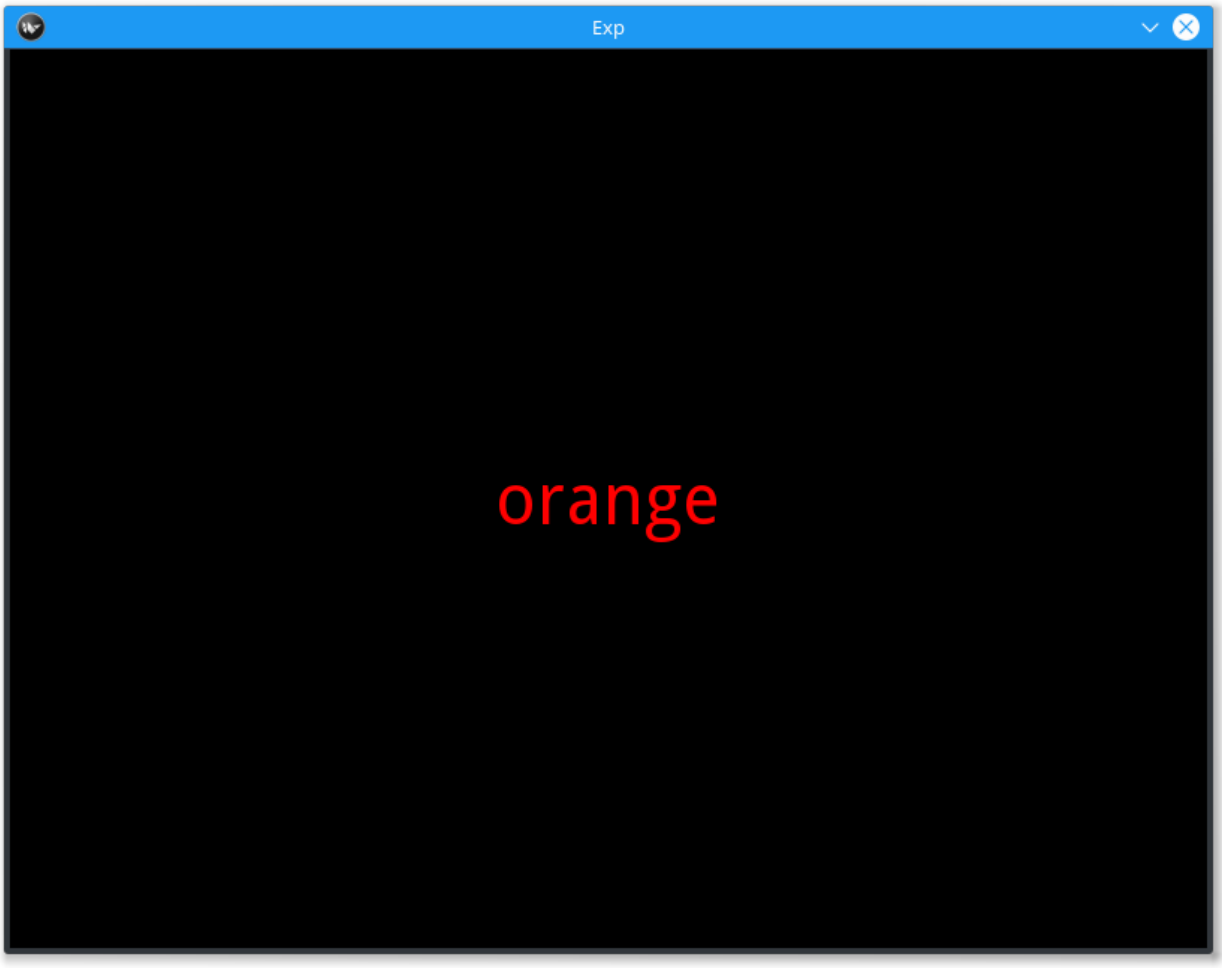
```
from smile.common import *

exp = Experiment()

#From here you can see setup for a ButtonPress state.
with ButtonPress(correct_resp='left', duration=5) as bp:
    MouseCursor()
    Button(name='left', text='left', left=exp.screen.left,
           bottom=exp.screen.bottom)
    Button(name='right', text='right', right=exp.screen.right,
           bottom=exp.screen.bottom)
    Label(text='PRESS THE LEFT BUTTON FOR A CORRECT ANSWER!')
Wait(.2)
with If(bp.correct):
    Label(text='YOU PICKED CORRECT', color='GREEN', duration=1)
with Else():
    Label(text='YOU WERE DEAD WRONG', color='RED', duration=1)

exp.run()
```

Full Experiments

Below are a few links to full, recognizable experiments that were coded up in SMILE. They include the idea behind the experiment, an explanation of the code, and they include a mini-analysis of the data collected. These real world examples will provide a better understanding into exactly how to code a SMILE experiment in real world conditions, rather than in bite-sized samples of code.

Experiments

Stroop Task



This is the stroop task. The participant is required to view a list of words, appearing one at a time on the screen, and say out loud the color of the text. Each sound file corresponding to each trial are saved out as *.wav* files, with the block and trial number in the filename.

The Experiment

First, let's do the imports that we need for this experiment. We are also going to execute the config.py file and the gen_stim.py file.

```
from smile.common import *
from smile.audio import RecordSoundFile

#execute both the configuration file and the
#stimulus generation file
from config import *
from gen_stim import *
```

For this experiment we defined two functions that would generate our list of lists of dictionaries full of the information we need to run each trial of our experiment. The first is called *gen_lists()*. The following is *gen_stim.py*.

```
def gen_lists():
    #First, let's define some variables.
    num_of_blocks = 4 #This is an arbitrary number of blocks.
    len_of_blocks = 24 #Once again, an arbitrary number of words in the block.
    total_words = num_of_blocks * len_of_blocks #The total number of words.
    dict_list = [] #The list to hold the dictionaries
    sample_list = [] #This list will hold a few dictionaries in order to provide a sample.

    """
    We will be creating dictionaries with the following keys:
        word                The actual word.
        color               The color the word will be presented as.
        matched             True or false (True if the word describes its own color, false otherwise

    """

    #So, now we begin to create the lists.
    for y in range (num_of_blocks):
        for x in range(len_of_blocks/8):
            block_list = []
            #This block will create the matched word/color pairs.
            r_trial = {'word':'red', 'color':'RED', 'matched':True}
            block_list.append(r_trial)
            sample_list.append(r_trial)
            b_trial = {'word':'blue', 'color':'BLUE', 'matched':True}
            block_list.append(b_trial)
            sample_list.append(b_trial)
            g_trial = {'word':'green', 'color':'GREEN', 'matched':True}
            block_list.append(g_trial)
            sample_list.append(g_trial)
            o_trial = {'word':'orange', 'color':'ORANGE', 'matched':True}
            block_list.append(o_trial)
            sample_list.append(o_trial)

            #This set of four will create the mismatched color lists.
            rf_trial = {'word':'red', 'color':randomize_color('red', x%3), 'matched':False}
            block_list.append(rf_trial)
            sample_list.append(rf_trial)
            bf_trial = {'word':'blue', 'color':randomize_color('blue', x%3), 'matched':False}
            block_list.append(bf_trial)
            sample_list.append(bf_trial)
            gf_trial = {'word':'green', 'color':randomize_color('green', x%3), 'matched':False}
            block_list.append(gf_trial)
            sample_list.append(gf_trial)
            of_trial = {'word':'orange', 'color':randomize_color('orange', x%3), 'matched':False}
            block_list.append(of_trial)
            sample_list.append(of_trial)
```

```
48
49              #And now we shuffle the lists to ensure randomness.
50              shuffle(block_list)
51              dict_list.append(block_list)
52      shuffle(dict_list)
53      return(dict_list, sample_list)
```

Inside this function we call another function that we used to give us the color of the mismatched trials. This function is called *randomize_color()*. This function will return a string representative of the color that that text of this trial will be. The following is the rest of *gen_stim.py*.

```
:lineno-start: 54

#This function will essentially select a random color from blue, orange, green, and red from amongst
def randomize_color(sColor, iColor):

    final_color = ''
    if(sColor == 'red'):
        if(iColor == 0):
            final_color = 'BLUE'
        elif(iColor == 1):
            final_color = 'ORANGE'
        else:
            final_color = 'GREEN'
    elif(sColor == 'blue'):
        if(iColor == 0):
            final_color = 'RED'
        elif(iColor == 1):
            final_color = 'GREEN'
        else:
            final_color = 'ORANGE'
    elif(sColor == 'green'):
        if(iColor == 0):
            final_color = 'ORANGE'
        elif(iColor == 1):
            final_color = 'BLUE'
        else:
            final_color = 'RED'
    elif(sColor == 'orange'):
        if(iColor == 0):
            final_color = 'RED'
        elif(iColor == 1):
            final_color = 'GREEN'
        else:
            final_color = 'BLUE'
    return final_color
#Generate the Stimulus
trials, sample_list = gen_lists(NUMBLOCKS, LENBLOCKS)
```

Now that we have our list gen setup, let's run our list gen and setup our experiment variables. The following is *config.py*.

```
1  #Read in the instructions
2  instruct_text = open('stroop_instructions.rst', 'r').read()
3  RSTFONTSIZE = 30
4  RSTWIDTH = 900
5  NUMBLOCKS = 4
6  LENBLOCKS = 24
7  recDuration = 2
```

```
interBlockDur = 2
interStimulusInterval = 2
```

Now we can start building our stroop experiment. The first line we run is *exp = Experiment()* to tell **SMILE** that we are ready to start defining the states in our state machine. The main states we are going to need when presenting any stimulus, in our case `Labels` of text, are `Loops`. The other state will be needed is the `Wait` state, to provide a much needed slight delay in the stimulus.

Below are the first few lines of our experiment. We setup the experiment variables and the loops that drive our experiment.

```
#Define the Experiment Variable
exp = Experiment()

#Show the instructions as an RstDocument Viewer on the screen
init_text = RstDocument(text=instruct_text, font_size=RSTFONTSIZE, width=RSTWIDTH, top=exp.screen.top
with UntilDone():
    #Once you press any key, the UntilDone will cancel the RstDocument,
    #allowing the rest of the experiment to continue running.
    keypress = KeyPress()

#Initialize the block counter, only used because we need
#unique names for the .wav files later.
exp.blockNum = 0

#Initialize the Loop as "with Loop(list_like) as reference_variable_name:"
with Loop(trials) as block:
    #Initialize the trial counter, only used because we need
    #unique names for the .wav files later.
    exp.trialNum = 0

    inter_stim = Label(text = '+', font_size = 80, duration = interBlockDur)
    #Initialize the Loop as "with Loop(list_like) as reference_variable_name:"
    with Loop(block.current) as trial:
```

We have now declared our 2 loops. One is to loop over our blocks, and one is to loop over our trials in each block. We also put an inter-stimulus fixation cross to show the participant where the stimulus will be presented. The next step is to define how our action states will work.

```
        #Display the word, with the appropriate colored text
        t = Label(text=trial.current['word'], font_size=48, color=trial.current['color'])
        with UntilDone():
            #The Label will stay on the screen for as long as
            #the RecordSoundFile state is active. The filename
            #for this state is different for each trial in each block.
            rec = RecordSoundFile(filename="b_" + Ref(str,exp.blockNum) + "_t_" + Ref(str, exp.trialN
                                  duration=recDuration)
        #Log the color and word that was presented on the screen,
        #as well as the block and trial number
        Log(name='Stroop', stim_word=trial.current['word'], stim_color=trial.current['color'],
            block_num=exp.blockNum, trial_num=exp.trialNum)
        Wait(interStimulusInterval)
        #Wait for a duration then present the fixation
        #cross again.
        inter_stim = Label(text = '+', font_size = 80, duration = interBlockDur)
        #Increase the trialNum
        exp.trialNum += 1
    #Increase the blockNum
    exp.blockNum += 1
```

```
#Run the experiment!
exp.run()
```

Analysis

The main way to analyze this data is to run all of your *.wav* files through some kind of program that deals with sifting through the important information that each file contains to remove errors. That info is what word they are saying in it and how long, from the start of recording, it took them to respond. With those two peices of information, you would be able to run stats on them along with the data from the experiment, i.e. the color and the text of the presented item during each trial.

How you go about getting the info from the *.wav* files might be hard, but getting the data from SMILE and into a data-frame is fairly easy. Below is a the few lines of code you would use to get at all of the data from all of your participants.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
dic_list = []
for sbj in subjects:
    #get at all the different subjects
    dic_list.append(lg.log2dl(log_filename="data/" + sbj + "Log_Stroop"))
#print out all of the stimulus words of the first subject's first trial
print dic_list[0]['stim_word']
```

You can also translate all of the *.slog* files into *.csv* files easily by running the command `log2csv()` for each participant. An example of this is located below.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
for sbj in subjects:
    #Get at all the subjects data, naming the csv appropriately.
    lg.log2csv(log_filename="data/" + sbj + "Log_Stroop", csv_filename=sbj + "_Stroop")
```

stroop.py in Full

```
from smile.common import *
from smile.audio import RecordSoundFile
from random import *
from math import *

#execute both the configuration file and the
#stimulus generation file
from config import *
from gen_stim import *


#Define the Experiment Variable
exp = Experiment()

#Show the instructions as an RstDocument Viewer on the screen
init_text = RstDocument(text=instruct_text, font_size=RSTFONTSIZE, width=RSTWIDTH, top=exp.screen.top
with UntilDone():
    #Once you press any key, the UntilDone will cancel the RstDocument,
```

```
    #allowing the rest of the experiment to continue running.
    keypress = KeyPress()

#Initialize the block counter, only used because we need
#unique names for the .wav files later.
exp.blockNum = 0

#Initialize the Loop as "with Loop(list_like) as reference_variable_name:"
with Loop(trials) as block:
    #Initialize the trial counter, only used because we need
    #unique names for the .wav files later.
    exp.trialNum = 0

    inter_stim = Label(text = '+', font_size = 80, duration = interBlockDur)
    #Initialize the Loop as "with Loop(list_like) as reference_variable_name:"
    with Loop(block.current) as trial:
        #Display the word, with the appropriate colored text
        t = Label(text=trial.current['word'], font_size=48, color=trial.current['color'])
        with UntilDone():
            #The Label will stay on the screen for as long as
            #the RecordSoundFile state is active. The filename
            #for this state is different for each trial in each block.
            rec = RecordSoundFile(filename="b_" + Ref(str,exp.blockNum) + "_t_" + Ref(str, exp.trialN
                                  duration=recDuration)
        #Log the color and word that was presented on the screen,
        #as well as the block and trial number
        Log(name='Stroop', stim_word=trial.current['word'], stim_color=trial.current['color'],
            block_num=exp.blockNum, trial_num=exp.trialNum)
        Wait(interStimulusInterval)
        #Wait for a duration then present the fixation
        #cross again.
        inter_stim = Label(text = '+', font_size = 80, duration = interBlockDur)
        #Increase the trialNum
        exp.trialNum += 1
    #Increase the blockNum
    exp.blockNum += 1
#Run the experiment!
exp.run()
```

config.py in Full

```
instruct_text = open('stroop_instructions.rst', 'r').read()
RSTFONTSIZE = 30
RSTWIDTH = 900
NUMBLOCKS = 4
LENBLOCKS = 24
recDuration = 2
interBlockDur = 2
interStimulusInterval = 2
```

gen_stim.py in Full

```
def gen_lists(num_of_blocks, len_of_blocks):
    #First, let's define some variables.
    total_words = num_of_blocks * len_of_blocks #The total number of words.
```

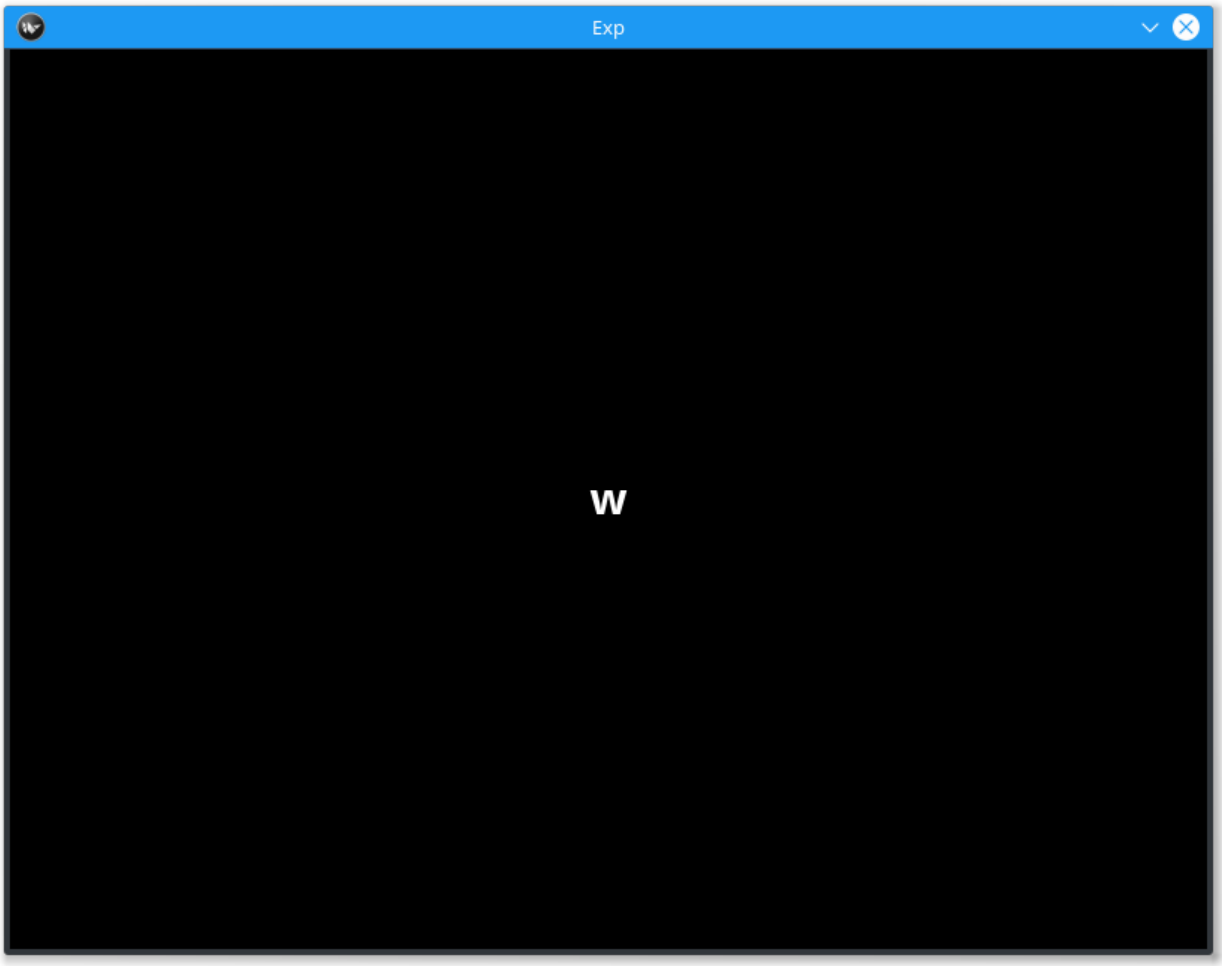
```
    dict_list = [] #The list to hold the dictionaries
    sample_list = [] #This list will hold a few dictionaries in order to provide a sample.

    """
    We will be creating dictionaries with the following keys:
        word                The actual word.
        color               The color the word will be presented as.
        matched             True or false (True if the word describes its own color, false otherwise

    """

    #Now we begin to create the lists.
    for y in range (num_of_blocks):
        for x in range(len_of_blocks/8):
            block_list = []
            #This block will create the matched word/color pairs.
            r_trial = {'word':'red', 'color':'RED', 'matched':True}
            block_list.append(r_trial)
            sample_list.append(r_trial)
            b_trial = {'word':'blue', 'color':'BLUE', 'matched':True}
            block_list.append(b_trial)
            sample_list.append(b_trial)
            g_trial = {'word':'green', 'color':'GREEN', 'matched':True}
            block_list.append(g_trial)
            sample_list.append(g_trial)
            o_trial = {'word':'orange', 'color':'ORANGE', 'matched':True}
            block_list.append(o_trial)
            sample_list.append(o_trial)

            #This set of four will create the mismatched color lists.
            rf_trial = {'word':'red', 'color':randomize_color('red', x%3), 'matched':False}
            block_list.append(rf_trial)
            sample_list.append(rf_trial)
            bf_trial = {'word':'blue', 'color':randomize_color('blue', x%3), 'matched':False}
            block_list.append(bf_trial)
            sample_list.append(bf_trial)
            gf_trial = {'word':'green', 'color':randomize_color('green', x%3), 'matched':False}
            block_list.append(gf_trial)
            sample_list.append(gf_trial)
            of_trial = {'word':'orange', 'color':randomize_color('orange', x%3), 'matched':False}
            block_list.append(of_trial)
            sample_list.append(of_trial)

            #And now we shuffle the lists to ensure randomness.
            shuffle(block_list)
            dict_list.append(block_list)
    shuffle(dict_list)
    return(dict_list, sample_list)



#This function will essentially select a random color from blue, orange, green, and red from amongst
def randomize_color(sColor, iColor):

    final_color = ''
    if(sColor == 'red'):
        if(iColor == 0):
            final_color = 'BLUE'
```

```
        elif(iColor == 1):
            final_color = 'ORANGE'
        else:
            final_color = 'GREEN'
    elif(sColor == 'blue'):
        if(iColor == 0):
            final_color = 'RED'
        elif(iColor == 1):
            final_color = 'GREEN'
        else:
            final_color = 'ORANGE'
    elif(sColor == 'green'):
        if(iColor == 0):
            final_color = 'ORANGE'
        elif(iColor == 1):
            final_color = 'BLUE'
        else:
            final_color = 'RED'
    elif(sColor == 'orange'):
        if(iColor == 0):
            final_color = 'RED'
        elif(iColor == 1):
            final_color = 'GREEN'
        else:
            final_color = 'BLUE'
    return final_color
#Generate the Stimulus
trials, sample_list = gen_lists(NUMBLOCKS, LENBLOCKS)
```

CITATION

```
Stroop, J.R. (1935), "Studies of interference in serial verbal reactions", Journal of Experimental Ps
```

Sternberg Task



This is the Sternberg task. Developed by Saul Sternberg in the 1960's, this task is designed to test a participants working memory by asking them to view a list of several stimuli, usually words, numbers, or letters, and then showing them a stimuli that may or may not have been in that list. They are then required to make a judgement on whether or not that word was in the list. Below is the SMILE version of that classic task. We use **Action** states like `KeyPress` and `Label` in this experiment, as well as **Flow** states like `UntilDone` and `Loop`.

Each participant of this experiment will have a different log that will contain all of the information about each block, as well as all of the information that would be needed to run analysis of this experiment, i.e. reaction times.

The Experiment

First, let's do the imports of the experiment. Below is the start of *stern.py*. We will also execute the configuration file and the stimulus generation file.

```
# global imports
import random
import string
# load all the states
from smile.common import *

```

```
#execute both the configuration file and the
#stimulus generation file
from config import *
from gen_stim import *
```

Easy! Now, let's also set up all the experiment variables. These are all the variables that are needed for generating stimuli, durations of states, and little things like instructions and the keys for **KeyPress** states. The following is *config.py*

```
# config vars
NUM_TRIALS = 2
#The trials, shuffled, for the stimulus generation.
NUM_ITEMS = [2,2,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,4,4,4]
random.shuffle(NUM_ITEMS)
ITEMS = string.ascii_lowercase
#instructions written in another document
instruct_text = open('stern_instructions.rst', 'r').read()
RSTFONTSIZE = 30
RSTWIDTH = 900
STUDY_DURATION = 1.2
STUDY_ISI = .4
RETENTION_INTERVAL = 1.0
#KeyPress stuff
RESP_KEYS = ['J','K']
RESP_DELAY = .2
ORIENT_DURATION = 1.0
ORIENT_ISI = .5
ITI = 1.0
FONTSIZE = 30
```

Next is the generation of our stimuli. In SMILE, the best practice is to generate lists of dictionaries to loop over, that way you don't have to do any calculations during the actual experiments. Next is the definition of a function that was written to generate a stern trial called *stern_trial()*, as well as where we call it to generate our stimulus. The following is *gen_stim.py*

```
# generate sternberg trial
def stern_trial(nitems=2, lure_trial=False,):
    if lure_trial:
        condition = 'lure'
        items = random.sample(ITEMS,nitems+1)
    else:
        condition = 'target'
        items = random.sample(ITEMS,nitems)
        # append a test item
        items.append(random.sample(items,1)[0])
    trial = {'nitems':nitems,
             'study_items':items[:-1],
             'test_item':items[-1],
             'condition':condition,}
    return trial

trials = []
for i in NUM_ITEMS:
    # add target trials
    trials.extend([stern_trial(i,lure_trial=False) for t in range(NUM_TRIALS)])
    # add lure trials
    trials.extend([stern_trial(i,lure_trial=True) for t in range(NUM_TRIALS)])

# shuffle and number
```

```
random.shuffle(trials)
for t in range(len(trials)):
    trials[t]['trial_num'] = t
```

After we generate our stimulus we need to set up our experiment. The comments in the following code explain what every few lines do.

```
#Define the experiment
exp = Experiment()
#Present the instructions to the participant
init_text = RstDocument(text=instruct_text, width=RSTWIDTH, font_size=RSTFONTSIZE, top=exp.screen.top
with UntilDone():
    #Once the KeyPress is detected, the UntilDone
    #cancels the RstDocument
    keypress = KeyPress()
# loop over study block
with Loop(trials) as trial:
    #Setup the list of study times.
    exp.study_times = []
    # orient stim
    orient = Label(text='+',duration=ORIENT_DURATION, font_size=FONTSIZE)
    Wait(ORIENT_ISI)
    # loop over study items
    with Loop(trial.current['study_items']) as item:
        # present the letter
        ss = Label(text=item.current, duration=STUDY_DURATION, font_size=FONTSIZE)
        # wait some jittered amount
        Wait(STUDY_ISI)
        # append the time
        exp.study_times+=[ss.appear_time['time']]
    # Retention interval
    Wait(RETENTION_INTERVAL - STUDY_ISI)
    # present the letter
    test_stim = Label(text=trial.current['test_item'], bold=True, font_size=FONTSIZE)
    with UntilDone():
        # wait some before accepting input
        Wait(RESP_DELAY)
        #After the KeyPress is detected, the UntilDone
        #cancels the Label test_stim and allows the
        #experiment to continue.
        ks = KeyPress(keys=RESP_KEYS,
                      base_time=test_stim.appear_time['time'])
    # Log the trial
    Log(trial.current,
        name="Stern",
        resp=ks.pressed,
        rt=ks.rt,
        orient_time=orient.appear_time['time'],
        study_times=exp.study_times,
        test_time=test_stim.appear_time['time'],
        correct=(((trial.current['condition']=='target')&
                  (ks.pressed==RESP_KEYS[0])) |
                 ((trial.current['condition']=='lure')&
                 (ks.pressed==RESP_KEYS[1]))))
    Wait(ITI)
# run that exp!
exp.run()
```

Analysis

When coding your experiment, you don't have to worry about losing any data because all of it is saved out into *.slog* files anyway. The thing you do have to worry about is whether or not you want that data to be easily available or if you want to spend hours **slogging** through your data. We made it easy for you to pick which data you want saved out during the running of your experiment with use of the **Log** state.

The relevant data that we need from a **Sternberg** task would be the reaction times for every test event, all of the presented letters from the study and test portion of the experiment, and whether they answered correctly or not. In the **Log** that we defined in our experiment above, we saved a little more than that out, because it is better to save out data and not need it, then to not save it and need it later.

If you would like to grab your data from the *.slog* files to analyze your data in python, you need to use the `log2dl()`. This function will read in all of the *.slog* files with the same base name, and convert them into one long list of dictionaries. Below is a the few lines of code you would use to get at all of the data from three imaginary participants, named as *s000*, *s001*, and *s002*.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
dic_list = []
for sbj in subjects:
    #get at all the different subjects
    dic_list.append(lg.log2dl(log_filename="data/" + sbj + "Log_Stern"))
#print out all of the study times in the first study block for
#participant one, block one
print dic_list[0]['study_times']
```

You can also translate all of the *.slog* files into *.csv* files easily by running the command `log2csv()` for each participant. An example of this is located below.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
for sbj in subjects:
    #Get at all the subjects data, naming the csv appropriately.
    lg.log2csv(log_filename="data/" + sbj + "Log_Stern", csv_filename=sbj + "_Stern")
```

stern.py in Full

```
# global imports
import random
import string
# load all the states
from smile.common import *

#execute both the configuration file and the
#stimulus generation file
from config import *
from gen_stim import *

#Define the experiment
exp = Experiment()
#Present the instructions to the participant
init_text = RstDocument(text=instruct_text, width=RSTWIDTH, font_size=RSTFONTSIZE top=exp.screen.top,
with UntilDone():
    #Once the KeyPress is detected, the UntilDone
```

```
    #cancels the RstDocument
    keypress = KeyPress()
# loop over study block
with Loop(trials) as trial:
    #Setup the list of study times.
    exp.study_times = []
    # orient stim
    orient = Label(text='+',duration=ORIENT_DURATION, font_size=FONTSIZE)
    Wait(ORIENT_ISI)
    # loop over study items
    with Loop(trial.current['study_items']) as item:
        # present the letter
        ss = Label(text=item.current, duration=STUDY_DURATION, font_size=FONTSIZE)
        # wait some jittered amount
        Wait(STUDY_ISI)
        # append the time
        exp.study_times+=[ss.appear_time['time']]
    # Retention interval
    Wait(RETENTION_INTERVAL - STUDY_ISI)
    # present the letter
    test_stim = Label(text=trial.current['test_item'], bold=True, font_size=FONTSIZE)
    with UntilDone():
        # wait some before accepting input
        Wait(RESP_DELAY)
        #After the KeyPress is detected, the UntilDone
        #cancels the Label test_stim and allows the
        #experiment to continue.
        ks = KeyPress(keys=RESP_KEYS,
                      base_time=test_stim.appear_time['time'])
    # Log the trial
    Log(trial.current,
        name="Stern",
        resp=ks.pressed,
        rt=ks.rt,
        orient_time=orient.appear_time['time'],
        study_times=exp.study_times,
        test_time=test_stim.appear_time['time'],
        correct=(((trial.current['condition']=='target')&
                  (ks.pressed==RESP_KEYS[0])) |
                  ((trial.current['condition']=='lure')&
                  (ks.pressed==RESP_KEYS[1]))))
    Wait(ITI)
# run that exp!
exp.run()
```

config.py in Full

```
# config vars
NUM_TRIALS = 2
NUM_ITEMS = [2,3,4]
ITEMS = string.ascii_lowercase
instruct_text = open('stern_instructions.rst', 'r').read()
RSTFONTSIZE = 30
RSTWIDTH = 900
STUDY_DURATION = 1.2
STUDY_ISI = .4
```

```
RETENTION_INTERVAL = 1.0
RESP_KEYS = ['J','K']
RESP_DELAY = .2
ORIENT_DURATION = 1.0
ORIENT_ISI = .5
ITI = 1.0
FONTSIZE = 30
```

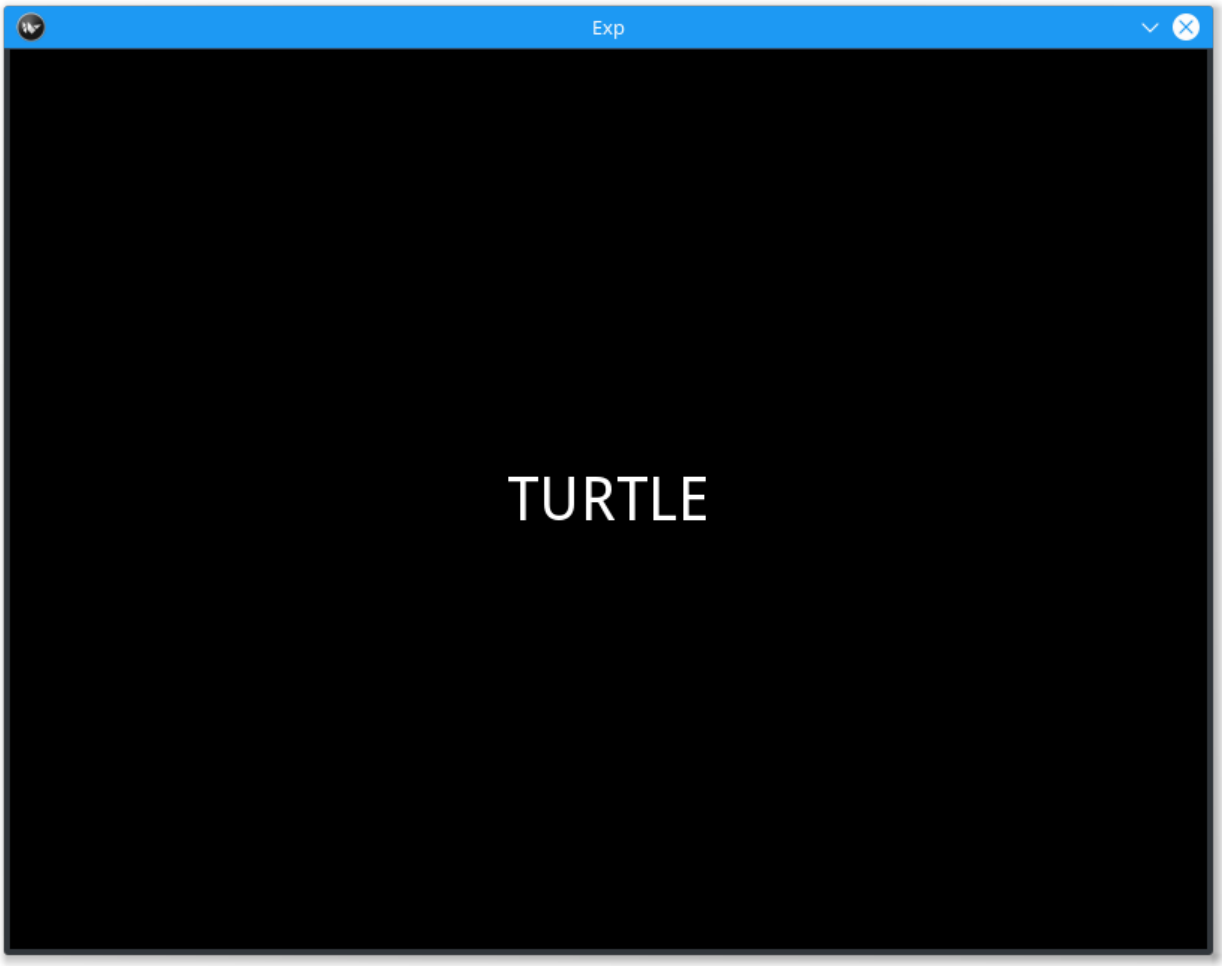
gen_stim.py in Full

```
# generate Sternberg trial
def stern_trial(nitems=2, lure_trial=False,):
    if lure_trial:
        condition = 'lure'
        items = random.sample(ITEMS,nitems+1)
    else:
        condition = 'target'
        items = random.sample(ITEMS,nitems)
        # append a test item
        items.append(random.sample(items,1)[0])
    trial = {'nitems':nitems,
             'study_items':items[:-1],
             'test_item':items[-1],
             'condition':condition,}
    return trial

trials = []
for i in NUM_ITEMS:
    # add target trials
    trials.extend([stern_trial(i,lure_trial=False) for t in range(NUM_TRIALS)])
    # add lure trials
    trials.extend([stern_trial(i,lure_trial=True) for t in range(NUM_TRIALS)])

# shuffle and number
random.shuffle(trials)
for t in range(len(trials)):
    trials[t]['trial_num'] = t
```

CITATION

```
Sternberg, S. (1966), "High-speed scanning in human memory", Science 153 (3736), 652-654
```

Free Recall



Free-Recall is a psychological paradigm where the participant is shown a list of words and is then asked to recall the displayed words in any order immediately after being shown or after a period of delay.

The kind of Free-Recall Experiment that we wrote is the Immediate Free-Recall task. Our participant will view 10, 15, or 20 words and then be asked to recall as many words as possible from the list in 20, 30, or 40 seconds respectively. This experiment will show you how to use the `Subroutine` called `FreeKey`, as well as things like `Label` and `Loop`.

Below we will show you the best practices for coding an experiment like this one.

The Experiment

The best thing to do when coding a SMILE experiment is to break up the experiment into 3 different files: the experiment file with all the SMILE code, the config file with all the experimental variables, and the stimulus generation file.

The first thing we will look at is *free_recall.py*. In this file we need to import smile as well as execute the *config.py* and the *gen_stim.py*.

```
1 #freekey.py
2 from smile.common import *
```

```
from smile.freekey import FreeKey

#execute both the configuration file and the
#stimulus generation file
from config import *
from gen_stim import *
```

Inside *config.py* we setup any variables that will need to be used during the experiment as well as open any files that we might need for list generation or instructions for the participant.

```
#Names of the stimulus files
filenameL = "pools/living.txt"
filenameN = "pools/nonliving.txt"

#Open the files and combine them
L = open(filenameL)
N = open(filenameN)
stimList = L.read().split('\n')
stimList.append(N.read().split('\n'))

#Open the instructions file
instruct_text = open('freekey_instructions.rst', 'r').read()

#Define the Experimental Variables
ISI = 2
IBI = 2
STIMDUR = 2
PFI = 4
FONTSIZE = 40
RSTFONTSIZE = 30
RSTWIDTH = 900

MINFKDUR = 20

NUMBLOCKS = 6
NUMPERBLOCK = [10,15,20]
```

Next we can take a look into our list gen. Simply, we generate a list of dictionaries where **study** points to a list of words and **duration** points to the duration that the participants have to freely recall the words.

```
import random

#Shuffle the stimulus
random.shuffle(stimList)

blocks = []
#Loop NUMBLOCKS times
for i in range(NUMBLOCKS):
    tempList = []
    #For each block, loop either 10, 15, or 20 times
    #Counter balanced to have equal numbers of each
    for x in range(NUMPERBLOCK[i%len(NUMPERBLOCK)]):
        tempList.append(stimList.pop())
    #Create tempBlock
    tempBlock = {"study":tempList,
                 "duration":(MINFKDUR + 10*i%len(NUMPERBLOCK))}
    blocks.append(tempBlock)
#Shuffle the newly created list of blocks
random.shuffle(blocks)
```

Finally we can get to the fun stuff! We now can start programming our SMILE experiment. The comments in the following section of code explain why we do each part of the experiment.

```
#Initialize the Experiment
exp = Experiment()

#Show the instructions to the participant
RstDocument(text=instruct_text, base_font_size=RSTFONTSIZE, width=RSTWIDTH, height=exp.screen.height)
with UntilDone():
    #When a KeyPress is detected, the UntilDone
    #will cancel the RstDocument state
    KeyPress()
#Start the experiment Loop
with Loop(blocks) as block:
    Wait(IBI)
    with Loop(block.current['study']) as study:
        #Present the Fixation Cross
        Label(text="+", duration=ISI, font_size=FONTSIZE)
        #Present the study item
        Label(text=study.current, duration=STIMDUR, font_size=FONTSIZE)
    Wait(PFI)
    #Start FreeKey
    fk = FreeKey(Label(text="XXXXXXX", font_size=FONTSIZE), max_duration=block.current['duration'])
    #Log everything!
    Log(block,
        name="FreeKey",
        responses = fk.responses)
#Run the experiment
exp.run()
```

Analysis

When coding your experiment, you don't have to worry about losing any data because all of it is saved out into *.slog* files anyway. The thing you do have to worry about is whether or not you want that data to be easily available or if you want to spend hours **slogging** through your data. We made it easy for you to pick which data you want saved out during the running of your experiment with use of the **Log** state.

Relevant data from the **Free-Recall** task would be the responses from each **FreeKey** state. In the **Log** that we used in the experiment above, we log everything in each *block* of the experiment, i.e. the stimulus and the duration that they are allowed to respond in, and the responses from **FreeKey**.

If you would like to grab your data from the *.slog* files to analyze your data in python, you need to use the `log2dl()`. This function will read in all of the *.slog* files with the same base name, and convert them into one long list of dictionaries. Below is a the few lines of code you would use to get at all of the data from three imaginary participants, named as *s000*, *s001*, and *s002*.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
dic_list = []
for sbj in subjects:
    #get at all the different subjects
    dic_list.append(lg.log2dl(log_filename="data/" + sbj + "Log_FreeKey"))
#print out all of the study times in the first study block for
#participant one, block one
print dic_list[0]['study_times']
```

You can also translate all of the *.slog* files into *.csv* files easily by running the command `log2csv()` for each participant. An example of this is located below.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
for sbj in subjects:
    #Get at all the subjects data, naming the csv appropriately.
    lg.log2csv(log_filename="data/" + sbj + "Log_FreeKey", csv_filename=sbj + "_FreeKey")
```

free_recall.py in Full

```
#freekey.py
from smile.common import *
from smile.freekey import FreeKey

#execute both the configuration file and the
#stimulus generation file
from config import *
from gen_stim import *

#Initialize the Experiment
exp = Experiment()

#Show the instructions to the participant
RstDocument(text=instruct_text, base_font_size=RSTFONTSIZE, width=RSTWIDTH, height=exp.screen.height)
with UntilDone():
    #When a KeyPress is detected, the UntilDone
    #will cancel the RstDocument state
    KeyPress()
#Start the experiment Loop
with Loop(blocks) as block:
    Wait(IBI)
    with Loop(block.current['study']) as study:
        #Present the Fixation Cross
        Label(text="+", duration=ISI, font_size=FONTSIZE)
        #Present the study item
        Label(text=study.current, duration=STIMDUR, font_size=FONTSIZE)
    Wait(PFI)
    #Start FreeKey
    fk = FreeKey(Label(text="XXXXXXX", font_size=FONTSIZE), max_duration=block.current['duration'])
    #Log everything!
    Log(block,
        name="FreeKey",
        responses = fk.responses)
#Run the experiment
exp.run()
```

config.py in Full

```
#Names of the stimulus files
filenameL = "pools/living.txt"
filenameN = "pools/nonliving.txt"

#Open the files and combine them
```

```
L = open(filenameL)
N = open(filenameN)
stimList = L.read().split('\n')
stimList.append(N.read().split('\n'))

#Open the instructions file
instruct_text = open('freekey_instructions.rst', 'r').read()

#Define the Experimental Variables
ISI = 2
IBI = 2
STIMDUR = 2
PFI = 4
FONTSIZE = 40
RSTFONTSIZE = 30
RSTWIDTH = 900

MINFKDUR = 20

NUMBLOCKS = 6
NUMPERBLOCK = [10,15,20]
```

gen_stim.py in Full

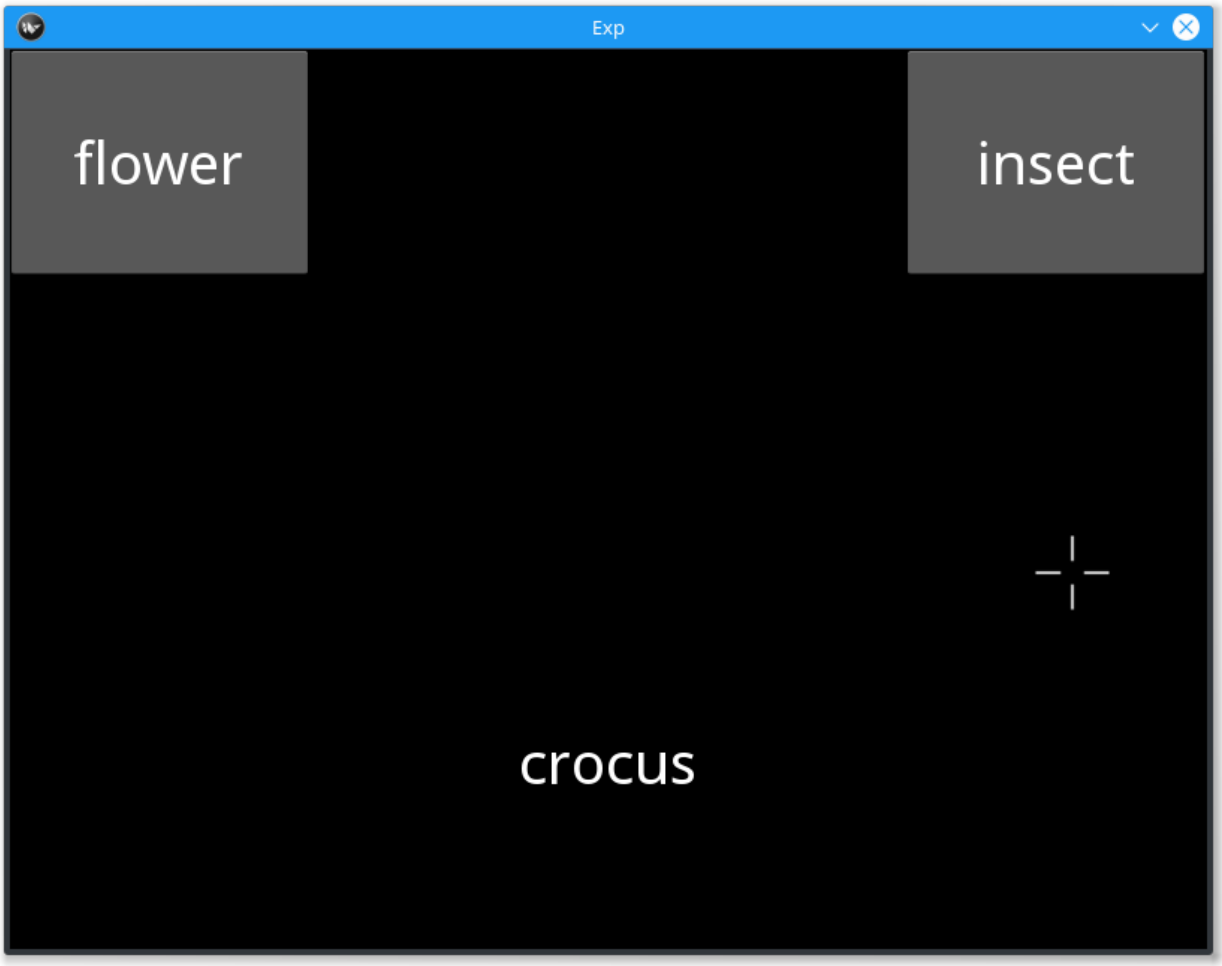
```
import random

#Shuffle the stimulus
random.shuffle(stimList)

blocks = []
#Loop NUMBLOCKS times
for i in range(NUMBLOCKS):
    tempList = []
    #For each block, loop either 10, 15, or 20 times
    #Counter balanced to have equal numbers of each
    for x in range(NUMPERBLOCK[i%len(NUMPERBLOCK)]):
        tempList.append(stimList.pop())
    #Create tempBlock
    tempBlock = {"study":tempList,
                 "duration":(MINFKDUR + 10*i%len(NUMPERBLOCK))}
    blocks.append(tempBlock)
#Shuffle the newly created list of blocks
random.shuffle(blocks)
```

CITATION

```
Murdock, Bennet B. (1962), "The serial position effect of free recall", Journal of Experimental Psych
```

```
Waugh, Nancy C. (1961), "Free versus serial recall", Journal of Experimental Psychology 62 (5): 496-5
```

IAT Mouse Tracking



The IAT, or Implicit-Association Test, was introduced by Anthony Greenwald et al. in 1998. This task is designed to get at the individual differences in our implicit associations with different concepts that make up our lives. The basic IAT requires the participant to rapidly categorize two target concepts with an attribute, such that the response times will be faster with strongly associated pairings and slower for weakly associated pairings. The key here is that the participant should act as quickly as they can so this experiment can better get at the implicit associations rather than the surface level associations.

The study that we are showing off in smile (Yu, Wang, Wang et al. 2012) designed an IAT that incorporated *mouse tracking* into their study to better get at the underlying mechanisms of implicit-association. We ask our participants to view names of flowers, names of bugs, positively associated nouns, and negatively associated nouns and to classify them into categories. The blocks of stimuli will be better explained below in the **Stimulus Generation** section of this document.

This SMILE experiment will utilize many of the *Mouse* Classes in *mouse.py*, including `MouseCursor` and `MouseRecord`. We will also be using many of the SMILE **flow** states like `Loop` and `Meanwhile` and `If`. Along with the use of **Mouse** states, we will be using `ButtonPress` as our main form of input for the experiment.

The Experiment

When writing any experiment in smile, it is usually best to split it over multiple files so that you can better organize your experiment. In this example, we split our experiment into 3 different files, *gen_stim.py*, *config.py*, and *iat_mouse.py*.

In *iat_mouse.py* we have the imports that we need for the experiment. Below are those imports.

```
from smile.common import *
from config import *
from gen_stim import *
```

Our experiment first imports smile.commmon, where all of the most used states are imported from, as well as config and gen_stim. Let's take a look into config, where we set and define our global variables for the experiment.

```
#RST VARIABLES
RSTFONTSIZE = 50
RSTWIDTH = 600
instruct1 = open('iat_mouse_instructions1.rst', 'r').read()
instruct2 = open('iat_mouse_instructions2.rst', 'r').read()
instruct3 = open('iat_mouse_instructions3.rst', 'r').read()
instruct4 = open('iat_mouse_instructions4.rst', 'r').read()
instruct5 = open('iat_mouse_instructions5.rst', 'r').read()
instruct6 = open('iat_mouse_instructions6.rst', 'r').read()
instruct7 = open('iat_mouse_instructions7.rst', 'r').read()

#MOUSE MOVING VARIABLES
WARNINGDURATION = 2.0
MOUSEMOVERADIUS = 100
MOUSEMOVEINTERVAL = 0.400

#BUTTON VARIABLES
BUTTONHEIGHT = 150
BUTTONWIDTH = 200

#GENERAL VARIABLES
FONTSIZE = 40
INTERTRIALINTERVAL = 0.750
```

After defining our global variables, we should define our stimulus generator. In *gen_stim.py* we define a function that generates lists of dictionaries that represent out blocks of trials. The following is our *gen_stim.py*, where we first set up our lists of stimuli to be pulled from.

```
import random as rm
from config import instruct1,instruct2,instruct3,instruct4,instruct5,instruct6,instruct7

# WORDLISTS FROM Greenwald et al. 1998
filenameI = "pools/insects.txt"
filenameF = "pools/flowers.txt"
filenameP = "pools/positives.txt"
filenameN = "pools/negatives.txt"

I = open(filenameI)
F = open(filenameF)
P = open(filenameP)
N = open(filenameN)

stimListI = I.read().split('\n')
stimListF = F.read().split('\n')
stimListP = P.read().split('\n')
```

```
stimListN = N.read().split('\n')

#pop off the trailing line
stimListI.pop(len(stimListI)-1)
stimListF.pop(len(stimListF)-1)
stimListP.pop(len(stimListP)-1)
stimListN.pop(len(stimListN)-1)
```

Next we define our *gen_blocks()* function. At the bottom of *gen_stim.py* we also call *gen_blocks()* so our iat_mouse.py doesn't have to.

```
def gen_blocks(type):

    sampI = rm.sample(stimListI, 10)
    sampF = rm.sample(stimListF, 10)
    sampP = rm.sample(stimListP, 10)
    sampN = rm.sample(stimListN, 10)

    #Generate the blocks
    list1 = {"left_word":"flower", "right_word":"insect", "instruct":instruct1,
             "words":([{"correct":"right", "center_word":I} for I in sampI] +
                      [{"correct":"left", "center_word":F} for F in sampF])}

    list2 = {"left_word":"positive", "right_word":"negative", "instruct":instruct2,
             "words":([{"correct":"left", "center_word":P} for P in sampP] +
                      [{"correct":"right", "center_word":N} for N in sampN])}

    list3 = {"left_word":"flower positive", "right_word":"insect negative", "instruct":instruct3,
             "words":([{"correct":"right", "center_word":I} for I in rm.sample(sampI[:], 5)] +
                      [{"correct":"left", "center_word":F} for F in rm.sample(sampF[:], 5)] +
                      [{"correct":"left", "center_word":P} for P in rm.sample(sampP[:], 5)] +
                      [{"correct":"right", "center_word":N} for N in rm.sample(sampN[:], 5)])}

    list4 = {"left_word":"flower positive", "right_word":"insect negative", "instruct":instruct4,
             "words":([{"correct":"right", "center_word":I} for I in sampI] +
                      [{"correct":"left", "center_word":F} for F in sampF] +
                      [{"correct":"left", "center_word":P} for P in sampP] +
                      [{"correct":"right", "center_word":N} for N in sampN])}

    list5 = {"left_word":"insect", "right_word":"flower", "instruct":instruct5,
             "words":[{"correct":"left", "center_word":I} for I in sampI] + [{"correct":"right", "cen

    list6 = {"left_word":"insect positive", "right_word":"flower negative", "instruct":instruct6,
             "words":([{"correct":"left", "center_word":I} for I in rm.sample(sampI[:], 5)] +
                      [{"correct":"right", "center_word":F} for F in rm.sample(sampF[:], 5)] +
                      [{"correct":"left", "center_word":P} for P in rm.sample(sampP[:], 5)] +
                      [{"correct":"right", "center_word":N} for N in rm.sample(sampN[:], 5)])}

    list7 = {"left_word":"insect positive", "right_word":"flower negative", "instruct":instruct7,
             "words":([{"correct":"left", "center_word":I} for I in sampI] +
                      [{"correct":"right", "center_word":F} for F in sampF] +
                      [{"correct":"left", "center_word":P} for P in sampP] +
                      [{"correct":"right", "center_word":N} for N in sampN])}
    rm.shuffle(list1['words'])
    rm.shuffle(list2['words'])
    rm.shuffle(list3['words'])
    rm.shuffle(list4['words'])
    rm.shuffle(list5['words'])
```

```
    rm.shuffle(list6['words'])
    rm.shuffle(list7['words'])

    #If type 1, then do critical compatible lists
    if type == 1:
        return [list1, list2, list3, list4, list5, list6, list7]
    #if type 2, then do critical incompatible lists
    else:
        return [list5, list2, list6, list7, list1, list3, list4]
#GenBlocks
BLOCKS = gen_blocks(1)
```

Now we can look at the rest of *iat_mouse.py*. The following is the setup of the block loop and the setup of the trial loop. At the beginning of each loop, you will see a new instructions page and will not be able to go on with the experiment until you press a key. The block loop will loop over the *BLOCKS* that were defined in *gen_stim.py*, whereas the trial loop will loop over the *words* key that is attached to each block's dictionary.

```
#Set up the Block loop, where *block* is a
#Reference to the variable you are looping over
with Loop(BLOCKS) as block:
    #Show the instructions to the participant
    RstDocument(text=block.current['instruct'], base_font_size=RSTFONTSIZE, width=RSTWIDTH, height=ex
    with UntilDone():
        #When a KeyPress is detected, the UntilDone
        #will cancel the RstDocument state
        KeyPress()
    #Setup a loop over each Trial in a Block. *block.current* references the
    #current iteration of the loop, which is a dictionary that contains the list
    #words. *trial* will be our reference to the current word in our loop.
    with Loop(block.current['words']) as trial:
```

The core of this experiment is the trial level loop. Below is the code that defines the states that run each and every trial for the participant. This is the section of code that defines the button press, the things that happen while the buttons are waiting to be pressed, and the Log the logs out the information from each trial. It also sets up the MouseRecord that tracks the mouse positions that need to be analyzed for this experiment.

```
        #initialize our testing variable in Experiment Runtime
        #exp.something = something will create a Set state
        exp.mouse_test = False
        #The following is a ButtonPress state. This state works like KeyPress,
        #but instead waits for any of the buttons that are its children to be
        #pressed.
        with ButtonPress(correct_resp=trial.current['correct']) as bp:
            #block.current is a dictionary that has all of the information we
            #would need during each individual block, including the text that is
            #in these buttons, which differs from block to block
            Button(text=block.current['left_word'], name="left", left=0,
                   top=exp.screen.top, width = BUTTONWIDTH, height=BUTTONHEIGHT, text_size = (170, No
                   font_size=FONTSIZE, halign='center')
            Button(text=block.current['right_word'], name="right",
                   right=exp.screen.right, top=exp.screen.top,
                   width = BUTTONWIDTH, height = BUTTONHEIGHT, text_size = (170, None),
                   font_size=FONTSIZE, halign='center')
            #Required to see the mouse on the screen!
            MouseCursor()
        #while those buttons are waiting to be pressed, go ahead and do the
        #children of this next state, the Meanwhile
        with Meanwhile():
```

```
            #The start button that is required to be pressed before the trial
            #word is seen.
            with ButtonPress():
                Button(text="Start", bottom=exp.screen.bottom, font_size=FONTSIZE)
            #Do all of the children of a Parallel at the same time.
            with Parallel():
                #display target word
                target_lb = Label(text=trial.current['center_word'], font_size=FONTSIZE, bottom=exp.s
                #Record the movements of the mouse
                MouseRecord(name="MouseMovements")
                #Setup an invisible rectangle that is used to detect exactly
                #when the mouse starts to head toward an answer.
                rtgl = Rectangle(center=MousePos(), width=MOUSEMOVERADIUS,
                                 height=MOUSEMOVERADIUS, color=(0,0,0,0))
                with Serial():
                    #wait until the mouse leaves the rectangle from above
                    wt = Wait(until=(MouseWithin(rtgl) == False))
                    #If they waited too long to start moving, tell the experiment
                    #to display a warning message to the participant
                    with If(wt.event_time['time'] - wt.start_time > MOUSEMOVEINTERVAL):
                        exp.mouse_test = True
        with If(exp.mouse_test):
            Label(text="You are taking too long to move, Please speed up!",
                  font_size=FONTSIZE, color="RED", duration=WARNINGDURATION)
        #wait for the interstimulus interval
        Wait(INTERTRIALINTERVAL)
        #WRITE THE LOGS
        Log(name="IAT_MOUSE",
            left=block.current['left_word'],
            right=block.current['right_word'],
            word=trial.current,
            correct=bp.correct,
            reaction_time=bp.press_time['time']-target_lb.appear_time['time'],
            slow_to_react=exp.mouse_test)
#This starts the experiment
exp.run()
```

Analysis

When coding your experiment, you don't have to worry about losing any data because all of it is saved out into *.slog* files anyway. The thing you do have to worry about is whether or not you want that data easily available or if you want to spend hours **slogging** through your data. We made it easy for you to pick which data you want saved out during the running of your experiment with use of the **Log** state.

Relevant data from the **IAT MOUSE TRACKING** task would be the responses from the **ButtonPress** and the mouse movements that are saved in the *.slog* files.

If you would like to grab your data from the *.slog* files to analyze your data in python, you need to use the `log2dl()`. This function will read in all of the *.slog* files with the same base name, and convert them into one long list of dictionaries. Below is a the few lines of code you would use to get at all of the data from three imaginary participants, named as *s000*, *s001*, and *s002*.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
dic_list = []
mouse_list = []
```

```
for sbj in subjects:
    #get at all the different subjects
    dic_list.append(lg.log2dl(log_filename="data/" + sbj + "Log_IAT_MOUSE"))
    mouse_list.append(lg.log2dl(log_filename="data/" + sbj + "record_MouseMovements"))
#print out all of the study times in the first study block for
#participant one, block one
print dic_list[0]['reaction_time']
```

You can also translate all of the *.slog* files into *.csv* files easily by running the command `log2csv()` for each participant. An example of this is located below.

```
from smile.log as lg
#define subject pool
subjects = ["s000/","s001/","s002/"]
for sbj in subjects:
    #Get at all the subjects data, naming the csv appropriately.
    lg.log2csv(log_filename="data/" + sbj + "Log_IAT_MOUSE", csv_filename=sbj + "_IAT_MOUSE")
    lg.log2csv(log_filename="data/" + sbj + "record_MouseMovements", csv_filename=sbj + "_IAT_MOUSE_M
```

iat_mouse.py in full

```
from smile.common import *
from config import *
from gen_stim import *

#Start setting up the experiment
exp = Experiment()

#Show the instructions to the participant
RstDocument(text=instruct_text, base_font_size=RSTFONTSIZE, width=RSTWIDTH, height=exp.screen.height)
with UntilDone():
    #When a KeyPress is detected, the UntilDone
    #will cancel the RstDocument state
    KeyPress()
#Setup the Block loop, where *block* is a
#Reference to the variable you are looping over
with Loop(BLOCKS) as block:
    #Setup a loop over each Trial in a Block. *block.current* references the
    #current iteration of the loop, which is a dictionary that contains the list
    #words. *trial* will be our reference to the current word in our loop.
    with Loop(block.current['words']) as trial:
        #initialize our testing variable in Experiment Runtime
        #exp.something = something will create a Set state
        exp.mouse_test = False
        #The following is a ButtonPress state. This state works like KeyPress,
        #but instead waits for any of the buttons that are its children to be
        #press.
        with ButtonPress(correct_resp=trial.current['correct']) as bp:
            #block.current is a dictionary that has all of the information we
            #would need during each individual block, including the text that is
            #in these buttons, which differs from block to block
            Button(text=block.current['left_word'], name="left", left=0,
                   top=exp.screen.top, width = BUTTONWIDTH, height=BUTTONHEIGHT, text_size = (170, No
                   font_size=FONTSIZE, halign='center')
            Button(text=block.current['right_word'], name="right",
                   right=exp.screen.right, top=exp.screen.top,
                   width = BUTTONWIDTH, height = BUTTONHEIGHT, text_size = (170, None),
```

```
                    font_size=FONTSIZE, halign='center')
            #Required to see the mouse on the screen!
            MouseCursor()
        #while those buttons are waiting to be pressed, go ahead and do the
        #children of this next state, the Meanwhile
        with Meanwhile():
            #The start button that is required to be pressed before the trial
            #word is seen.
            with ButtonPress():
                Button(text="Start", bottom=exp.screen.bottom, font_size=FONTSIZE)
            #Do all of the children of a Parallel at the same time.
            with Parallel():
                #display target word
                target_lb = Label(text=trial.current['center_word'], font_size=FONTSIZE, bottom=exp.s
                #Record the movements of the mouse
                MouseRecord(name="MouseMovements")
                #Setup an invisible rectangle that is used to detect exactly
                #when the mouse starts to head toward an answer.
                rtgl = Rectangle(center=MousePos(), width=MOUSEMOVERADIUS,
                                 height=MOUSEMOVERADIUS, color=(0,0,0,0))
                with Serial():
                    #wait until the mouse leaves the rectangle from above
                    wt = Wait(until=(MouseWithin(rtgl) == False))
                    #If they waited too long to start moving, tell the experiment
                    #to display a warning message to the participant
                    with If(wt.event_time['time'] - wt.start_time > MOUSEMOVEINTERVAL):
                        exp.mouse_test = True
        with If(exp.mouse_test):
            Label(text="You are taking too long to move, Please speed up!",
                  font_size=FONTSIZE, color="RED", duration=WARNINGDURATION)
        #wait the interstimulus interval
        Wait(INTERTRIALINTERVAL)
        #WRITE THE LOGS
        Log(name="IAT_MOUSE",
            left=block.current['left_word'],
            right=block.current['right_word'],
            word=trial.current,
            correct=bp.correct,
            reaction_time=bp.press_time['time']-target_lb.appear_time['time'],
            slow_to_react=exp.mouse_test)
#the line required to run your experiment after all
#of it is defined above
exp.run()
```

config.py in Full

```
#RST VARIABLES
RSTFONTSIZE = 50
RSTWIDTH = 600
instruct1 = open('iatmouse_instructions1.rst', 'r').read()
instruct2 = open('iatmouse_instructions2.rst', 'r').read()
instruct3 = open('iatmouse_instructions3.rst', 'r').read()
instruct4 = open('iatmouse_instructions4.rst', 'r').read()
instruct5 = open('iatmouse_instructions5.rst', 'r').read()
instruct6 = open('iatmouse_instructions6.rst', 'r').read()
instruct7 = open('iatmouse_instructions7.rst', 'r').read()
```

```

#MOUSE MOVING VARIABLES
WARNINGDURATION = 2.0
MOUSEMOVERADIUS = 100
MOUSEMOVEINTERVAL = 0.400

#BUTTON VARIABLES
BUTTONHEIGHT = 150
BUTTONWIDTH = 200

#GENERAL VARIABLES
FONTSIZE = 40
INTERTRIALINTERVAL = 0.750
```

gen_stim.py in Full

```
import random as rm
from config import instruct1,instruct2,instruct3,instruct4,instruct5,instruct6,instruct7

# WORDLISTS FROM Greenwald et al. 1998
filenameI = "pools/insects.txt"
filenameF = "pools/flowers.txt"
filenameP = "pools/positives.txt"
filenameN = "pools/negatives.txt"

I = open(filenameI)
F = open(filenameF)
P = open(filenameP)
N = open(filenameN)

stimListI = I.read().split('\n')
stimListF = F.read().split('\n')
stimListP = P.read().split('\n')
stimListN = N.read().split('\n')

#pop off the trailing line
stimListI.pop(len(stimListI)-1)
stimListF.pop(len(stimListF)-1)
stimListP.pop(len(stimListP)-1)
stimListN.pop(len(stimListN)-1)

def gen_blocks(type):

    sampI = rm.sample(stimListI, 10)
    sampF = rm.sample(stimListF, 10)
    sampP = rm.sample(stimListP, 10)
    sampN = rm.sample(stimListN, 10)

    #Generate the blocks
    list1 = {"left_word":"flower", "right_word":"insect", "instruct":instruct1,
             "words":([{"correct":"right", "center_word":I} for I in sampI] +
                      [{"correct":"left", "center_word":F} for F in sampF])}

    list2 = {"left_word":"positive", "right_word":"negative", "instruct":instruct2,
             "words":([{"correct":"left", "center_word":P} for P in sampP] +
                      [{"correct":"right", "center_word":N} for N in sampN])}

```

```
    list3 = {"left_word":"flower positive", "right_word":"insect negative", "instruct":instruct3,
             "words":([{"correct":"right", "center_word":I} for I in rm.sample(sampI[:], 5)] +
                      [{"correct":"left", "center_word":F} for F in rm.sample(sampF[:], 5)] +
                      [{"correct":"left", "center_word":P} for P in rm.sample(sampP[:], 5)] +
                      [{"correct":"right", "center_word":N} for N in rm.sample(sampN[:], 5)])}

    list4 = {"left_word":"flower positive", "right_word":"insect negative", "instruct":instruct4,
             "words":([{"correct":"right", "center_word":I} for I in sampI] +
                      [{"correct":"left", "center_word":F} for F in sampF] +
                      [{"correct":"left", "center_word":P} for P in sampP] +
                      [{"correct":"right", "center_word":N} for N in sampN])}

    list5 = {"left_word":"insect", "right_word":"flower", "instruct":instruct5,
             "words":[{"correct":"left", "center_word":I} for I in sampI] + [{"correct":"right", "cen

    list6 = {"left_word":"insect positive", "right_word":"flower negative", "instruct":instruct6,
             "words":([{"correct":"left", "center_word":I} for I in rm.sample(sampI[:], 5)] +
                      [{"correct":"right", "center_word":F} for F in rm.sample(sampF[:], 5)] +
                      [{"correct":"left", "center_word":P} for P in rm.sample(sampP[:], 5)] +
                      [{"correct":"right", "center_word":N} for N in rm.sample(sampN[:], 5)])}

    list7 = {"left_word":"insect positive", "right_word":"flower negative", "instruct":instruct7,
             "words":([{"correct":"left", "center_word":I} for I in sampI] +
                      [{"correct":"right", "center_word":F} for F in sampF] +
                      [{"correct":"left", "center_word":P} for P in sampP] +
                      [{"correct":"right", "center_word":N} for N in sampN])}
    rm.shuffle(list1['words'])
    rm.shuffle(list2['words'])
    rm.shuffle(list3['words'])
    rm.shuffle(list4['words'])
    rm.shuffle(list5['words'])
    rm.shuffle(list6['words'])
    rm.shuffle(list7['words'])

    #If type 1, then do critical compatible lists
    if type == 1:
        return [list1, list2, list3, list4, list5, list6, list7]
    #if type 2, then do critical incompatible lists
    else:
        return [list5, list2, list6, list7, list1, list3, list4]
#GenBlocks
BLOCKS = gen_blocks(1)
```

CITATION

```
Greenwald, Anthony G.; McGhee, Debbie E.; Schwartz, Jordan L.K. (1998), "Measuring Individual Differe
```

```
Yu, Wang, Wang (2012), "Beyond Reaction Times: Incorporating Mouse-Tracking Measures into the Implici
```

Data accessing and Processing

Saving your Data into SLOG Files

In SMILE, each state will produce what is called a *.slog* file by default. This file is a specially compressed file composed of all the important data associated with the state. It not only logs every parameter, but also all of the variables listed in the *Logged Attributes* section of the docstrings. In most cases, every state will save out 2 rows of data to the *.slog* file. The first row is the field names, and the second row will be the data for each of those fields.

The kind of state that write multiple lines out to its associated *.slog* file is the `Log` state. Wherever the class is inserted into the experiment, the class will log the values of all of the keywords/argument pairs it is passed. If the *Log* exists within a loop, it will write out the values of the keyword/argument pairs during each iteration of the loop during experimental runtime. In this case, the *.slog* file will have the first row be the keywords, and the subsequent rows be all of the data for each *Log* during each iteration of the loop.

Note: Every instance of the Log state in the experiment will save to a separate file.

Below is an example of a *Log* state.

```
with Loop(10) as trial:
    lb = Label(text=trial.i, duration=2)
    Log(trial.current,
        name="looping_log",
        label_appear_time=lb.appear_time['time'])
```

This example will save 11 rows into a *.slog* file. If **trial.current** is the first argument for *Log*, then it will save out all of the information about the looping variable out in different columns.

A Record state will record all of the references given. It will write a line to the *.slog* file every time one of the references changes. It will also log the time at which the given reference changed.

Reading your SLOG files in python

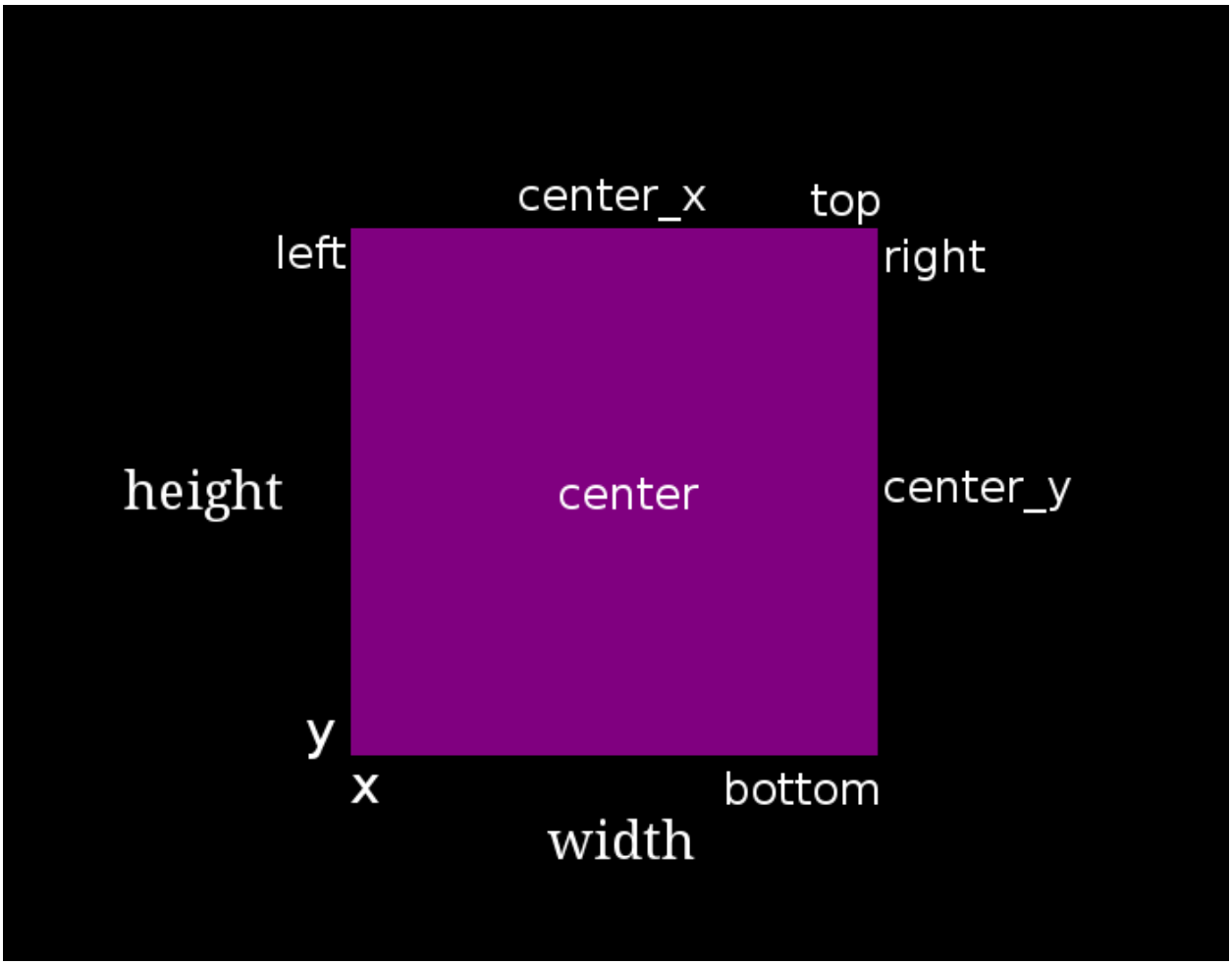
In order to slog through data, one of two things are first needed to be completed. The first is to pull the data into python by using the `Log` method called `log2dl()`. This method converts the *.slog* file to a list of dictionaries so that you can perform any pythonic functions on it in order to analyze the data. *log2dl* has one required parameter, *log_filename*, which should be a string that starts out *log_* and ends with a user chosen *name* parameter of the *Log* in the user's experiment.

If there are multiple files with the same name, they have trailing *_#* in the filename. *log2dl* will pull all of the files with the same base name, and concatenate them into a single list of dictionaries.

The other way data can be access is by converting all of the *.slog* files to *.csv* files. This can be accomplished by running the `Log2csv()` method. This method will take two parameters, *log_filename* and *csv_filename*. *log_filename* works the same way as in *log2dl*, where a string that is *log_* plus the name which was provided in the *name* parameter of the *Log* is passed. If no *csv_filename* is given, then it will be saved as the same name as the *log_filename* plus *.csv*. From there, one can use their preferred method of data analysis.

Advanced SMILEing

Screen Placement of Visual States



In SMILE, any state that displays something to the screen is known as a `VisualState`. These states share the ability to set their size, position, and relative position to each other. Every visual state has the following basic attributes, and all of the following attributes can be passed into the initialization of the visual states in your code:

- width
- height
- x
- y

Now, imagine a scenario where one would want to place a `Label` 400 pixels above a `TextInput`, which is 200 pixels to the left of the bottom right hand corner of the screen. Hard calculations of those numbers by hand or relativistic positioning attributes could be employed to yield the answer.

By utilizing the relative position attributes, the **VisualStates** can be initialized to the left or right, above or below, of each other. An example of this is as follows:

```
1 from smile.common import *
2
```

```
exp = Experiment()

with Parallel():
    lb1 = Label(text="I AM NEAR THE BOTTOM", right=exp.screen.right - 200,
                bottom=exp.screen.bottom, duration=5)
    lb2 = Label(text="I AM ABOVE THE OTHER LABEL", right=lb1.right,
                bottom=lb1.top + 400, duration=5)

exp.run()
```

In the above example, the *right* attribute of the visual states is used as both initialization parameters and attributes. This can be accessed from one state and applied to the next. We also used the attribute *bottom* which works the exact same way. The following are a list of all the attributes that are in terms of x, y, width, and height:

- bottom : y
- top : y + height
- left : x
- right : x + width
- center_x : (x + width) / 2
- center_y : (y + height) / 2

Multiple of these can be combined together to access a tuple value that contains both pieces of information. These combined attributes are listed below in terms of x, y, width, and height:

- center : ((x + width) / 2, (y + height) / 2)
- center_top : ((x + width) / 2, y + height)
- center_bottom : ((x + width) / 2, y)
- left_center : (x, (y + height) / 2)
- left_bottom : (x, y)
- left_top : (x, y + height)
- right_center : (x + width, (y + height) / 2)
- right_bottom : (x + width, y)
- right_top : (x + width, y + height)

Extending Smile

There may be cases where SMILE lacks functionality needed to run an experiment properly. Several different methods can be employed to **extend** SMILE's functionality. The first method is **Subroutine**, which is a section of state machine code that can be run at several different points in an experiment, similar to a function. The second is referred to as **Wrapping Widgets**. Any widgets written and defined in **Kivy** can be wrapped into a SMILE `WidgetState`.

Defining Subroutines

In SMILE, there exists special states called `Subroutines`. **Subroutines** are special states that contain small chunks of state machine code that the main experiment will need to run over and over again. Like a function, a **Subroutine** is defined with the python *def* followed by the name of the Subroutine. In SMILE, it is proper practice to name any state with the first letter of every word a capital letter.

Note: Please note that Subroutines should only be used as self contained snipits of state-machine. Only write a subroutine if the section of state-machine you are trying to replicate would rely only on the parameters passed into it. You should never try to change the value of a parameter inside the Subroutine from outside the Subroutine. However, you have read-only access to any variable set using the **self** reference explained below. If you would like to have access to the height of a Label inside your subroutine outside your subroutine, you must set **self** variable to the Height of your Label during Experimental Build Time.

The following is an example on how to define a **Subroutine** that displays a `Label` that will display a number that counts up from a passed in minimum number.

In the subroutine file (*test_sub.py*), first import all of SMILE's common states:

```
from smile.common import *
```

Warning: Be advised, the above line does not always give every necessary state for an experiment, just the States that are available on every platform.

Next, the definition line needs to be written for the subroutine:

```
@Subroutine
def CountUpFrom(self, minVal):
```

First, notice the *@Subroutine*. This allows *CountUpFrom* to be a subclass of *Subroutine*, the general subroutine state.

Note: Please note the *self* as the first argument passed into a subroutine. If self is not passed, SMILE will throw an error. Please remember to pass in *self* as the first parameter when defining a subroutine.

Now we can write state machine code for the **Subroutine**:

```
from smile.common import *
@Subroutine
def CountUpFrom(self, minVal):

    # Initialize counter, Creates a Set state
    # and sets the variable at Experimental Runtime.
    # After this line, self.counter is a reference object
    # that can be reference anywhere else in this subroutine.
    self.counter = minVal
    # Define the Loop, loop 100 times
    with Loop(100):
        # Apply the plus-equals operator to
        # self.counter to add 5
        self.counter += 5
        # Display the reference self.counter in
        # string form. Ref(str, self.counter) is required
        # to apply the str() function to self.counter during
        # Experimental Runtime instead of Buildtime
        Label(text=Ref(str,self.counter), duration=.2)
```

Warning: When writting a Subroutine, you can only use SMILE States. A Subroutine will only run any general pythonic code ONCE when the Subroutine is first built during Experimental Build Time. It is best practice to only use SMILE states, sets, and gets during in a Subroutine. If you need to run some kind of complex function in order to run your subroutine, use the **Func** state to run a function during Experimental Run Time.

Notice *self.counter*, it creates a `Set` state that will set a new attribute to the **Subroutine** called *counter* and will initialize it to *minVal* during **:ref:'Experimental Runtime <run_build_time>'_**.

Anything initialized with the *self.* will be able to be accessed from outside of the **Subroutine**. If the above Subroutine is used as an example, the **Subroutine** as *cup = CountUpFrom()* can be initialized and *cup.counter* can be called to get at the value of the counter.

The following is an example of calling this subroutine during an actual experiment:

```
from smile.common import *

from countup import CountUpFrom

exp = Experiment()

# Just like writing any other state declaration
cuf = CountUpFrom(10)
# Print out the value of the counter in CountUpFrom
# To the command line
Debug(name="Count Up Stuff", end_counter=cuf.counter)

exp.run()
```

Wrapping Kivy Widgets

Currently, most of the visual states in SMILE are *wrapped* Kivy widgets. `Rectangle`, `Image`, and `Video` are all examples of Kivy widgets that were wrapped in the *video.py* code and turned into `WidgetStates`.

if there is a desired function that SMILE can't performed using pre-written states, and the function cannot be created by writing a `Subroutine`, Kivy widgets can be written to achieve this functionality. To write a Kivy widget for SMILE, the knowledge of the SMILE backend and Kivy is needed. This section is only for those who want to write their own widgets!

The *My First Widget<https://kivy.org/docs/tutorials/firstwidget.html>* gives a thorough examination on how to create a very basic Kivy widget and display it on a Kivy app. This also provides sufficient start on how to create a Kivy widget.

For following example, dotbox.py will be examined. A program was written to produce tiny dots on the screen in an area. The most efficient way accomplish this is through the creation of a Kivy widget.

Here is the definition of our *DotBox*:

```
@WidgetState.wrap
class DotBox(Widget):
    """Display a box filled with random square dots.

    Parameters
    ----------
    num_dots : integer
        Number of dots to draw
    pointsize : integer
        Radius of dot (see *Point*)
    color : tuple or string
        Color of dots
    backcolor : tuple or string
        Color of background rectangle

    """
```

```
    # Define the widget Parameters for Kivy
    color = ListProperty([1, 1, 1, 1])
    backcolor = ListProperty([0, 0, 0, 0])
    num_dots = NumericProperty(10)
    pointsize = NumericProperty(5)
```

In *DotBox* several different parameters are needed to be passed into the *__init__* method in order to create different kinds of DotBoxes.

- Color : A list of float values that represent the RGBA of the dots
- backcolor : A list of float values that represent the RGBA of the background
- num_dots : The number of random dots to generate
- pointsize : How big to draw the dots, pointsize by pointsize squares in pixels

Next, the '__init__' method is declared for our 'DotBox' widget:

```
def __init__(self, **kwargs):
    super(type(self), self).__init__(**kwargs)

    # Initialize variables for Kivy
    self._color = None

    self._backcolor = None

    self._points = None

    # Bind the variables to the widget
    self.bind(color=self._update_color,
              backcolor=self._update_backcolor,
              pos=self._update,
              size=self._update,
              num_dots=self._update_locs)

    # Call update_locs() to initialize the
    # point locations
    self._update_locs()
```

The *.bind()* method will bind each different attribute of the dot box to a method callback that might want to run if any of those attributes change. An example of this is if, in SMILE, an `UpdateWidget` state is created where it updates a **DotBox** attribute, e.g. *num_dots* attribute. The attribute change will cause Kivy to callback the corresponding function attached with *.bind()*. Now the functions can be defined:

```
# Update self._color.rgba
def _update_color(self, *pargs):
    self._color.rgba = self.color

# Update self._backcolor.rgba
def _update_backcolor(self, *pargs):
    self._backcolor.rgba = self.backcolor

# Update the locations of the dots, then
# Call self._update() to redraw
def _update_locs(self, *pargs):
    self._locs = [random.random()
                  for i in xrange(int(self.num_dots)*2)]
    self._update()
```

```
# Update the size of all of the dots
def _update_pointsize(self, *pargs):
    self._points.pointsize = self.pointsize

# Draw the points onto the Kivy Canvas
def _update(self, *pargs):
    # calc new point locations
    bases = (self.x+self.pointsize, self.y+self.pointsize)
    scales = (self.width-(self.pointsize*2),
              self.height-(self.pointsize*2))
    points = [bases[i % 2]+scales[i % 2]*loc
              for i, loc in enumerate(self._locs)]

    # draw them
    self.canvas.clear()
    with self.canvas:
        # set the back color
        self._backcolor = Color(*self.backcolor)
        # draw the background
        Rectangle(size=self.size,
                  pos=self.pos)
        # set the color
        self._color = Color(*self.color)
        # draw the points
        self._points = Point(points=points, pointsize=self.pointsize)
```

Any visual widget created in Kivy will require some kind of drawing to the canvas. In the above example, the line *with self.canvas* was used to define the area in which calls to the graphics portion of Kivy were made, *kivy.graphics*. The color of what to be drawn was set, then it was drawn. For example, *Color()* sets the draw color, then *Rectangle()* tells **kivy.graphics** to draw a rectangle of that color to the screen.

Since this Widget defined in Kivy will be wrapped with a **WidgetState**, it can be assumed that this widget will have access to arguments like *self.pos, self.size*, and obviously arguments like *self.x, self.y, self.width, self.height*.

dotbox.py in Full

```
@WidgetState.wrap
class DotBox(Widget):
    """Display a box filled with random square dots.

    Parameters
    ----------
    num_dots : integer
        Number of dots to draw
    pointsize : integer
        Radius of dot (see *Point*)
    color : tuple or string
        Color of dots
    backcolor : tuple or string
        Color of background rectangle

    """
    color = ListProperty([1, 1, 1, 1])
    backcolor = ListProperty([0, 0, 0, 0])
    num_dots = NumericProperty(10)
    pointsize = NumericProperty(5)
```

```
    def __init__(self, **kwargs):
        super(type(self), self).__init__(**kwargs)

        self._color = None
        self._backcolor = None
        self._points = None

        self.bind(color=self._update_color,
                  backcolor=self._update_backcolor,
                  pos=self._update,
                  size=self._update,
                  num_dots=self._update_locs)
        self._update_locs()

    def _update_color(self, *pargs):
        self._color.rgba = self.color

    def _update_backcolor(self, *pargs):
        self._backcolor.rgba = self.backcolor

    def _update_locs(self, *pargs):
        self._locs = [random.random()
                      for i in xrange(int(self.num_dots)*2)]
        self._update()

    def _update_pointsize(self, *pargs):
        self._points.pointsize = self.pointsize

    def _update(self, *pargs):
        # calc new point locations
        bases = (self.x+self.pointsize, self.y+self.pointsize)
        scales = (self.width-(self.pointsize*2),
                  self.height-(self.pointsize*2))
        points = [bases[i % 2]+scales[i % 2]*loc
                  for i, loc in enumerate(self._locs)]

        # draw them
        self.canvas.clear()
        with self.canvas:
            # set the back color
            self._backcolor = Color(*self.backcolor)

            # draw the background
            Rectangle(size=self.size,
                      pos=self.pos)

            # set the color
            self._color = Color(*self.color)

            # draw the points
            self._points = Point(points=points, pointsize=self.pointsize)
```

Setting a variable in RT

Like it is stated in *Build Time VS Run Time*, in order to set a variable in SMILE during **RT**, the *exp.variable_name* syntax must be used. In this section, the results of calling 'exp.variable_name' in SMILE will be examined.

The following is a sample experiment where *exp.display_me* is set to a string:

```
from smile.common import *

exp = Experiment()

exp.display_me = "LETS DISPLAY THIS SECRET MESSAGE"
Label(text=exp.display_me)

exp.run()
```

This is a very simple experiment. It must be understood that *exp.display_me = "LETS DISPLAY THIS SECRET MESSAGE"* creates a `Set` state. A **Set** state takes a string *var_name* that refers to a variable in an **Experiment** or to a newly created variable, and a *value* that refers to the value that the variable is assigned to take on. The important takeaway is that 'value' can be referenced to a value. If 'value' is a reference, it will be evaluated during **RT**. Below is an example of what the experiment would look like if the 3rd line is changed:

```
from smile.common import *

exp = Experiment()

Set(var_name="display_me", value="LETS DISPLAY THIS SECRET MESSAGE")
Label(text=exp.display_me)

exp.run()
```

Both sample experiments run the exact same way, but the only difference is how the code looks to the end user. The Set state is untimed, so it changes the value of the variable immediately at enter. For more information look at the docstring for `Set` and the code behind the **smile.experiment.Experiment.set_var()** method.

Performing Operations and Functions in RT

Until this point, new methods that run during **RT** have not run correctly. In this section, examining why this happens and correcting this issue will be discussed.

Since every SMILE experiment is separated into **BT** and **RT**, any calls to functions or methods without using the proper SMILE syntax will run in **BT** and not **RT**. In order to run a function or method, a `Ref` or a `Func` is needed to be used. As stated in *The Reference Section* of the state machine document, a **Ref** is a delayed function call.

When it is desired to pass in the return value of a function to a SMILE state as a parameter, it is appropriate use Ref. The first parameter for a **Ref** call is always the function desired to run, and the other parameter to that function call are the rest of the parameters to the **Ref**.

Below is an example of a loop that displays the counter of the loop in a label on the center of the screen. Since the `Loop` counter is an integer, the integer must first be changed to a string. This can be performed by creating a **Ref** to call 'str()'.

```
with Loop(100) as lp:
    #This Ref is a delayed function call to str where
    #one of the parameters is a reference. Ref also
    #takes care of evaluating references.
    Label(text=Ref(str, lp.i), duration=0.2)
```

To run a function during RT the **Func** state is needed. **Func** creates a state that will not run the passed in function call until the previous state leaves. The following is an example of using a **Func** to generate the next set of stimulus for each iteration of a **Loop**. To access the return value of a method or function call, the *.result* attribute of the **Func** state must be accessed.

```
#Assume DisplayStim is a predefined Subroutine
#that displays a list of stimulus, and assume that
#gen_stim is a predefined function that generates
#that stimulus
with Loop(10) as lp:
    stim = Func(gen_stim, length=lp.i)
    DisplayStim(stim.result, duration=5)
```

Note: Remember that you can pass in keyword arguments AND regular arguments into both Func states and Ref calls.

Effective timing of KeyPress

In order to increase the effectiveness of a **KeyPress** state, you can set a *base_time* parameter. A **KeyPress** will calculate the reaction time, or *rt*, by subtracting the *base_time* from the *press_time*. If no *base_time* is passed in as a paramter to **KeyPress**, SMILE will set the *base_time* to the **KeyPresses** *start_time*.

When you want someone to press a button **immediately** after they see a stimulus, you need to set the *base_time* as the *appear_time['time']*. See an example of this below.

```
press = Label(text="Press NOW!")
with UntilDone():
    Wait(min_response_time)
    kp = KeyPress(base_time=press.appear_time['time'])
```

When you want a participant to press a button **immediately** after they see a stimulus disappear off the screen, you need to set the *base_time* as the *disappear_time['time']*. See an example of this below.

```
press = Label(ext="Press When I Disappear", duration=2.0)
Wait(until=press.disappear_time)
kp = KeyPress(base_time=press.disappear_time['time'])
```

Timing the Screen Refresh VS Timing Inputs

Before examining this section, it is important to understand how SMILE displays each frame of your experiment. SMILE runs on a two buffer system, where when a frame is being prepared, it is drawn to a *back buffer*. When everything is drawn and/or ready, the *back buffer* is flipped to the *front buffer*, then the back buffer is cleared to get ready for more drawing.

The following is a detailed example: an experiment wants to display a new `Label` onto the screen. The first thing SMILE does is draw the Label onto the back buffer, then calls for a **Blocking Flip**. A **Blocking Flip** is when SMILE waits for everything to be finished writing to the screen, then flips the next time it passes through the event loop if it is around the flip interval. Then SMILE flips into **NonBlocking Flip** Mode. In this mode, SMILE will try and flip the buffer as soon as anything changes. SMILE switches to this mode to allow Kivy to update the screen whenever it needs to. The other time in a Visual State's lifespan where SMILE calls for a **Blocking Flip** is when it disappears from the screen. SMILE uses **Blocking Flips** for the appearance and disappearance of a VisualState to accurately track the timing of those two events.

In SMILE, the end user can force the 2 different modes of updating the screen using `BlockingFlip` and `NonBlockingFlip`. They both are important, for they both grant the ability to prioritize different aspects of an experiment, *input* or *output*, when it comes to timing things as accurately as possible.

A **NonBlockingFlip** is used when the timing of visual stimulus isn't the most important. If SMILE is forced into this mode, timing of input can be made much more accurate, like mouse and keyboard. SMILE can be forced into NonBlockingFlips by putting this state in parallel with what is desired to run in NonBlockingFlip Mode.

The following is a mini example of such a **Parallel**:

```
with Parallel() as p:
    NonBlockingFlip()
    lb = Label(text="PRESS NOW!!!")
    with UntilDone():
        Wait(until=lb.appear_time)
        kp = KeyPress(base_time = lb.appear_time['time'])
```

A **BlockingFlip** is used when the timing of screen appearance takes priority over when the timing of inputs occur. Using this mode, the changes in *exp._last_flip* can be `Record`.

An example of this is as follows:

```
with Parallel():
    BlockingFlip()
    vd = Video(source="test_vid.mp4")
    Record(name="video_record", flip=exp._last_flip)
```

Information for SMILE Developers

Below will be several sections that better explain all of the intricacies of SMILE's backend. Look at this section only if you are interested in creating your own states, or better understanding how SMILE does what it does.

The States of a State

Every state in SMILE runs through 6 main function calls. These function calls are automatic and never need to be called by the end user, but it is important to understand what they do and when they do it to fully understand SMILE. These function calls are *__init__*, *.enter()*, *.start()*, *.end()*, *.leave()*, and *.finalize()*. Each of these calls happen at different parts of the experiment, and have different functions depending on the subclass.

.__init__ happens during **BT** and is the only one to happen at **BT**. This function usually sets up all of the references, processes some of the parameters, and knows what to do if a parameter is missing or wasn't passed in.

.enter() happens during **RT** and will be called after the previous state calls *.leave()*. This function will evaluate all of the parameters that were references, and set all the values of the remaining parameters. It will also schedule a start time for this state.

.start() is a class of function calls that, during **RT**, the state starts doing whatever makes it special. This function is not always called *.start()*. In the case of an `Image` state, *.start()* is replaced with *.appear()*. The *.start()* functions could do anything from showing an image to recording a keypress. After *.start()* this state will begin actually performing its main function.

Note: A *.start()* kind of call will only exist in an Action State (see below).

.end() is a class of function calls that, during **RT**, ends whatever makes the state special. In the case of an Image, *.end()* is replaced with *.disappear()*. After *.end()*, *.leave()* is available to be called.

Note: A *.end()* kind of call will only exist in an Action State (see below).

.leave() happens during **RT** and will be called whenever the duration of a state is over, or whenever the rules of a state says it should end. A special case for this is the *.cancel()* call. If a state should need to be ended early for whatever reason, the *Experiment* will call the state's *.cancel()* method and that method will setup an immediate call to both *.leave()* and *.finalize()*.

.finalize() happens during **RT** but not until after a state has left. This call usually happens whenever the clock has extra time, i.e. during a `Wait` state. This call will save out the logs, setup callbacks to the `ParentState` to tell it that this state has finished, and set *self.active* to false. This call is used to clean up the state sometime after the state has run *.leave()*.

The SMILE timing Algorithm

Write up coming soon.

Want to Contribute to SMILE?

SMILE has a GitHub page that, if you find an issue and fix it or want to add functionality to SMILE, you may make a pullrequest to. At GitWash you can find documents to better understand how to make use Git and how to make changes and update SMILE.

Seeking Help?

SMILE has a Google group where you can discuss SMILE with other people who are trying to learn how to use it, as well as see if anyone is having the same problems that you are. This group is located at smile-users.

SMILE also has a GitHub page where you can report any issues that you have.

SMILE package

smile.audio module

smile.clock module

smile.dag module

smile.experiment module

smile.freekey module

smile.keyboard module

smile.kivy_overrides module

smile.log module

smile.mouse module

smile.pulse module

smile.ref module

smile.state module

smile.utils module

smile.video module

Module contents

CHAPTER 3

Funding Sources