BCI2000 and Python

Jeremy Hill

Wadsworth Institute
New York State Dept. of Health

Albany, NY
1. Python
2. BCI2000 Architecture and Python
3. Object-Oriented Programming in Python
4. BCPy2000's Application Framework (with exercises)
...is a general-purpose high-level programming language;
... is a general-purpose high-level programming language;
... uses clear, minimalist syntax that promotes readability;
Python

- ... is a general-purpose high-level programming language;
- ... uses clear, minimalist syntax that promotes readability;
- ... has a very large, comprehensive standard library;
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;
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.
Simple TCP/IP socket programming in C
The same thing in Python
The same thing in Python

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

# A simple echo client
import sys, socket
host = 'localhost'
port = 50000
size = 10

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Create socket
s.connect((host, port))
# Connect
nBytes = s.send(' '.join(sys.argv[1:]))
# Send
while nBytes > 0:
    data = s.recv(size)
    sys.stdout.write(data)
    nBytes -= len(data)
print
s.close()

# A simple echo server
import sys, socket
host = ''
port = 50000
backlog = 5
size = 32

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Create socket
s.bind((host, port))
# Bind to
s.listen(backlog)
# Mark the
while 1:
    client, address = s.accept() # Wait for a client to connect
    while 1:
        data = client.recv(size) # Receive message from client
        if not data: break # Break if no more data from the
    client.send(data) # Echo message back to client
    sys.stdout.write(data)
    # Write the message to stdout
print # Write a final newline
client.close() # Close client socket
```
Free high-quality Python software exists for...

numeric array processing
signal / wavelet processing
machine-learning / data-mining
parallel computing
2-D plotting
3-D rendering
neuroimaging
image processing
audio processing
symbolic math
neural network modelling
visual stimulus presentation
game programming
physics modelling
virtual reality
cryptography
file management
string & XML processing
midi interfacing
music theory
web programming
database access
content management
source code version control
GUI building
office applications
financial data processing
...
Outline

Python

Architecture

Objects

PythonApp

Signal Source

Signal Processing

User Application

State Variables

State Variables

State Variables

File

BCPy2000
Structure by Analogy with Other Modules
Structure by Analogy with Other Modules
Structure by Analogy with Other Modules
Structure by Analogy with Other Modules
Structure by Analogy with Other Modules
Structure by Analogy with Other Modules

Outline
Python
Architecture
Objects
PythonApp

Your code

Toolbox
Toolbox
Toolbox

MatlabSignalProcessing.exe

N' → T' → Toolbox

N → T → Toolbox

$
Structure by Analogy with Other Modules
Required and Recommended Packages

**Python** 2.4, 2.5 or 2.6
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
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
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
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
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)
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)
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
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
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 Shinners & others
- PyOpenGL by Mike C. Fletcher
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 Shinners & others
  - PyOpenGL by Mike C. Fletcher
- (pywin32 by Mark Hammond & others)
- (PyAudio by Hubert Pham)
class dog:
    def bark(self):
        print "Woof."
Object-Oriented Programming in Python

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

>>> scooby = dog()
>>> scooby.bark()
Woof.
Object-Oriented Programming in Python

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

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

*dog* is a *class*. 
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.
Object-Oriented Programming in Python

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

- **dog** is a *class*.
- **bark()** is a *dog* method.
- **scooby** is a *dog* instance.

```python
>>> scooby = dog()
>>> scooby.bark()
Woof.
```
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**.
Object-Oriented Programming in Python

class dog:
    def __init__(self, size="big"):
        self.size = size

    def bark(self):
        if self.size == "small":
            print "Yip!"
        else:
            print "Woof."
Object-Oriented Programming in Python

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

__init__ is a special method called the constructor.
Object-Oriented Programming in Python

class dog:
    def __init__(self, size="big"):
        self.size = size

    def bark(self):
        if self.size == "small":
            print "Yip!"
        else:
            print "Woof."

- `__init__` is a special method called the constructor.
- Information about a dog’s size is stored in the attribute `self.size`.
Object-Oriented Programming in Python

```python
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`.

class dog:

    def __init__(self, size="big"):  # __init__ is a special method called the constructor.
        self.size = size

    def bark(self):
        if self.size == "small":
            print "Yip!"
        else:  # Information about a dog's size is stored in the attribute self.size.
            print "Woof."

>>> scrappy = dog(size="small")  # Here, if the constructor argument size is missing, it defaults to the string "big".
>>> scrappy.bark()
Yip!
Object-Oriented Programming in Python

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!

>>> scooby = dog()
>>> print scooby.size
big
>>> scooby.bark()
Woof.

- `__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”.
Object-Oriented Programming in Python

class chihuahua (dog):
    def __init__(self, size="small"):
        self.size = size
Object-Oriented Programming in Python

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

chihuahua is a subclass of the dog class.
Object-Oriented Programming in Python

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

- `chihuahua` is a subclass of the `dog` class.
- It inherits the all the methods of a `dog` automatically...
Object-Oriented Programming in Python

```python
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...
Object-Oriented Programming in Python

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.
A simple BciApplication.py file

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

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

class BciApplication (BciGenericApplication):
    pass
A simple BciApplication.py file

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

class BciApplication (BciGenericApplication):
    pass

The Python keyword `pass` is required in order to signify “this section of code is empty”.
A simple `BciApplication.py` file

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

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

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

Keep this file open for later. Switch back to the BCI2000 Operator, press **Set Config** and then **Start**.
Launch BCI2000 by double-clicking on `batch\TUTORIAL.BAT`. From one of the IPython windows, type `edit Tutorial.py`

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

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

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

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

class BciApplication (BciGenericApplication):
    pass

This is a pretty boring application.
An Exercise in Debugging

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.)
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.
Standard "Hooks" (usable in all modules)

```python
out_signal = self.Process(in_signal)
```

is called on every new packet of signal data.
Standard “Hooks” (usable in all modules)

- `self.StartRun()` is called when the Start button is pressed.

- `out_signal = self.Process(in_signal)` is called on every new packet of signal data.
Standard “Hooks” (usable in all modules)

- `self.StartRun()` is called when the **Start** button is pressed.

- `out_signal = self.Process(in_signal)` is called on every new packet of signal data.

- `self.StopRun()` is called when the **Suspend** button is pressed or the run otherwise ends.
Standard “Hooks” (usable in all modules)

- `self.Initialize(in_signal_dims, out_signal_dims)` is called when the Set Config button is pressed and all the parameter values make sense.

- `self.StartRun()` is called when the Start button is pressed.

- `out_signal = self.Process(in_signal)` is called on every new packet of signal data.

- `self.StopRun()` is called when the Suspend button is pressed or the run otherwise ends.
Standard “Hooks” (usable in all modules)

- `out_signal_properties = self.Preflight(in_signal_properties)` is called when the Set Config button is pressed, to sanity-check parameter values. (The return argument can be omitted if there is no change.)

- `self.Initialize(in_signal_dims, out_signal_dims)` is called when the Set Config button is pressed and all the parameter values make sense.

- `self.StartRun()` is called when the Start button is pressed.

- `out_signal = self.Process(in_signal)` is called on every new packet of signal data.

- `self.StopRun()` is called when the Suspend button is pressed or the run otherwise ends.
Standard “Hooks” (usable in all modules)

- `param_def, state_def = self.Construct()` is called when the module starts up.

- `out_signal_properties = self.Preflight(in_signal_properties)` is called when the **Set Config** button is pressed, to sanity-check parameter values. (The return argument can be omitted if there is no change.)

- `self.Initialize(in_signal_dims, out_signal_dims)` is called when the **Set Config** button is pressed and all the parameter values make sense.

- `self.StartRun()` is called when the **Start** button is pressed.

- `out_signal = self.Process(in_signal)` is called on every new packet of signal data.

- `self.StopRun()` is called when the **Suspend** button is pressed or the run otherwise ends.
Standard “Hooks” (usable in all modules)

- `param_defs, state_defs = self.Construct()` is called when the module starts up.

- `out_signal_properties = self.Preflight(in_signal_properties)` is called when the Set Config button is pressed, to sanity-check parameter values. (The return argument can be omitted if there is no change.)

- `self.Initialize(in_signal_dims, out_signal_dims)` is called when the Set Config button is pressed and all the parameter values make sense.

- `self.StartRun()` is called when the Start button is pressed.

- `out_signal = self.Process(in_signal)` is called on every new packet of signal data.

- `self.StopRun()` is called when the Suspend button is pressed or the run otherwise ends.

- `self.Destruct()` is called when the module shuts down.
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!')
```
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 `.color`, `.position` or `.anchor`.
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()`:

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

```python
from random import randint

def Phases(self):
    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
```
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.
Quit, then launch the “Triangle” demo using `batch\PythonDemo1_Triangle.bat`
Quit, then launch the "Triangle" demo using \texttt{batch\PythonDemo1\_Triangle.bat} and edit \texttt{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.
Quit, then launch the “Triangle” demo using `batch\PythonDemo1_Triangle.bat`

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

**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:

```
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`).
Further Debugging Resources

- Play back an existing file (see `self.doc('Replay')`).
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).
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.
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.

Thanks

Thomas Schreiner
(original author of BCPy2000 beta)

Bernhard Schölkopf
(director at the Max Planck Institute, Tübingen)

Christian Puzicha
Jason Farquhar
Bernd Battes

Jürgen Mellinger
Gerwin Schalk