Python

- is a general-purpose high-level programming language;
- uses clear, minimalist syntax that promotes readability;
- has a very large, comprehensive standard library;
- has a large, active user community among scientists;
- is open-source, with a permissive license.
Structure by Analogy with Other Modules

Required and Recommended Packages

Python 2.4, 2.5 or 2.6

For Signal Processing and Signal Source modules:
- NumPy by Travis Oliphant — for representing numeric arrays
- BCPy2000
- IPython by Fernando Perez — for interactive debugging
  - pyreadline also required under Windows
- SciPy by Enthought — useful for signal processing
- (Matplotlib by John D. Hunter — for plotting during debugging)

For the Application module:
- NumPy, BCPy2000, IPython & pyreadline
- VisionEgg by Andrew Straw — for stimulus display
  - Python Imaging Library by Fredrik Lundh
  - pygame by Pete Shimmers & others
  - PyOpenGL by Mike C. Fletcher
- (pywin32 by Mark Hammond & others)
- (PyAudio by Hubert Pham)
Object-Oriented Programming in Python

```
class dog:
    def bark(self):
        print "Woof."

>>> scooby = dog()
>>> scooby.bark()
Woof.
```

- `dog` is a class.
- `bark()` is a `dog` method.
- `scooby` is a `dog` instance.
- When executing his own methods, `scooby` refers to himself (i.e. the current instance) as `self`.

```
class dog:
    def __init__(self, size="big"):
        self.size = size
    def bark(self):
        if self.size == "small":
            print "Yip!"
        else:
            print "Woof."

>>> scrappy = dog(size="small")
>>> scrappy.bark()
Yip!
```

- `__init__` is a special method called the constructor.
- Information about a dog’s size is stored in the attribute `self.size`.
- Here, if the constructor argument `size` is missing, it defaults to the string “big”.

```
class chihuahua (dog):
    def __init__(self, size="small"):
        self.size = size

>>> gidget = chihuahua()
>>> gidget.bark()
Yip!
```

- `chihuahua` is a subclass of the `dog` class.
- It inherits all the methods of a `dog` automatically...
- ...except that we have replaced the constructor such that the default size is different.

```
class BciApplication (BciGenericApplication):
    pass
```

- The Python keyword `pass` is required in order to signify “this section of code is empty”.

Launch BCI2000 by double-clicking on `batch\TUTORIAL.BAT`. From one of the IPython windows, type `edit Tutorial.py`

```
class BciApplication (BciGenericApplication):
    pass
```

A `BciApplication` instance is created when the application module is launched. From the prompt in the “PythonApp” window, it can be addressed as `self`. Explore `self.params`, `self.states`, `self.stimuli`, `self.screen`, ...

Keep this file open for later. Switch back to the BCI2000 Operator, press Set Config and then Start.
Getting Help and Documentation

Use the tab key to explore. Start by typing `self.screen` but instead of pressing return, press tab.

Among others, the `self.screen` object appears to have attributes called `size` and `color`. Find out what their values are. Does changing `self.screen.color` do what you expect?

The question-mark is also a useful tool. Functions or object methods reveal their prototypes, and maybe also their documentation: try entering `self.doc?` and see what you get. Or type `self.doc??` for a source-code listing of that method.

Now that we know about the `self.doc()` method, let's use it: try `self.doc('API')` as an example.

An Exercise in Debugging

```python
class BciApplication (BciGenericApplication):
    pass
```

This is a pretty boring application. Re-run it but with the ShowSignalTime parameter turned on. (Suspend the run, press Config, and you'll find it in the “PythonApp” tab.)

While it is running, the system speed can be changed from the Signal Source module. So, switch to the “PythonSrc” console window. Watch the clock and explore the effects of `self.stop()`, `self.step()`, and `self.cont()`.

Note also the effect on `self.states`.

Creating Stimuli in the Application Module

Stimuli should be created in the `Initialize()` hook. They are made by passing a VisionEgg Stimulus subclass, together with the necessary parameters for constructing an instance later on, to the API method `self.stimulus()`:

```python
from VisionEgg.Text import Text
class BciApplication (BciGenericApplication):
    def Initialize(self, indims, outdims):
        self.stimulus('greeting', Text, text='Hello World!')
```

A `BciStimulus` object is created and returned as a convenient wrapper around the VisionEgg information. The first argument is a unique name that lets you retrieve the object later—as `self.stimuli['greeting']` or `self.stimuli.greeting` in this case.

Stimuli may also be created on-the-fly from the PythonApp prompt, if you have already pressed Set Config. You can also change `self.stimuli.greeting.text` while running, or similarly `self.color`, `self.position` or `self.anchor`.

Try it. After changing and saving the Python code, you will have to Quit and re-launch BCI2000 completely.
Hooks Specific to the Application Module

The “phase machine” is a thread which changes to different parts of an experimental trial at the appropriate times. Define it in `Phases`:

```python
def Phases(self):
    from random import randint
    self.phase(name='rest', duration=randint(1000,3000), next='baseline')
    self.phase(name='baseline', duration=3000, next='startcue')
    self.phase(name='startcue', duration=1000, next='imagine')
    self.phase(name='imagine', duration=5000, next='stopcue')
    self.phase(name='stopcue', duration=1000, next='rest')
    self.design(start='rest', new_trial='baseline')
```

On every phase transition, `Transition` is called:

```python
def Transition(self, phase_name):
    pass
```

Implement a `Phases` hook for a hypothetical BCI experiment. Implement a `Transition` hook such that the name of the current phase is displayed on the screen.

Further Debugging Resources

- Play back an existing file (see `self.doc('Replay')`).
- Enter Python’s command-line debugger `pdb` by calling `self.dbstop()` in your code (warning: flaky).
- Use `--PythonSrcClassFile=BCI2000Tools/AudioSourceModule.py` to record real-time signals from the microphone—useful for realtime debugging even when you don’t have an EEG amp available, and to verify the timing of your auditory stimuli.

Quit, then launch the “Triangle” demo using `batch\PythonDemo1_Triangle.bat`

```python
edit Triangle.py and identify the parts of the python code that
- interpret parameters,
- define stimuli,
- schedule the sequence of events in each trial,
- manipulate stimulus properties.
```

Start a new run after turning on `VisualizeSpatialFilter` and `VisualizePythonSigFilter` in the “Visualize” config tab. These show the input and output of the `PythonSignalProcessing` filter. Now switch to the “PythonSig” command window, and try replacing `self.Process` on the fly. Note the effect on the signals, and on the visual cursor behaviour:

```python
In[1]: def square(signal):
    .....: return signal.A ** 2
    .....:
In[2]: self.Process = square
```

Also try out the “Template” demo (which uses the default python filename, `BciApplication.py`).

Thanks

- Thomas Schreiner (original author of BCPy2000 beta)
- Christian Puzicha
- Jason Farquhar
- Bernd Battes
- Jürgen Mellinger
- Gerwin Schalk