The Unlock Project Documentation

Release 0.1

B. Galbraith, S. Lorenz, D. Smith, J. Brumberg, F. Guenther

August 30, 2012

CONTENTS

Welcome to the Unlock Project! Last updated **|Today|**

We are a non-profit group at Boston University dedicated to creating and providing resources for un-conventional means of communication to individuals who have lost voluntary motor control. Specifically we are working to create brain-computer interfaces (BCIs) for individuals with Locked-in Syndrome (LIS). We have chosen to research non-invasive techniques to easily acquire brain signals, and allow BCI users to live more comfortably by interacting with other people and their environment.

Contents:

CHAPTER

ONE

OUR METHOD

Locked-in Syndrome (LIS) is a neurological condition characterized by the loss of most or all motor control while retaining cognitive functions. As a consequence, locked-in individuals are unable to interact physically with their environment and have limited or no means to communicate. Brain Computer Interfaces (BCIs) offer a method for locked-in patients to communicate with others and to manipulate their surroundings.

Brain-computer interfaces require three main components. The first is a brain function or brain wave to monitor and study. This is a brain process that can be controlled or manipulated by the user in order to use the BCI. Some examples of this are Sensorimotor Rhythms (SMR) and Event-related potentials (ERP). SMR is a signal created in the primary motor cortex during intended limb motion. An ERP, such as P300, is a signal created in the parietal lobe in response to very specific visual stimulus, surrounded by non-important ones. The signals that we control with our BCI are Steady-State Visually-Evoked Potentials (SSVEP). SSVEPs are brain waves of the occipital lobe that are created when a user attends to a flashing light at a given frequency. The SSVEP reaches the occipital lobe at the same frequency and first harmonic of the flashing stimulus. For our purposes, we have tested with flashing light-emitting diodes (LED) and flashing rectangles on a computer monitor

The second component of any BCI is a method of acquiring and decoding brain functions Brain activity can be acquired by either invasive or non-invasive means. Invasive means require the opening of the skull to implant electrodes on the surface, or into the surface of the brain. This method provides high spatial resolution, but requires surgery and adds the risk of infection and damage to the neural tissue. Non-invasive means such as Electro-encephalograpy (EEG) have lower spatial resolution, but excellent temporal resolution, and no risk of injury. EEG is used for this project to record SSVEP from the occipital lobe of the user. EEG is usually decoded using a Fast-Fourier Transform (FFT), although their are other methods for decoding such as Canonical Correlation Analysis (CCA), and Minimum Energy Combination (MCE).

The third component of a BCI is an application to use the decoded data once it is received by the computer. It needs to be useful to the person that is using the BCI, and it also needs to be simple enough that the user can maintain control over this un-conventional task. Our goal is to aid individuals with motor impairment, so our applications (apps) focus on ways to make the lives of these individuals better. Examples of our apps are a grid of phrases which can be navigated to select a specific phrase, a TV remote control which interface with an IR transceiver to use a television, and a control interface for a Mobile Robotic Platform for remote accessibility and interaction.

CHAPTER

TWO

HOW TO GET STARTED

2.1 Python

All of our software is coded in the Python 2.7 Programming language. We also have a third party libraries that we use for graphics and mathmatical operations. These are Pyglet, and SciPy and NumPy. If you do not already have Python, or these libraries you can download it from the following links

2.2 Python Package Installation

- Download and Install Python 2.7.2
  - Windows download here.
  - Max OS X 10.6-10.7 download here.
  - Mac OS X 10.3-10.6(32-bit) download here.
- Download and Install Pyglet Version 1.1.4
  - Windows download here.
  - Mac download here.
- Download and Install Numpy Version 1.6.1
  - Windows download here.
  - Mac download here.
- Download and Install Scipy Version 0.10.1
  - Windows download here.
  - Mac download here.

2.3 Git

Git and GitHub are tools for file sharing and version management. We store our repository on GitHub. If you do not already have Git, you can download it from the links below. We also provided a link to helpful GUI interfaces for Git.

- Windows download here. A nice Windows Git GUI is TortoiseGit.
- Mac download here. A nice Mac Git GUI is Tower.

So now that your machine is all set up, let’s take a look at what is in the repository.

CHAPTER

THREE

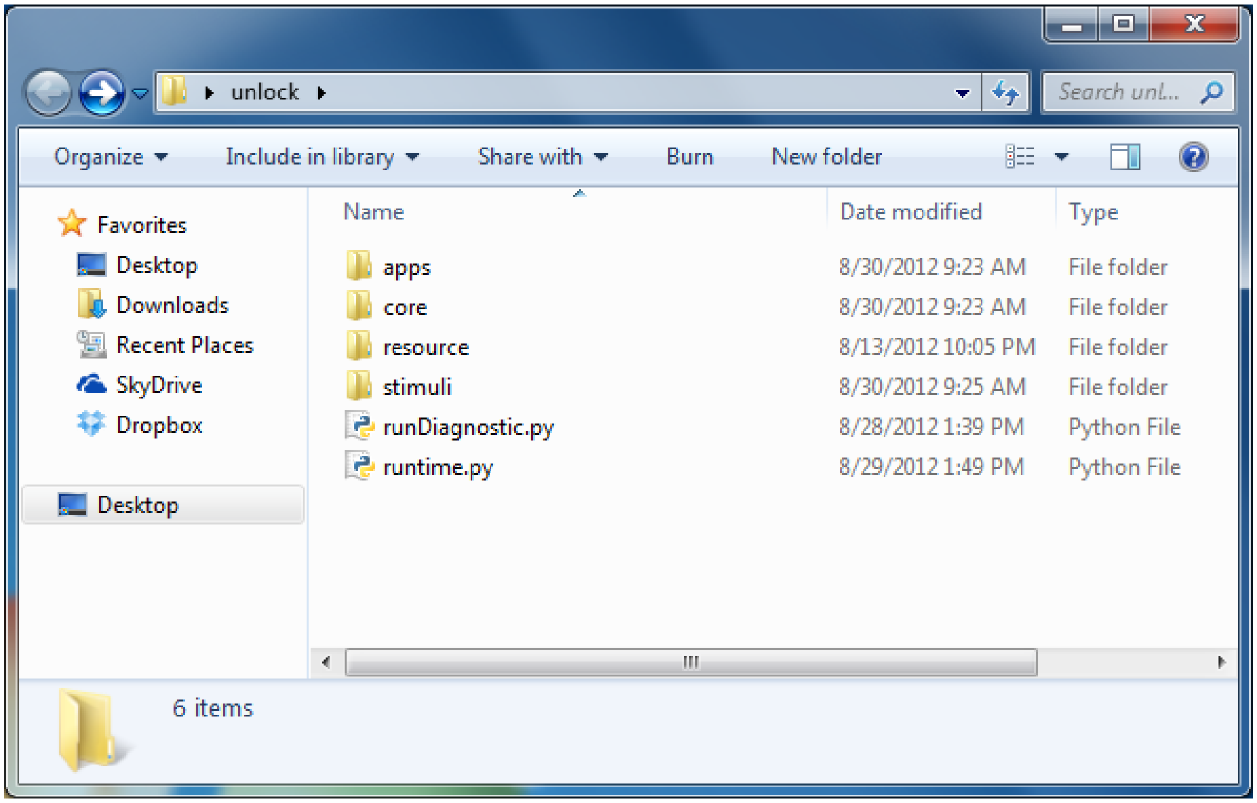
THE UNLOCK REPOSITORY

So now that you have downloaded Python, downloaded Git, and downloaded our Repository, what are you looking at?

The repository that we have given to you is specifically for working with the application Graphical user interface(GUI) Our hope is that it you will be able to use it to make your BCI based applications, or to carry out your own experiments to aid people with LIS, and other motor disabilities.

The next few page will give tutorials on how to use our existing interface, and build your own applications For now let's just take a quick look at the organization of the unlock repository.

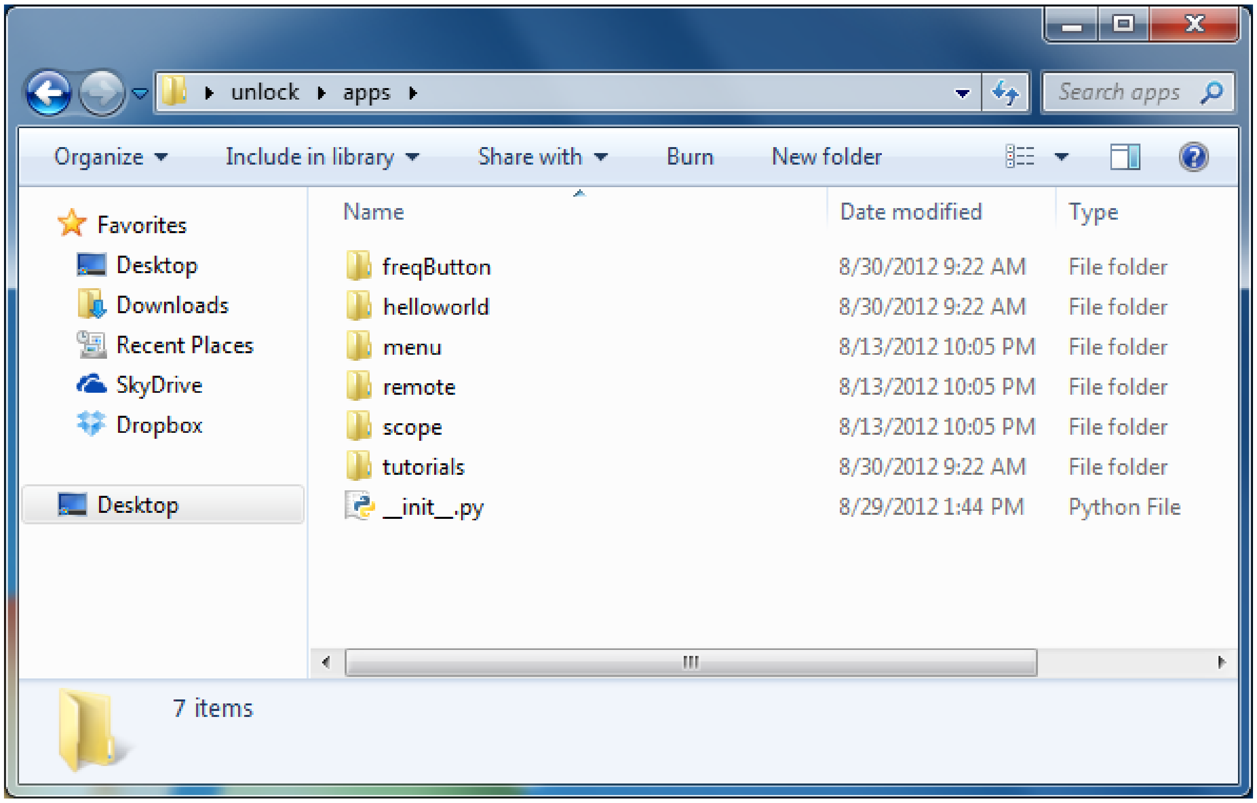
This is the top folder. It has four folders named apps, core, resource, and stimulus. It also has two scripts in the top folder which are runtime.py and runDiagnostic.py.



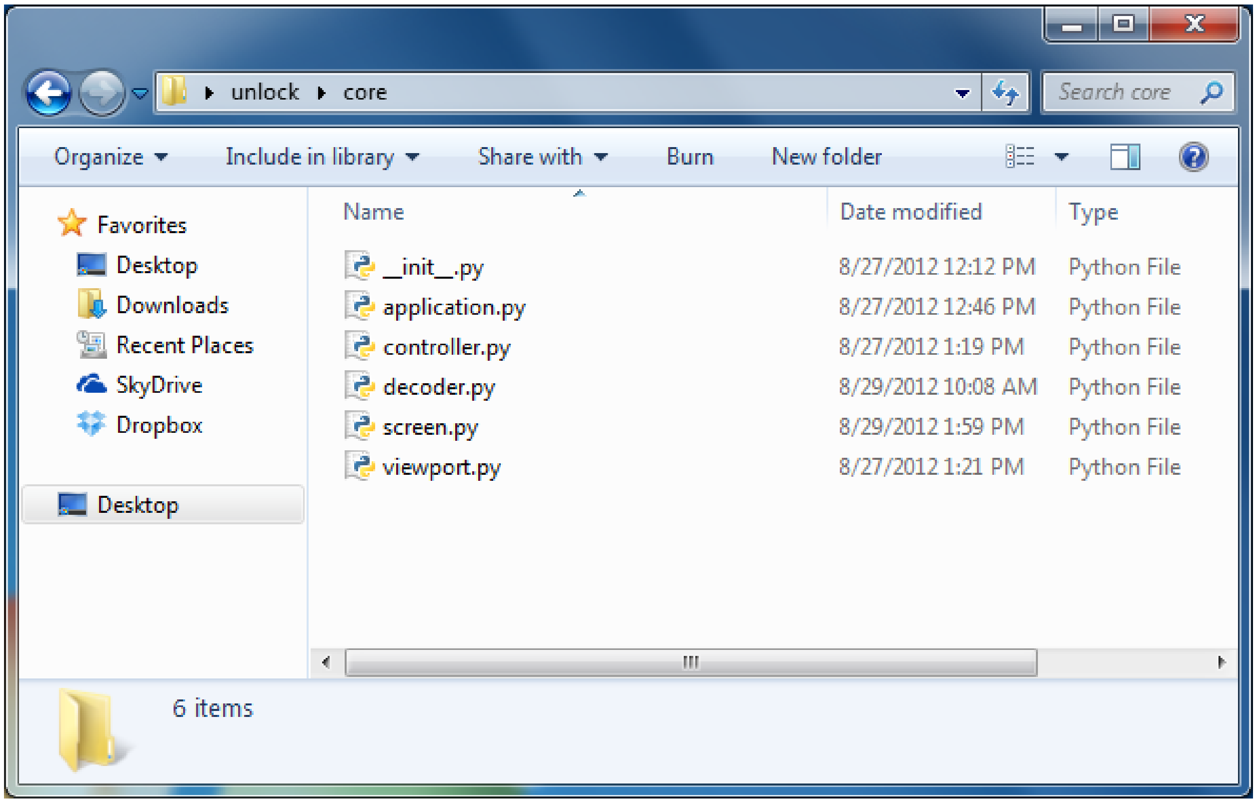
Name	Date modified	Type
apps	8/30/2012 9:23 AM	File folder
core	8/30/2012 9:23 AM	File folder
resource	8/13/2012 10:05 PM	File folder
stimuli	8/30/2012 9:25 AM	File folder
runDiagnostic.py	8/28/2012 1:39 PM	Python File
runtime.py	8/29/2012 1:49 PM	Python File

The apps folder is where we store the application modules.

The core folder is where we store modules for the under-laying frame of the app GUI.

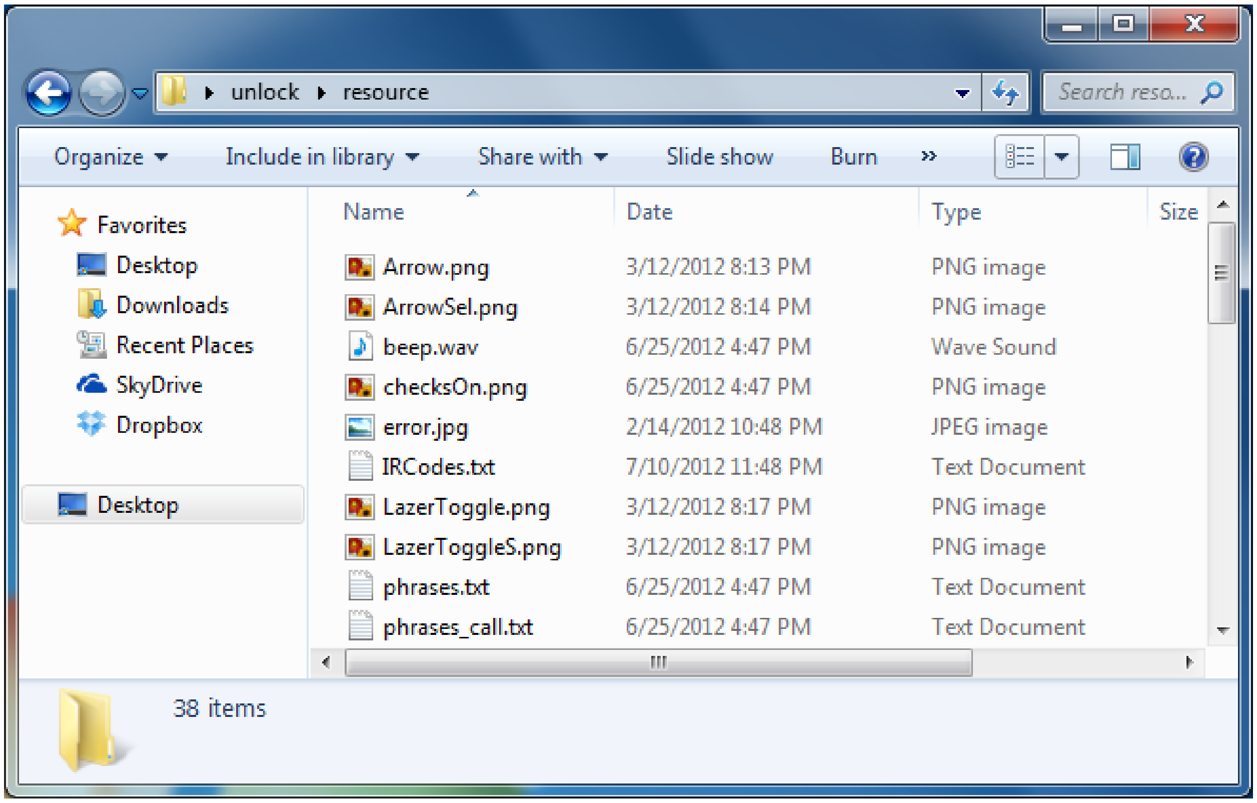


Name	Date modified	Type
freqButton	8/30/2012 9:22 AM	File folder
helloworld	8/30/2012 9:22 AM	File folder
menu	8/13/2012 10:05 PM	File folder
remote	8/13/2012 10:05 PM	File folder
scope	8/13/2012 10:05 PM	File folder
tutorials	8/30/2012 9:22 AM	File folder
__init__.py	8/29/2012 1:44 PM	Python File



Name	Date modified	Type
__init__.py	8/27/2012 12:12 PM	Python File
application.py	8/27/2012 12:46 PM	Python File
controller.py	8/27/2012 1:19 PM	Python File
decoder.py	8/29/2012 10:08 AM	Python File
screen.py	8/29/2012 1:59 PM	Python File
viewport.py	8/27/2012 1:21 PM	Python File

The resource folder stores relevant image and text files that are used by the apps.



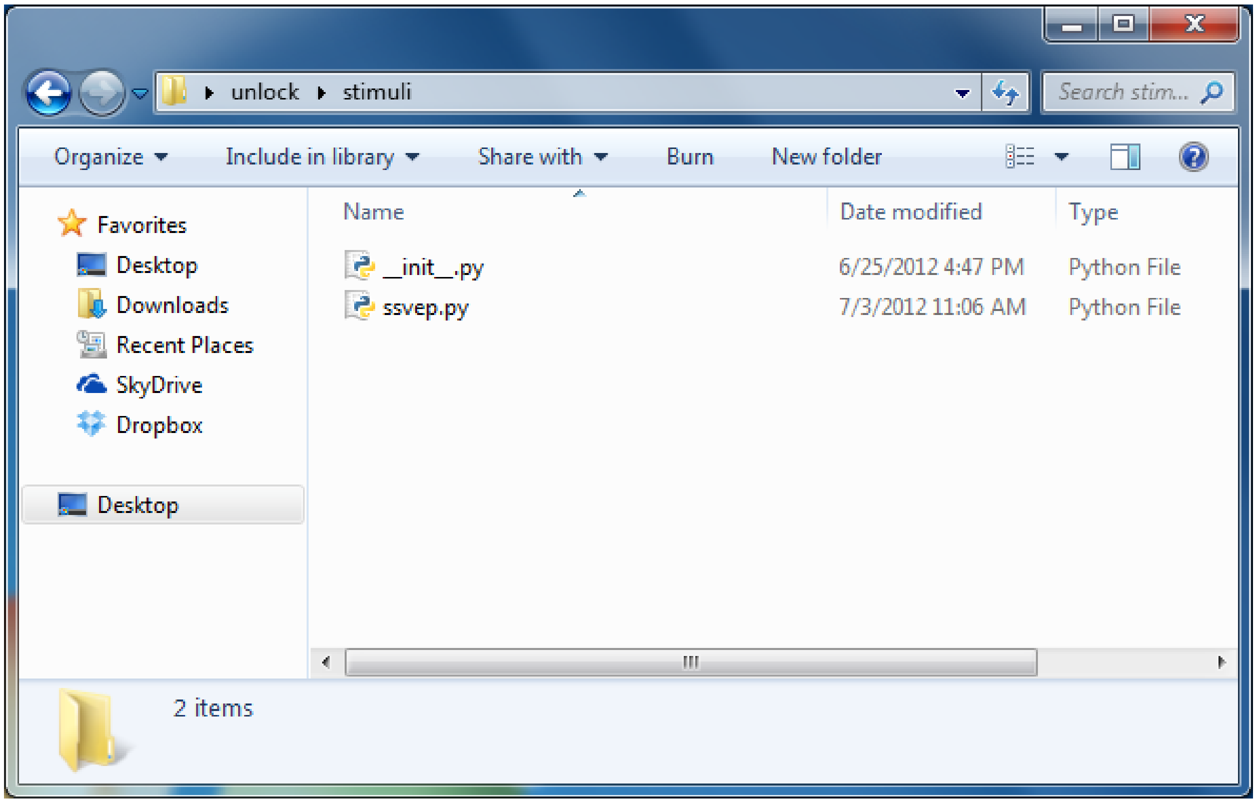
Name	Date	Type	Size
Arrow.png	3/12/2012 8:13 PM	PNG image	
ArrowSel.png	3/12/2012 8:14 PM	PNG image	
beep.wav	6/25/2012 4:47 PM	Wave Sound	
checksOn.png	6/25/2012 4:47 PM	PNG image	
error.jpg	2/14/2012 10:48 PM	JPEG image	
IRCodes.txt	7/10/2012 11:48 PM	Text Document	
LazerToggle.png	3/12/2012 8:17 PM	PNG image	
LazerToggleS.png	3/12/2012 8:17 PM	PNG image	
phrases.txt	6/25/2012 4:47 PM	Text Document	
phrases_call.txt	6/25/2012 4:47 PM	Text Document	

Finally, the stimuli folder holds the module for creating the SSVEP stimuli.

Runtime.py is the only application that you will need run to at any given time. If you have an experiment or demo that uses a specific set up of stimuli and applications, you could use a version of the runtime script for that specific purpose. runDiagnostic.py is one example of this. runDiagnostic.py is a version of runtime.py that is et up to check for a couple of things. It has a square stimulus in the bottom left corner, and the freqButton app running in the top left. This allows the speed of the stimulus to be sped up or slowed down. On the right there is a timescope and a frequency scope. These make sure that the EEG cap is recording data from the user, checks that the decoder is decoding data accurately, and can give a measure of how an user reacts to a given stimulus on the left hand side.

So now that you have had a tour of the repository. You can begin with the tutorial on how to use it.

If instead you would like to go straight to the Unlock API. Click on modules in the upper or lower right.



Name	Date modified	Type
__init__.py	6/25/2012 4:47 PM	Python File
ssvep.py	7/3/2012 11:06 AM	Python File

CHAPTER

FOUR

TUTORIAL: MAKING A BCI APPLICATION(BCI APP)

For now, let's focus on how to make and run an application. To start let's look at the format of an application script. Below is a script for a hello world application.

```
>>> from core import UnlockApplication
...
... class HelloWorld(UnlockApplication):
...     name = "Hello World"
...
...     def __init__(self, screen):
...         super(self.__class__, self).__init__(screen)
...
...         self.text = screen.drawText('Hello World!', screen.width / 2,
...                                     screen.height / 2)
...
...     def update(self, dt, decision, selection):
...         pass
```

The following code will display the words "Hello World" in the center of the given screen, as shown below.

You will also notice there is a screen refresh rate in the lower left corner. This is included for all apps.

Let's look at each individual part of this code.

First you need to import the UnlockApplication Module from application.py in the core folder.

```
>>> from core import UnlockApplication
...
```

This is a module that sets up the basic infrastructure of this application, and creates some easy functions, such as starting the application, closing it, and running the update cycle. You will not need to know what the UnlockApplication Module does. Just make sure that it is imported in the same manner shown, as well as calling it as part of the class. This is shown below.

Now this is your opportunity to name the class of your application. Maybe it will be BCIbrowser or BCIremote. For this example it is HelloWorld.

```
>>> from core import UnlockApplication
...
... class HelloWorld(UnlockApplication):
```

After writing the name of the class, beneath it you must assign a name member to the class. This reiterates the name of the app, and is required for the UnlockApplication module. Below this name is place to put other class members as you will see in other applications

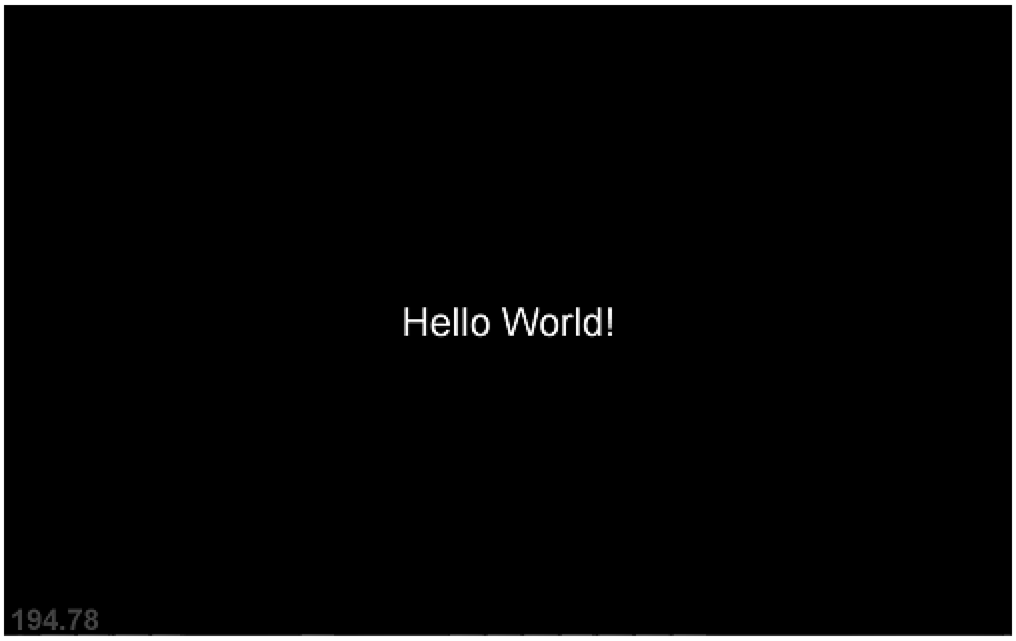


Figure 4.1: Hello World

```
>>> from core import UnlockApplication
...
... class HelloWorld(UnlockApplication):
...     name = "Hello World"
```

Next comes the methods. There are two methods that every application script is expected to have, the first of which is the __init__. For those of you who are not well versed in object oriented programming this is an opportunity to pick up on two important notes about OOP. The __init__ method is the only method that is run when an instance of a class is created. This means that when the HelloWorld is called, the only code that will be executed is code located in the __init__ method. All of the variables and functions of __init__ will run. If there are other methods of the class, they can be called by the __init__ method, but otherwise are run by being calling specifically from another piece of code.

The other important thing to learn about here is the keyword self. In python, and other languages, the word self relates to the class where it is contains. This allows the object (instance of the class), to be called without having the same name as the class, while still being able to call the class' variables and methods. You will notice that when creating global variables in the __init__ method, self is used before the variable name. It is also included by default as a input variable to most methods.

__init__ is where the preliminary variables are set, as well as defining the input variables. Every BCI application will want the screen variable passed to the __init__ method, and the variable declaration super(self.__class__, self).__init__(screen) as seen below.

```
>>> from core import UnlockApplication
...
... class HelloWorld(UnlockApplication):
...     name = "Hello World"
...
...     def __init__(self, screen):
...         super(self.__class__, self).__init__(screen)
...
...         self.text = screen.drawText('Hello World!', screen.width / 2,
...                                     screen.height / 2)
...
```

From here you can assign any other variables you would like. For the example of Hello World, we create text using the screen module. Screen is a basic command that we will go over in a later tutorial.

The second method that is required in the unlock application is the update method. This requires three variables to be passed to it: dt, decision, and selection. dt provides a time step to update in case your application needs it, decision provides a choice that the user made, such as an up, down, left, and right. Finally selection is a fifth choice such as choosing an item the cursor is on. For this example we are not updating the Hello World app so we will use pass on this method.

This is what the script looks all together.

```
>>> from core import UnlockApplication
...
... class HelloWorld(UnlockApplication):
...     name = "Hello World"
...
...     def __init__(self, screen):
...         super(self.__class__, self).__init__(screen)
...
...         self.text = screen.drawText('Hello World!', screen.width / 2,
...                                     screen.height / 2)
...
...     def update(self, dt, decision, selection):
...         pass
```

At this point you can add whatever other methods you want or need for your app.

Now this class will not work on its own. It requires two more scripts to run. The Next Tutorial on runtime.py will discuss this further.

CHAPTER

FIVE

TUTORIAL: USING RUNTIME.PY

As mentioned, each app, as well as most of the scripts of the Unlock Repository, is located within folders for better organization. Runtime.py is the exception. It sits outside of the lower folders in the top folder. This is the only script you should ever have to run. This is the script that collects your data, processes the recordings, and displays your apps. Of course, it does not do all these things by itself; it calls other scripts to do the work for it. This program is also good as a test bench for your applications because it will set up the whole structure of the program, every time you run. Let's look at a basic version of the script for running an app:

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
...
...     full = Screen(0,0,window.width, window.height)
...
...     app1 = HelloWorld(full)
...
...     return [app1]
...
... if __name__ == '__main__':
...     from core import viewport
...     viewport.controller.apps = get_apps(viewport.window)
...     viewport.start()
```

It starts out by importing Screen.

```
>>> from core import Screen
...
```

As I mentioned, this is a script that will be discussed in a future tutorial. In the meantime think of it as a script that defines the conditions of the computer screen that you are using.

It also imports an app from the apps folder

```
>>> from core import Screen
... from apps import HelloWorld
```

Apps refers to the repository folder that contains BCI apps. In this case we are importing HelloWorld. HelloWorld is the name of the class of the app.

Warning: Capitalization is important.

Next comes the method get_apps.

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
```

This method is given the window size as a variable, and it returns the apps that we want to display on the screen. To do this we will set a portion of the screen named full.

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
...
...     full = Screen(0,0,window.width, window.height)
```

Notice how we used the Screen class to create the screen partition.

Now that we have the screen size set up, we can send it to our app. We only have one app. So we write it like this:

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
...
...     full = Screen(0,0,window.width, window.height)
...
...     app1 = HelloWorld(full)
```

And then this after it:

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
...
...     full = Screen(0,0,window.width, window.height)
...
...     app1 = HelloWorld(full)
...
...     return [app1]
```

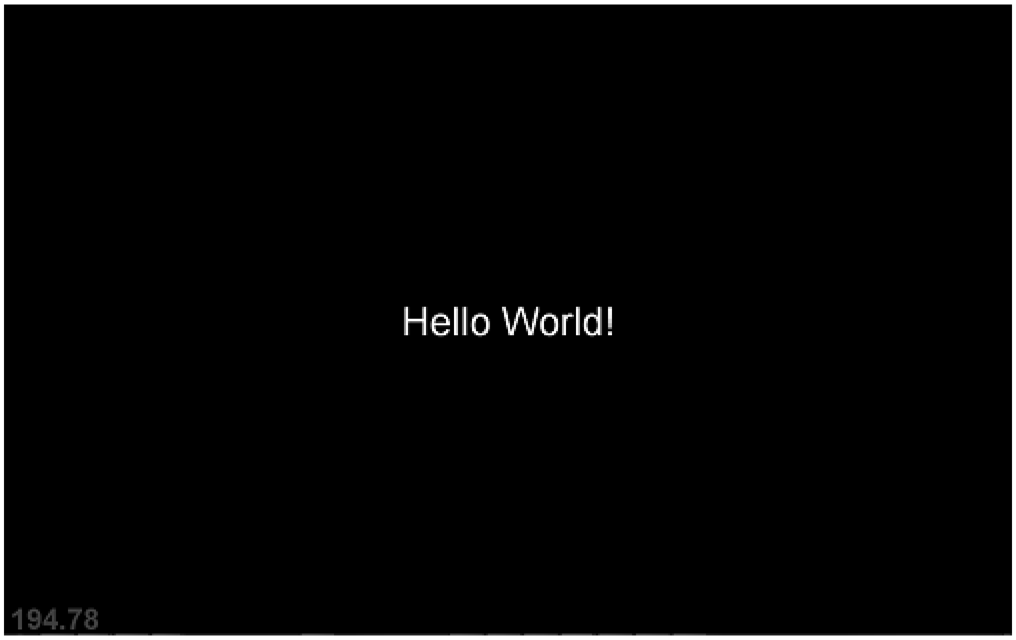
Using the return function allows the names of the apps to be given to the method.

Now the get_apps method is just that; only a method. We need a command to run get_apps This command will be placed at the bottom of the script

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
...
...     full = Screen(0,0,window.width, window.height)
...
...     app1 = HelloWorld(full)
...
...     return [app1]
...
... if __name__ == '__main__':
...     from core import viewport
...     viewport.controller.apps = get_apps(viewport.window)
...     viewport.start()
```

You will notice we are now importing a new module names viewport. For the most part you won't need to know what viewport does. It and another module named controller handle the underworkings of the app process. In fact you should not need to touch this portion ever. The only important thing you need to know about viewport is that ir requires the Pyglet Graphics Library. If you do not already have it, you should download Pyglet, and install it. It also here where viewport.start is run to start the program.

This is what the runtime.py script looks like all together, and when all said and done, it should look like this.


Hello World!

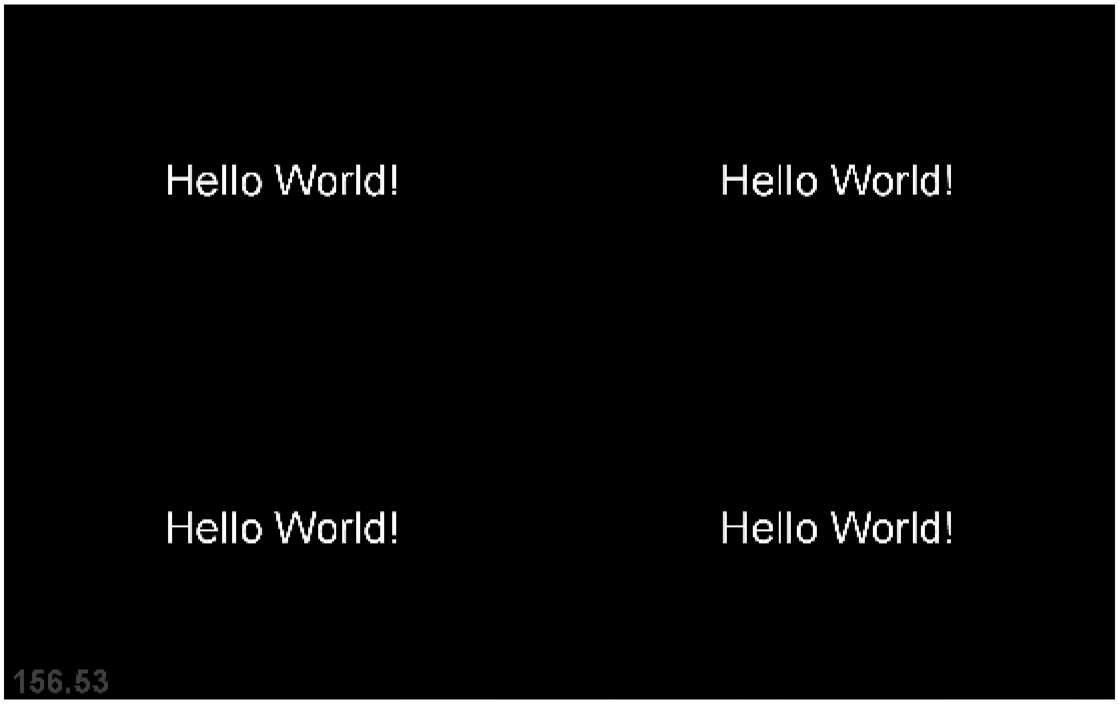
194.78

Let's say though, that you don't just want one app, but four apps running at the same time. We would need to partition the screen differently. We would also need to add new apps to run. To do this, we would change the script like this:

```
>>> from core import Screen
... from apps import HelloWorld
...
... def get_apps(window):
...
...     w2 = window.width / 2
...     h2 = window.height / 2
...     full = Screen(0,0,window.width, window.height)
...     bottom_left = Screen(0, 0, w2, h2)
...     bottom_right = Screen(w2, 0, w2, h2)
...     top_left = Screen(0, h2, w2, h2)
...     top_right = Screen(w2, h2, w2, h2)
...
...
...     app1 = HelloWorld(bottom_left)
...
...     app2 = HelloWorld(bottom_right)
...
...     app3 = HelloWorld(top_left)
...
...     app4 = HelloWorld(top_right)
```

```
...
...     return [app1, app2, app3, app4]
...
... if __name__ == '__main__':
...     from core import viewport
...     viewport.controller.apps = get_apps(viewport.window)
...     viewport.start()
```

Which would create this:



Now there is one more step for making this app appear. It is a quick fix, but an important one. It deals with a file called __init__.py.

CHAPTER

SIX

TUTORIAL: __INIT__.PY

If you have looked in the containing folders of the unlock folder you will see the file names of the apps, but you will also see a filename that is repeated many times over.

This file is __init__.py

It serves one very important function, which is to allow the developer to organize their scripts in containing folders for ease of use, access and storage. The __init__.py file has the exact same purpose as the __init__ method of the class we looked at in the App tutorial. Whatever is contained within the script of __init__ will run when it is called. In this case, whenever a folder containing a __init__.py file is imported, the commands of that __init__.py script are run.

Lets look at an example. Here is the top portion of the runtime.py script that we made in the last tutorial

```
>>> from core import Screen
... from apps import HelloWorld
...
```

These are two of the import functions used in the script. The first is importing the Screen class from the core folder. The second is importing the HelloWorld class from the apps folder. What does this actually mean? It means runtime.py is going to look at the folder it is stored in (unlock project top folder) and will look for an item named core, and an item named apps.

If it finds a file named apps, it will look in the file for a class named HelloWorld. In this case it will find the folder named apps, open it, and will look for a class named HelloWorld.

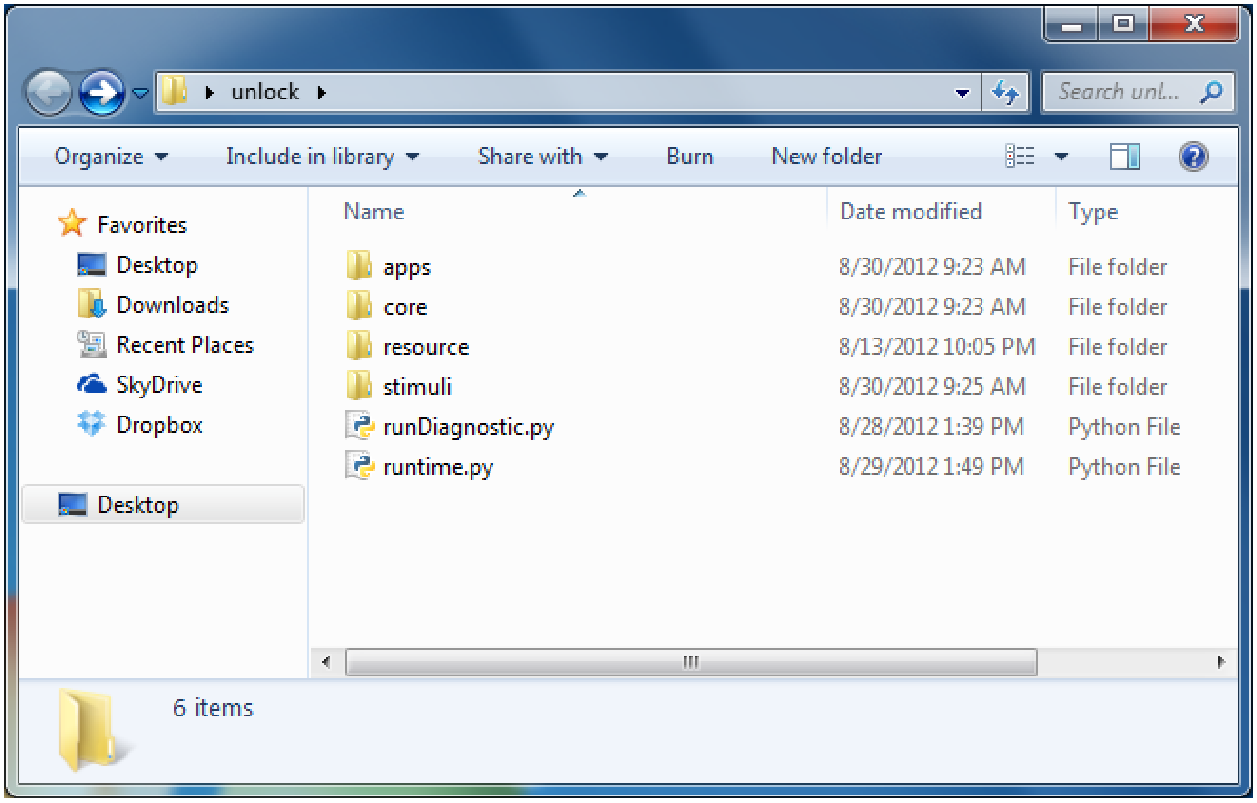
When the folder is opened the __init__.py script will run. If the HelloWorld class is in there, the file will import. However, what you will find in all of our __init__.py files are lines that look like this:

```
>>> from menu import Menu
... from remote import BCIRemote
... from robot import Robot
... from scope import TimeScope, SpectrumScope
... from grid import GridHierarchy
... from helloworld import HelloWorld
... from diagnostic import FreqButton
```

More import statements! These are the contents of the __init__.py file within the apps folder that you see at the bottom. These import statements are now selecting new folders in which to seek the same class we were looking for before: HelloWorld. Now, the import statement is looking in the folder helloworld of the apps folder. For ease I will start using this folder hierarchy notation: apps.helloworld. When the helloworld folder is opened, there are two items, another __init__.py and a script named helloworld.py

Again, the __init__.py will run and within this script we see:

```
>>> from helloworld import HelloWorld
...
```

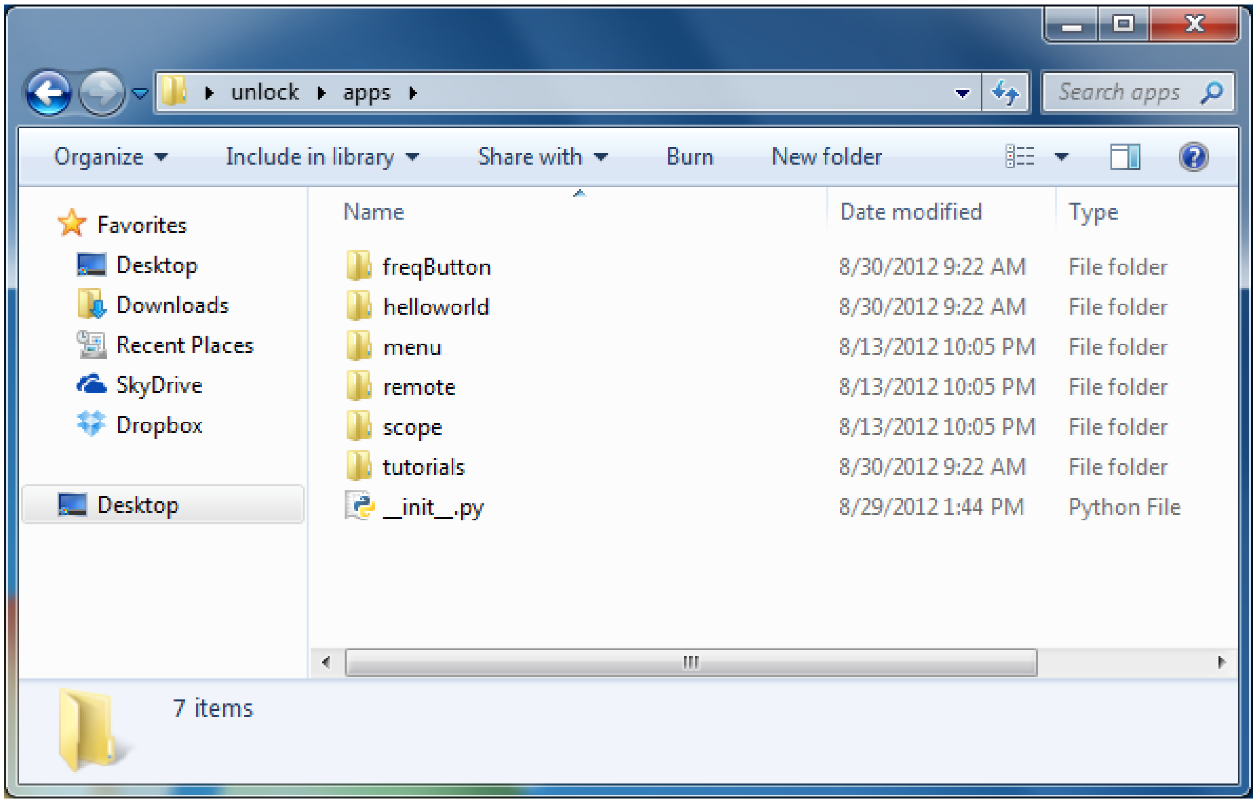


Name	Date modified	Type
apps	8/30/2012 9:23 AM	File folder
core	8/30/2012 9:23 AM	File folder
resource	8/13/2012 10:05 PM	File folder
stimuli	8/30/2012 9:25 AM	File folder
runDiagnostic.py	8/28/2012 1:39 PM	Python File
runtime.py	8/29/2012 1:49 PM	Python File

Figure 6.1: Unlock Project Top Folder

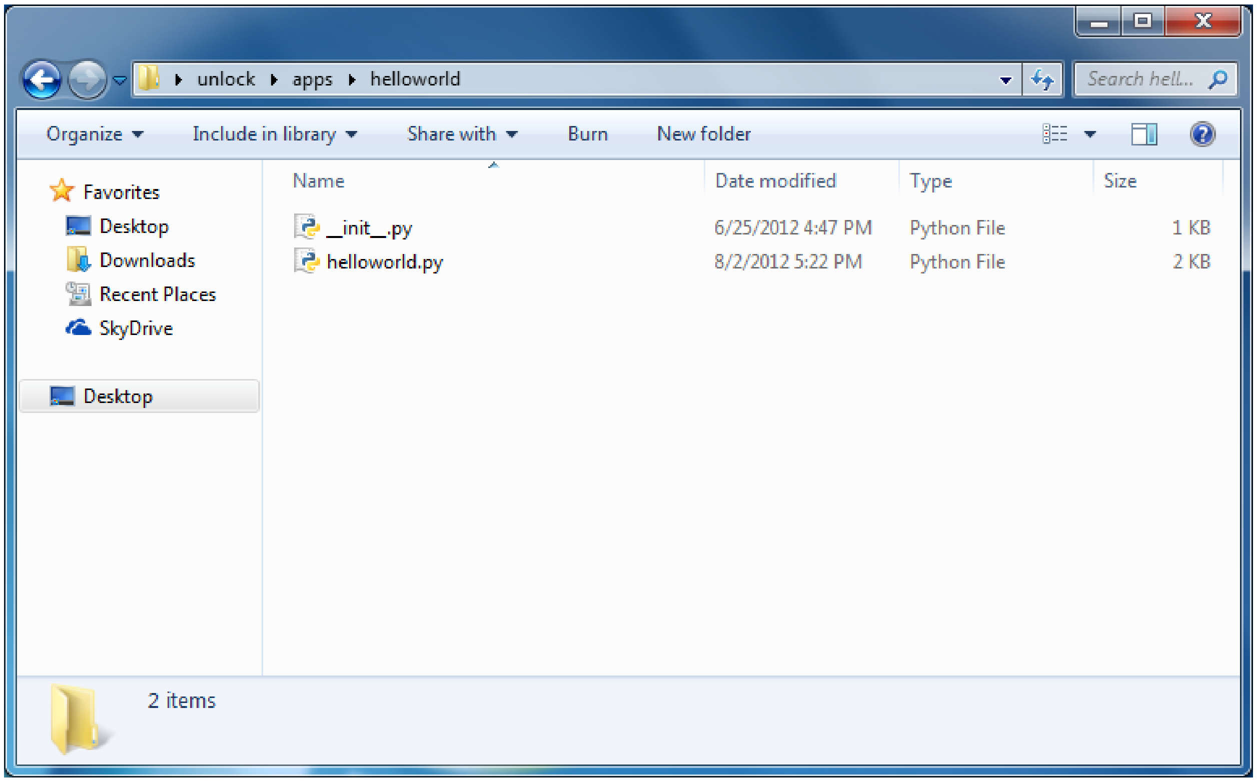
This is __init__.py again saying, go find an item named helloworld, and within that item, find the class HelloWorld. Notice the formatting of the capital letters. This will make a difference when trying to import the classes you have made.

So when building your apps, it might be most useful to store your app in a containing folder. In those cases, you will need to create __init__.py files to allow for proper imports. Now let's go over the Tutorial on the Screen Class and Pyglet.



Name	Date modified	Type
freqButton	8/30/2012 9:22 AM	File folder
helloworld	8/30/2012 9:22 AM	File folder
menu	8/13/2012 10:05 PM	File folder
remote	8/13/2012 10:05 PM	File folder
scope	8/13/2012 10:05 PM	File folder
tutorials	8/30/2012 9:22 AM	File folder
__init__.py	8/29/2012 1:44 PM	Python File

Figure 6.2: Unlock Project apps Folder



Name	Date modified	Type	Size
__init__.py	6/25/2012 4:47 PM	Python File	1 KB
helloworld.py	8/2/2012 5:22 PM	Python File	2 KB

Figure 6.3: Unlock Project apps.helloworld Folder

CHAPTER

SEVEN

SCREEN.PY TUTORIAL PART 1: PROPERTIES AND LINES

We have gone through the steps for a basic BCI app, and how to properly call it and run it. All of our apps are GUI based. There is some graphical element to what is being used whether it is an internet browser, or a e-mail client. There are many graphics libraries available for Python, however the one we have chose is Pyglet. Pyglet is a wonderful low-level, Open.GL based graphics library, that allows for hardware optimization. Its only fault is that Pyglet's syntax can be time consuming to learn and use. Therefore we have created this class Screen in order to use Pyglet's function but in an intuitive way.

For developers that would rather use the API documentation to learn more about screen.py, can go straight to the core.screen module

Screen gives you five main methods for drawing objects to the screen. These are drawText, drawLine, drawLinePlot, drawRect, and loadSprite. There are also four methods that return properties of the screen. These are get_offset, get_size, get_width, and get_height. There is a final method called addGroup which will be explained last.

7.1 Lines

So the easiest thing to draw to the screen is a line. A line is just the shortest distance between two points. So in order to have a line we need to start with two points. Let's look at some example code below.

```
>>> from core import UnlockApplication
...
... class Line1(UnlockApplication):
... name = "line_1"
...
... def __init__(self, screen):
...
...     super(self.__class__, self).__init__(screen)
...
...     w        = screen.get_width()
...     h        = screen.get_height()
...
...     half_w   = w/2
...     half_h   = h/2
...
...     self.line1   = screen.drawLine(0, half_h, w, half_h)
...
... def update(self, dt, decision, selection):
...     """Updates with every new decision or selection"""
...     pass
```

This should look very much like the app code we had for HelloWorld with a few things changed. First of all the class name is changed, as well as the class member: “name”. I have also defined some parameters from the screen that I want to use.

```
>>> w       = screen.get_width()
... h       = screen.get_height()
...
... half_w  = w/2
... half_h  = h/2
```

screen.get_width and screen.get_height do exactly what they say. They return the width and height respectively of the screen being used. screen.get_size is similar that it returns both the width and height as a list.

I also defined half of the screen vertically and horizontally so I can use those values as well. I will refer to the horizontal plane as the x-direction, and the vertical direction as the y-direction. The origin will be in the lower left hand corner of the screen. For this script, all I am interested is one line drawn across the center of the screen horizontally. That means my first point will have x-coordinate of 0 and a y-coordinate of half of the screen height. My second point will have x-coordinate of the full screen width, and y-coordinate again at half of the screen height. The function to draw this line will be screen.drawLine(x1, y1, x2, y2), and will be called like this.

```
>>>
...     self.line1  = screen.drawLine(0, half_h, w, half_h)
```

I put in the values of my first point which are (0, half_h) and then the values of my second point (w,half_h)

Now if we want to see what this looks like, we need to make sure to set up the other two parts of running apps. The first is to add it out runtime script like so:

```
>>> from core import Line1
... from apps import HelloWorld
...
... def get_apps(window):
...
...     full = Screen(0,0,window.width, window.height)
...
...     app1 = Line1(full)
...
...     return [app1]
...
... if __name__ == '__main__':
...     from core import viewport
...     viewport.controller.apps = get_apps(viewport.window)
...     viewport.start()
```

As well as adding the correct __init__.py files in the app folder and subfolders. This app on the screen will look something like this.

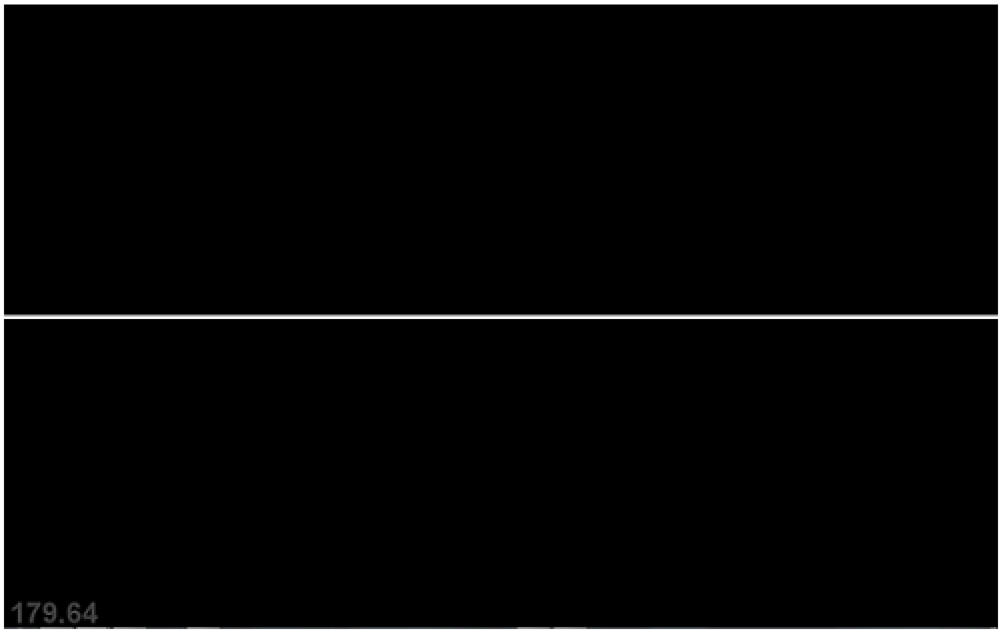
So now we have the most basic shape on the drawn on the screen!

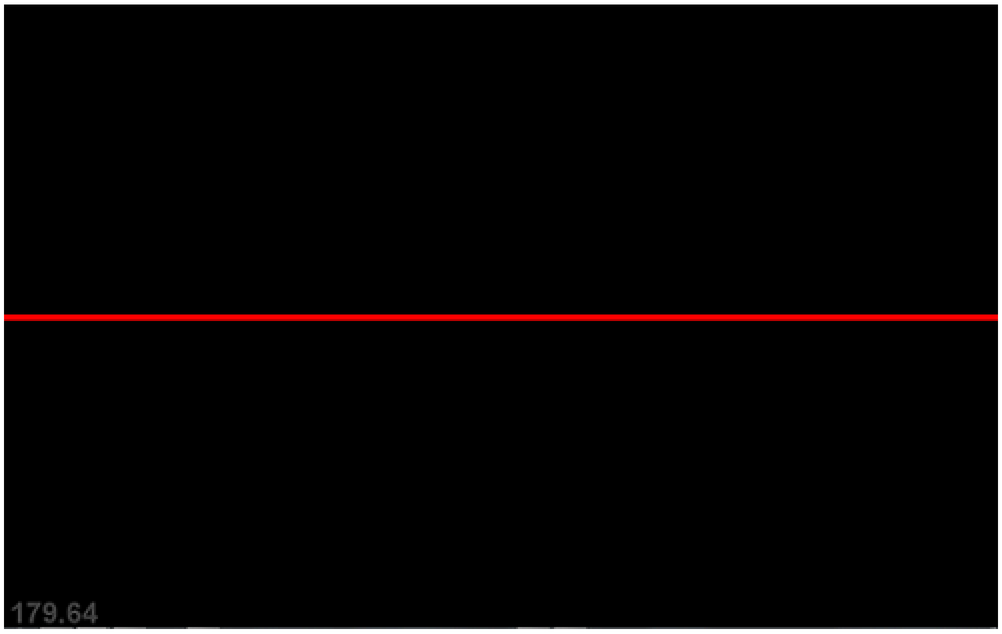
The default color of lines are white, but we can make it whichever color we want. For example if we want the line to be red, we can set the parameter in the function

```
>>>
...     red =(255,0,0)
...     self.line1  = screen.drawLine(0, half_h, w, half_h, color=red)
```

and it will show up like this:

Note: The value for line color is a tuple of length = 3


179.64


179.64

Now, let's move on to rectangles and text.

CHAPTER

EIGHT

SCREEN.PY TUTORIAL PART 2: RECTANGLES AND DECISIONS

8.1 Rectangles

Rectangles are very similar to lines in that it requires a set of points, and some very basic variables The function to use to draw a rectangle is screen.drawRect(x,y,width,height). X is the x-coordinate of the leftmost coordinate of the rectangle, and Y is the y-coordinate of the bottommost coordinate of the rectangle. Width is the width of the rectangle, and height is the height. These are not necessarily the points of the topmost and rightmost coordinates, but instead the number of pixels away from the rectangles (x,y) origin. Let's take a look at what an app with rectangles looks like.

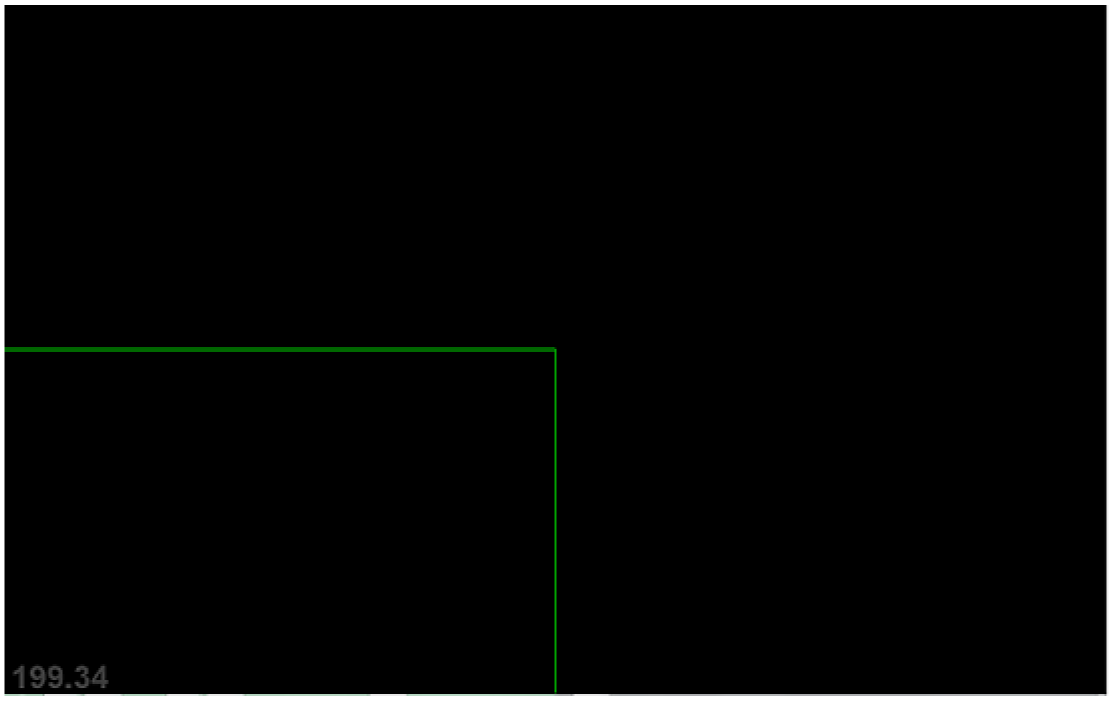
```
>>> from core import UnlockApplication
...
... class Rectangle1(UnlockApplication):
... name = "rectangle_1"
...
...    def __init__(self, screen):
...
...        super(self.__class__, self).__init__(screen)
...        w        = screen.get_width()
...        h        = screen.get_height()
...
...        half_w   = w/2
...        half_h   = h/2
...
...        green    = (0,255,0)
...
...        self.rect1  = screen.drawRect(0,0, half_w, half_h, color=green)
...
...    def update(self, dt, decision, selection):
...        """Updates with every new decision or selection"""
...        pass
```

As you can see we are trying to draw one rectangle, with an origin at (0,0). It will have width and height of the half the screen width and half the screen height.

This is what it looks like however, when it is called in the runtime script.

We can't see the bottom or left parts of the rectangle. Its helpful, when drawing shapes to the screen, to be aware of little things like this, when we can't see all of the parts of the screen.

We can do two things to fix this. The first is to not draw shapes so close to the edge of the screen. The second is to draw in a buffer. In the second version of the code, I have placed the green rectangle in the center of the screen,

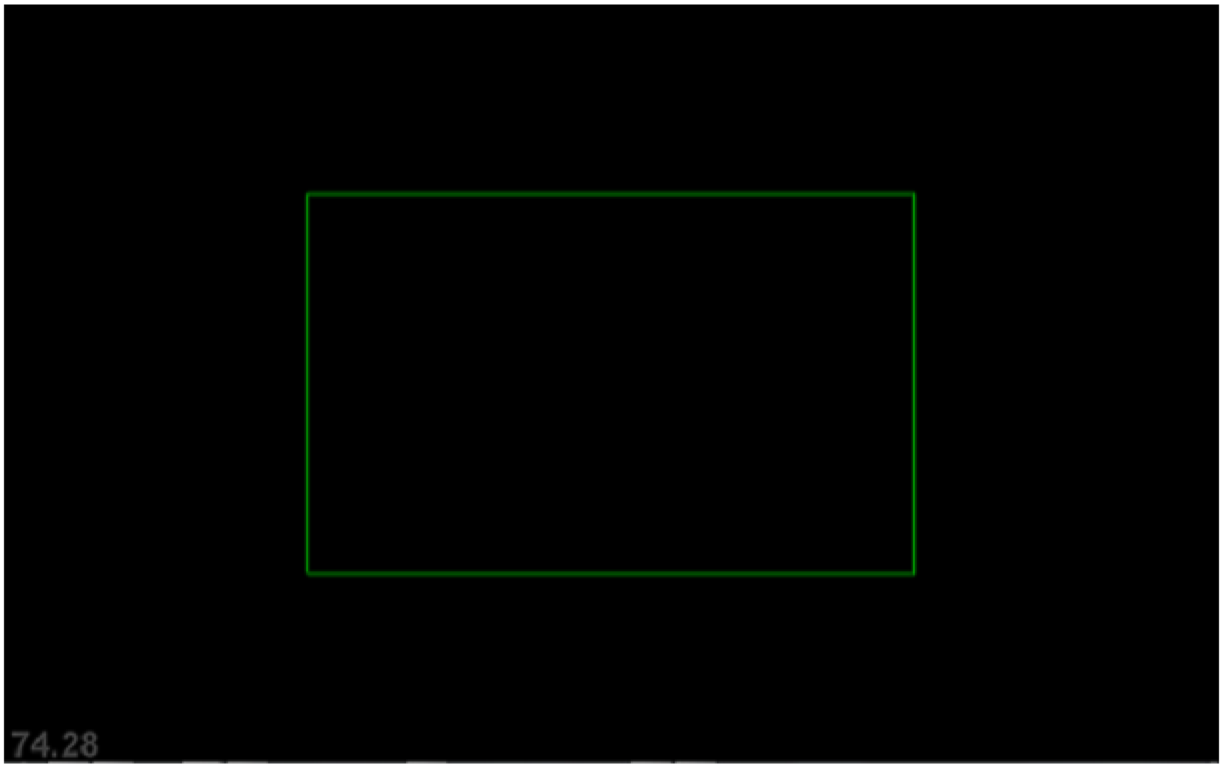


The code looks like this now.

```
>>> from core import UnlockApplication
...
... class Rectangle2(UnlockApplication):
... name = "rectangle_2"
...
...    def __init__(self, screen):
...
...        super(self.__class__, self).__init__(screen)
...        w       = screen.get_width()
...        h       = screen.get_height()
...
...        buffer  = {'g':0.25}
...        originG = [w*buffer['g'],h*buffer['g']]
...        sizeG   = [w*(1-2*buffer['g']),h*(1-2*buffer['g'])]
...
...        green   = (0,255,0)
...
...        self.rect1  = screen.drawRect(originG[0],originG[1], sizeG[0], sizeG[1], color=green)
...
...    def update(self, dt, decision, selection):
...        """Updates with every new decision or selection"""
...        pass
```

And appears like this:

In the Rectangle2 class you will notice that i have set three new variables. buffer is a dictionary that associates 'g' with 0.25. originG sets my origin for the rectangle. In this case I want the x-coordinate of the rectangle to be at 25% of the total width, and the y-coordinate of the rectangle to be at 25% of the total height. sizeG sets the total size of the rectangle. This is trickier I start with the total width and height of the screen times 1. From this I want to subtract 25% of the screen on both sides. So I do 2*my buffer percentage subtracted by 1 to give me the amount of screen I want in each direction.



There is one more trick I want to share with you about rectangles. That is moving them around the screen. In the next example code, class Rectangle3, I take the rectangle from before and I add a new method. This method is moveRect(box, x_step, y_step)

```
>>> from core import UnlockApplication
...
... class Rectangle3(UnlockApplication):
... name = "rectangle_3"
...
...     def __init__(self, screen):
...
...         super(self.__class__, self).__init__(screen)
...         w        = screen.get_width()
...         h        = screen.get_height()
...
...         buffer   = {'g':0.25}
...         originG  = [w*buffer['g'],h*buffer['g']]
...         sizeG    = [w*(1-2*buffer['g']),h*(1-2*buffer['g'])]
...
...         green    = (0,255,0)
...
...         self.rect1   = screen.drawRect(originG[0],originG[1], sizeG[0], sizeG[1], color=green)
...
...     def update(self, dt, decision, selection):
...         """Updates with every new decision or selection"""
...         if decision:
...             if decision == 1:
...                 self.moveBox(self.rect1,0,1)
...             elif decision == 2:
...                 self.moveBox(self.rect1,0,-1)
...             elif decision == 3:
```

```
...                 self.moveBox(self.rect1,-1,0)
...             elif decision == 4:
...                 self.moveBox(self.rect1,1,0)
...
...     def moveBox(self, box, x_step, y_step):
...         """Moves box by n x_step or y_step. """
...         if x_step:
...             box.vertices[::2] = [i + int(x_step)*self.x_pix for i in box.vertices[::2]]
...         if y_step:
...             box.vertices[1::2] = [i + int(y_step)*self.y_pix for i in box.vertices[1::2]]
```

I define how many pixels I want every x movement and y movement to be in the __init__ method. I then add code to the update method with a decision. Decisions are either a 1,2,3 or 4 representing up, down, left and right. As you can see I have set the update function to add or subtract a step from the rectangle vertices depending on what decision it receives. For BCI, the decision is something a user will make with the EEG system. For testing purposes, you can use the arrow keys. Try out class Rectangle3.

Now that we have mastered Rectangles we can move on to Text and Sprites.

CHAPTER

NINE

SCREEN.PY TUTORIAL PART 3: TEXT, AND SELECTION

9.1 Text

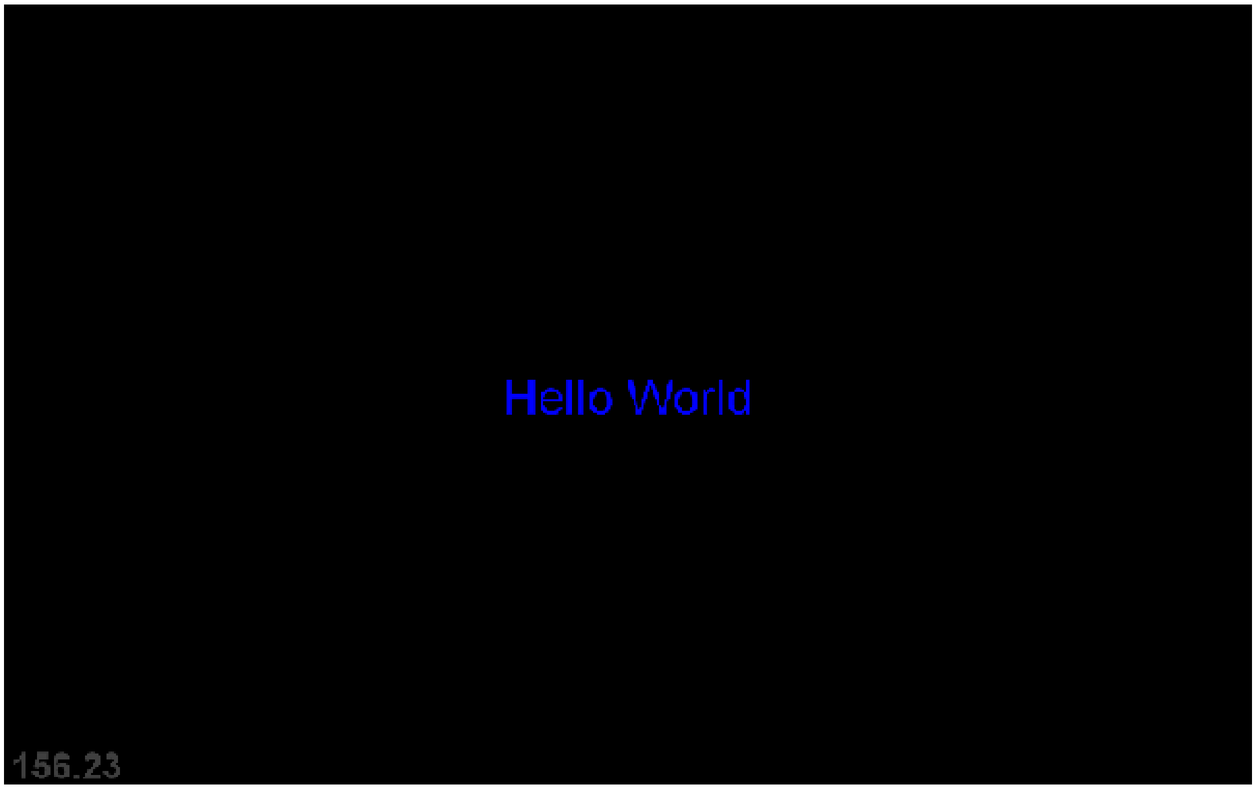
Drawing text to the screen is a little bit easier than line or rectangles. To draw text we will use the method screen.drawText(text, x, y). It is a simple as that. You choose a string to be your text, and then an anchor where you want the center of the text to be.

Implementing this in code would look something like this:

```
>>> from core import UnlockApplication
...
... class Text1(UnlockApplication):
...     name = "text_1"
...
...     def __init__(self, screen):
...
...         super(self.__class__, self).__init__(screen)
...         w        = screen.get_width()
...         h        = screen.get_height()
...
...        half_w   = w/2
...        half_h   = h/2
...
...        blue     = (0,0,255,255)
...
...        text = 'Hello World'
...
...        self.text1 = screen.drawText(text,half_w,half_h, color=blue)
...
...    def update(self, dt, decision, selection):
...        """Updates with every new decision or selection"""
...        pass
```

Note: Text colors values are a tuple of length four, the last value being opacity.

This is very similar to the Hello World version we looked at in the first tutorial. Notice though that the color value is a tuple of length four not three. This will make it harder to use one value for color when creating your apps. The result of this app looks like this:



One neat thing about text is that you can change the parameters of the object in the update function. For example, here is another version of the code that changes the color of the text at every selection.

```
>>> from core import UnlockApplication
... from random import randint
...
... class Text2(UnlockApplication):
...     name = "text_2"
...
...     def __init__(self, screen):
...
...         super(self.__class__, self).__init__(screen)
...         w       = screen.get_width()
...         h       = screen.get_height()
...
...         half_w  = w/2
...         half_h  = h/2
...
...         blue    = (0,0,255,255)
...
...         text = 'Hello World'
...
...         self.text1 = screen.drawText(text,half_w,half_h, color=blue)
...
...     def update(self, dt, decision, selection):
...         """Updates with every new decision or selection"""
...         if selection:
...             self.text1.color = (randint(0,255), randint(0,255),
...                                 randint(0,255), 255)
```

Try this out for yourself to see how this works. Selection is the fifth input that your app can receive, after the four directional decisions. This is meant to be a method of choosing one item or another. If for example you have a cursor that moves over multiiple items, the selection input can allow one or more to be chosen.

There you go. There are the three most important parts of the core.screen() module. The apps that are currently in the repository make use of these methods, as well as a few more non-important ones. More screen Tutorials are soon to follow.

CHAPTER

TEN

BCI APPLICATIONS

10.1 frequency Button Selector

10.1.1 `freqButton` Module

class `apps.freqButton.freqButton.`**`FreqButton`**(*screen*, *freqs*, *stimulus*, *Spectrum=None*)

Bases: `core.application.UnlockApplication`

Buttons for selecting frequency of flash rate

This app takes in a set of frequency rates and arranges them in a grid. The technician can select a frequency to be the frequency of the stimulus that is presented to the subject. It requires a SSVEP stimulus variable to be passed to the script. An optional variable for spectrum scope variable can be passed to the script so that an indicator moves on the Spectrum Scope

Parameters

- **screen** – Screen which to draw grid to
- **freqs** – List of frequency values to be displayed as floats
- **stimulus** – The stimulus you wish to control with buttons
- **spectrum** – the spectrum scope object you are using if it includes an expected frequency indicator

`moveBox`(*box*, *x_step*, *y_step*)

Moves the square cursor around the grid. Allows items to be selected

Parameters

- **box** – Variable name of pyglet rectangle to move
- **x_step** – Amount of step sizes to move horizontally. Either positive or negative
- **y_step** – Number of step sizes to move vertically. Either Positive or negative

`name` = **'freqButton'**

`update`(*dt*, *decision*, *selection*)

Updated with every new decision or selection.

Parameters

- **dt** – Time step
- **decision** – Decision for the app. Usually 0-3(4 decisions) for directional movement
- **selection** – Either 0 or 1.

10.2 helloworld Package

10.2.1 `helloworld` Module

class `apps.helloworld.helloworld.`**`HelloWorld`**(*screen*)

Bases: `core.application.UnlockApplication`

Sample Unlock application for testing framework. Displays the words Hello World. Arrow keys move the words, and hitting the space bar re-centers them and changes the color of the text

`name` **= 'Hello World'**

`update`(*dt*, *decision*, *selection*)

Updates with every new decision or selection

10.3 remote Package

10.3.1 `remote` Module

class `apps.remote.remote.`**`BCIRemote`**(*screen*, *comP=4*)

Bases: `core.application.UnlockApplication`

A TV remote interface with buttons for 0-9, power, channel up and down, volume up and down, and favorite channel. The menu can be navigated by arrow keys for testing and a selection is determined by the space bar. These commands send a one character code through a COM port to an Arduino IR Remote. currently, our IR code transceivers only function with Sony TV. There is also functionality in the Arduino code for servos to move the base of the remote for panning and tilting. Full functionality of this feature is not done in this version.

Parameters

- **screen** – Window Size
- **comP** – COM Port+1 where the IR transceiver is plugged in, if any.

`moveBox`(*box*, *x_step*, *y_step*)

Moves box by n x_step or y_step. x_step and y_step are defined by the height of the grid elements

`moveLazer`()

When run, checks the value of self.servo and outputs correct arduino command

`name` **= 'Remote'**

`sendCode`()

Transmits one character code to COM port designated in beginning of program

`switchState`()

Switch between remote Function state and remote Movement State

Relies on self.state variable. Turns on and off commands and abilities depending on value of self.state.

`update`(*dt*, *decision*, *selection*)

Updated with every new decision or selection.

Parameters

- **dt** – Time step
- **decision** – Decision for the app. Usually 0-3(4 decisions) for directional movement
- **selection** – Either 0 or 1.

If there is a decision, update checks the current state, and moves the cursor around the screen in the appropriate direction for the appropriate state If there is a selection, update checks the current state, and activates the function or movement that has been selected. This is done by running either the sendCode or

10.4 scope Package

10.4.1 `frequency` Module

class `apps.scope.frequency.`**`SpectrumScope`**(*screen*, *numchan=1*, *labels=None*, *dur=2*, *fs=256*, *magYlim=(-90, 200)*, *refresh=5*, *mode='auto'*, *debug=False*, *graph='pwr'*, *mag='abs'*, *decoder=False*)

Bases: `core.application.UnlockApplication`

A 2-D plot displaying the relative magnitude of the FFT taken from recorded EEG data

This program takes in one or multiple channels of voltage data, converts each channel from the time domain to the frequency domain, and plots each as a function of Magnitude vs. Frequency.

Parameters

- **screen** – Window Size
- **freqs** – The frequencies being used for stimulation.
- **numchan** – Number of channels expected in output buffer
- **labels** – Labels for the channels
- **duration** – Sampling time
- **fs** – Sampling rate
- **magYlim** – set limits for data channel amplitudes
- **refresh** – update rate for plotter
- **mode** – whether to update ylim after every <duration> cycle
- **debug** – Using real-time data or random data for testing
- **graph** – Outputs the graph either as power spectrum(pwr) or HSD(hsd)
- **mag** – Graphs power output as either tha absolute value('abs') or log in decibels ('dB')
- **decodeInfo** – Variables and modules for decoding time-series data

`gets_samples` **= True**

`moveBox`(*box*, *x_step*, *y_step*)

Moves the frequency slider along the X axis

Parameters **box** – name of pygame rectangle to be moved

:x_step

`moveFreqIndi`(*freq*)

Graphical representation of current stimulating frequency.

Parameters **freq** – the frequency currently being outputed by the frequency selector

`name` **= 'SpectrumScope'**

pushBuffer(*data*)
Pushes incoming data onto the plotting buffer.

Parameters **data** – incoming data, assumed to be interleaved

sample(*data*)
Collects data from socket, and sends it to pushBuffer. If debug mode is turned on generates random test data

Parameters **data** – data collected by socket buffer.

screenMapX(*x*)
Sets scale for length of data with number of pixels on screen

Parameters **x** – x-coordinate(s) of data value. Can be list of x values, or a single value.

Returns Converted x-coordinates to pixels on screen

screenMapY(*y*, *ii=0*)
Sets scale for length of data with number of pixels on screen

Parameters

- **y** – y-coordinate(s) of data value. Can be list of x values, or a single value.
- **ii** – optional. integer spacing for multiple data values

Returns Converted y-coordinates to pixels on screen

update(*dt*, *decision*, *selection*)
Updated with every new decision or selection.

Parameters

- **dt** – Time step
- **decision** – Decision for the app. Usually 0-3(4 decisions) for directional movement
- **selection** – Either 0 or 1.

10.4.2 `time` Module

class `apps.scope.time.`**TimeScope**(*screen*, *numchan=1*, *labels=None*, *duration=2*, *fs=256*, *ylim=(-100, 100)*, *refresh=33*, *mode='auto'*, *debug=False*)
Bases: `core.application.UnlockApplication`

Class to output contents of a multidimensional list [chan x time] to a line plot - yvals are inverted

Parameters

- **screen** – Window Size
- **numchan** – number of channels expected in output buffer
- **labels** – Labels for the channels
- **duration** – Sampling time in seconds
- **fs** – Sampling rate of data
- **ylim** – set limits for data channel amplitudes
- **refresh** – update rate for plotter
- **mode** – whether to update ylim after every <duration> cycle
- **debug** – Using real-time data or random data for testing

`gets_samples` = **True**

`name` = **'Time Scope'**

`pushBuffer`(*data*)
Pushes incoming data onto the plotting buffer.

Parameters **data** – incoming data, assumed to be interleaved

`sample`(*data*)
Collects data from socket, and sends it to pushBuffer. If debug mode is turned on generates random test data

Parameters **data** – data collected by socket buffer.

`screenMapX`(*x*)
Sets scale for length of data with number of pixels on screen

Parameters **x** – x-coordinate(s) of data value. Can be list of x values, or a single value.

Returns Converted x-coordinates to pixels on screen

`screenMapY`(*y*, *ii=0*)
Sets scale for length of data with number of pixels on screen

Parameters

- **y** – y-coordinate(s) of data value. Can be list of x values, or a single value.
- **ii** – optional. integer spacing for multiple data values

Returns Converted y-coordinates to pixels on screen

`update`(*dt*, *decision*, *selection*)
Updated with every new decision or selection.

Parameters

- **dt** – Time step
- **decision** – Decision for the app. Usually 0-3(4 decisions) for directional movement
- **selection** – Either 0 or 1.

CHAPTER

ELEVEN

CORE MODULES

11.1 Core.application()

11.1.1 `application` Module

class `core.application.`**`UnlockApplication`**(*screen*)

Bases: `object`

Unlock Application base class

Sets methods to be used by all Unlock BCI applications

Parameters **screen** – Screen to be passed to the app

`attach`(*app*)

add an app to the list of apps :param app: the app module to be added

`close`(***kwargs*)

Method to close an app given by controller

`gets_samples` **= False**

`name` **= 'Unlock Application'**

`on_open`(*kwargs*)

Called when the application is opened.

`on_return`(*kwargs*)

Called when a parent receives control from a child app.

`open`(*name*, ***kwargs*)

Method to open an app given to controller

Parameters **name** – Name of app

`quit`()

Called when the display is shutting down to enable graceful cleanup.

`root`()

Returns the root app associated with this app

`sample`(*data*)

Samples data in the application makes use of raw EEG data

Parameters **data** – list of time-series data

`update`(*dt*, *decision*, *selection*)

Unlock Applications must define the update method

Parameters

- **dt** – time step of update cycle
- **decision** – An integer number given as 1-4 to determine a direction of movement
- **selection** – Boolean 0 or 1 to signify the selection of an object where the cursor is

11.2 Core.controller()

11.2.1 `controller` Module

class `core.controller.`**`Controller`**(*apps*)

Handles the import of data and sends the decisions and selections to the update methods of each app. Also handles opening and closing of apps.

`acquire`()

Receives raw data from ports as well decision and selection values

`draw`()

Draws apps that are listed in the list of apps

`quit`()

Closes all ports, and quits all apps

`update`(*dt*)

runs update method in list of apps

Also runs the acquire method to receive data, decisions and selections

Parameters **dt** – time step for each update cycle

11.3 Core.Screen()

11.3.1 `screen` Module

class `core.screen.`**`LinePlot`**(*line*, *y_offset*)

`updateData`(*y_data*, *offset=0*)

class `core.screen.`**`Screen`**(*x*, *y*, *width*, *height*)

`addGroup`(*order=0*)

adds a group to the list of groups

`drawLine`(*x1*, *y1*, *x2*, *y2*, *color=(255*, *255*, *255)*, *group=None*)

Draws a line between two points on the screen

Parameters

- **x1** – x-coordinate of first point (Pixels from left)
- **y1** – y-coordinate of first point (Pixels from bottom)
- **x2** – x-coordinate of second point (Pixels from left)
- **y2** – y-coordinate of second point (Pixels from bottom)

- **color** – Color of line. Tuple of length three.
- **group** – Batch group

`drawLinePlot` (*vertices*, *color=(255, 255, 255)*, *group=None*)

Given a set of vertices, will plot lines between each vertices in the list

Parameters

- **vertices** – list of vertices to be plotted.
- **color** – Color of Line plot. Tuple of length three
- **group** – batch group

`drawRect` (*x*, *y*, *width*, *height*, *color=(255, 255, 255)*, *fill=False*, *group=None*)

Draws a rectangle. Either just the frame, or a filled in shape.

Parameters

- **x** – x-ccordinate of leftmost vertice. In unit pixels from Left
- **y** – y-cooridnate of bottommost vertice. In unit pixels from Bottom
- **width** – width of rectangle. Unit pixels
- **height** – height of rectangle. Unit pixels
- **color** – color of rectangle. Tuple of length three.
- **fill** – Toggle whether the rectangle should be filled or left as a frame
- **group** – batch group

`drawText` (*text*, *x*, *y*, *font='Helvetica'*, *size=48*, *color=(255, 255, 255, 255)*, *group=None*)

Draws text at a specific point on the screen

Parameters

- **text** – Text to display
- **x** – x-coordinate of center of text(Pixels from left)
- **y** – y-coordinate of center of text(Pixels from bottom)
- **font** – Font of text
- **size** – Size of text
- **color** – Color of text. Tuple of length four.
- **group** – Batch group

`get_height` ()

Returns height of screen

`get_offset` ()

Returns the offset of the screen off the window

`get_size` ()

Returns height and width of screen

`get_width` ()

Returns width of screen

`loadSprite` (*filename*, *x*, *y*)

Load an image for use as an image

Parameters

- **filename** – name and location of file to use
- **x** – x-coordinate of center of sprite. In unit pixels from left
- **y** – y-coordinate of center of sprite. In unit pixels from bottom

11.4 Core.viewport()

11.4.1 `viewport` Module

`core.viewport.`**`on_draw`**`()`
clears the window, and draws results from controller

`core.viewport.`**`on_key_press`**`(`*symbol*, *modifiers*`)`
Handles Key pressing events.

`core.viewport.`**`start`**`()`
Starts the clock and starts the Unlock Interface

CHAPTER

TWELVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX