Pyff Documentation

Release 2009.9

Bastian Venthur

November 29, 2009

CONTENTS

This is Pyff's documentation. Feedback developers are probably interested in the documentation of *Feedback base classes*

Contents:

CHAPTER

ONE

LIBRARY DOCUMENTATION

This section contains the documentation for the various Python modules provided by Pyff. Developers of Feedbacks are probably interested only in the documentation of the Feedback base classes.

Contents:

1.1 bcinetwork — Networking between the different Pyff components.

class BciNetwork(*ip, port, myport=None*)

Wrapper for Communication between Feedback Controller and GUI.

getAvailableFeedbacks()

Get available Feedbacks from Feedback Controller.

get_variables()

Get variables (name, type and value) from currently running Feedback.

pause()

Send 'pause' to Feedback Controller.

play()

Send 'play' to Feedback Controller.

quit()

Send 'quit' to Feedback Controller.

receive(*timeout*)

Receive a signal.

send_init(*feedback*)

Send 'send_init(feedback)' to Feedback Controller.

send_signal(*signal*)

Send a signal.

stop()

Send 'stop' to Feedback Controller.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.2 bcixml — Encoding and Decoding of BCI XML messages.

Encoding and decoding of bci-xml packages.

class BciSignal (*data, commands, type*)
Represents a signal from the BCI network.

A BciSignal object can be translated to XML and vice-versa.

exception DecodingError
Message could not be decoded.

exception EncodingError
Something message cound not be encoded.

exception Error
Our own exception type.

class XmlDecoder ()
Parses XML strings and returns BciSignal containing the data of the signal.

Usage: decoder = XmlDecoder() try:

bcisignal = decoder.decode_packet(xml)

except DecodingError: ...

decode_packet (*packet*)
Parse the XML string and return a BciSignal.

A DecodingError is raised when the parsing of the packet failed.

class XmlEncoder ()
Generates an XML string from a BciSignal object.

Usage: enc = XmlEncoder() try:

xml = enc.encode_packet(bcisignal)

except EncodingError: ...

encode_packet (*signal*)
Generates an XML packet from a BciSignal object.

An EncodingError is raised if the encoding failed.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.3 feedbackcontroller — Feedback Controller.

class FeedbackController (*plugin=None, fbpath=None, port=None*)
Feedback Controller.

Controlls the loading, unloading, starting, pausing and stopping of the Feedbacks. Can query the Feedback for it's variables and can as well set them.

handle_signal (*signal*)
Handle incoming signal.

send_to_feedback(*signal*)
Send data to the feedback.

send_to_peer(*signal*)
Send signal to peer.

start()
Start the Feedback Controller's activities.

stop()
Stop the Feedback Controller's activities.

class UDPDispatcher(*fc*)
UDP Message Hanldeer of the Feedback Controller.

handle_read()
Handle incoming signals.

Takes incoming signals, decodes them and forwards them to the Feedback Controller.

send_signal(*signal*)
Send signal to the GUI.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.4 feedbackprocesscontroller — Controls the Feedback Processes.

class FeedbackProcess(*feedbackClass, ipcReady, port, fbplugin*)
Process that wrapps the Feedback's activities.

run()
Run the FeedbackProcess' activities in the new process.

class FeedbackProcessController(*plugindirs, baseclass, timeout*)
Takes care of starting and stopping of Feedback Processes.

get_feedbacks()
Return a list of available Feedbacks.

start_feedback(*name, port, fbplugin*)
Starts the given Feedback in a new process.

stop_feedback()
Stops the current Process.

First it tries to join the process with the given timeout, if that fails it terminates the process the hard way.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.5 ipc — Inter Process Communication.

Inter Process Communication.

this module provides classes to ease the inter process communication (IPC) between the Feedback Controller and the Feedbacks

class FeedbackControllerIPCChannel(*conn, fc*)
IPC Channel for Feedback Contoller's end.

handle_message(*message*)
Handle message from Feedback.

class FeedbackIPCChannel(*conn, feedback*)
IPC Channel for Feedback's end.

handle_message(*message*)
Handle message from Feedback Controller.

class IPCChannel(*conn*)
IPC Channel.

Base for the channels, the Feedback Controller and the Feedbacks need.

This Class transparently takes care of de-/serialization of the data which goes through the IPC. Derived classes should implement

handle_message(self, message)

to do something useful and use

send_message(self, message)

for sending messages via IPC.

collect_incoming_data(*data*)
Append incoming data to input buffer.

found_terminator()
Process message from peer.

handle_close()
Handle closing of connection.

handle_message(*message*)
Do something with the received message.

This method should be overwritten by derived classes.

send_message(*message*)
Send message to peer.

class IPCConnectionHandler(*fc*)
Waits for incoming connection requests and dispatches a FeedbackControllerIPCChannel.

close_channel()
Close the channel to the Feedback.

handle_accept()
Handle incoming connection from Feedback.

handle_close()
Handle closing of connection.

handle_error()
Handle error.

send_message(*message*)
Send the message via the currently open connection.

get_feedbackcontroller_connection()
Return a connection to the Feedback Controller.

ipcloop()
Start the IPC loop.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.6 `PluginController` — Finding and Loading Feedbacks.

class `PluginController`(*plugindirs, baseclass*)
Finds, loads and unloads plugins.

`find_plugins`()
Returns a list of available plugins.

`load_plugin`(*name*)
Loads the given plugin and unloads possibly sooner loaded plugins.

`test_plugin`(*root, filename*)
Test if given module contains a valid plugin instance.

Returns None if not or (name, modulename) otherwise.

`unload_plugin`()
Unload currently loaded plugin.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

1.7 `RollbackImporter` — Importing and Unloading of Modules.

class `RollbackImporter`()
RollbackImporter.

RollbackImporter instances install themselves as a proxy for the built-in __import__ function that lies behind the 'import' statement. Once installed, they note all imported modules, and when uninstalled, they delete those modules from the system module list; this ensures that the modules will be freshly loaded from their source code when next imported.

Usage:

if self.rollbackImporter: self.rollbackImporter.uninstall()

self.rollbackImporter = RollbackImporter() # import some modules

`uninstall`()
Unload all modules since __init__ and restore the original import.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

CHAPTER

TWO

FEEDBACK BASE CLASSES

This section provides documentation for the Feedback base classes available in Pyff.

Contents:

2.1 `Feedback` — Feedback Base Class

This module contains the Feedback class, which is the baseclass of all feedbacks.

class `Feedback`(*port_num=None*)

Base class for all feedbacks.

This class provides methods which are called by the FeedbackController on certain events. Override the methods as needed.

As a bare minimum you should override the on_play method in your derived class to do anything useful.

To get the data from control signals, you can use the "_data" variable in your feedback which will always hold the latest control signal.

To get the data from the interaction signals, you can use the variable names just as sent by the GUI.

This class provides the send_parallel method which you can use to send arbitrary data to the parallel port. You don't have to override this method in your feedback.

`inject`(*module*)

Inject methods from module to Feedback Controller.

`on_control_event`(*data*)

This method is called after the FeedbackController received a control signal. The FeedbackController parses the signal, extracts the values stores the resulting tuple in the object-variable "data" and calls this method.

Override this method if you want to react on control events.

`on_init`()

This method is called right after the feedback object was loaded by the FeedbackController.

Override this method to initialize everything you need before the feedback starts.

`on_interaction_event`(*data*)

This method is called after the FeedbackController received a interaction signal. The FeedbackController parses the signal, extracts the variable-value pairs, stores them as object-variables in your feedback and calls this method.

If the FeedbackController detects a “play”, “pause” or “quit” signal, it calls the appropriate on_-method after this method has returned.

If the FeedbackController detects an “init” signal, it calls “on_init” before “on_interaction_event”!

Override this method if you want to react on interaction events.

on_pause()
This method is called by the FeedbackController when it received a “Pause” event via interaction signal.

Override this method to pause your feedback.

on_play()
This method is called by the FeedbackController when it received a “Play” event via interaction signal.

Override this method to actually start your feedback.

on_quit()
This Method is called just before the FeedbackController will destroy the feedback object. The FeedbackController will not destroy the feedback object until this method has returned.

Override this method to cleanup everything as needed or save information before the object gets destroyed.

on_stop()
This method is called by the FeedbackController when it received a “Stop” event.

Override this method to stop your feedback. It should be possible to start again when receiving the on_start event.

send_parallel(*data, reset=True*)
Sends the data to the parallel port.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

2.2 MainloopFeedback — Base Class for Feedbacks with a Mainloop

class MainloopFeedback(*port_num=None*)
Mainloop Feedback Base Class.

This feedback derives from the Feedback Base Class and implements a main loop. More specifically it implements the following methods from it’s base:

on_init on_play on_pause on_stop on_quit

which means that you should not need to re-implement those methods. If you choose to do so anyways, make sure to call MainloopFeedback’s version first:

def on_play(): MainloopFeedback.on_play(self) # your code goes here

MainloopFeedback provides the following new methods:

init pre_mainloop post_mainloop tick pause_tick play_tick

the class takes care of the typical steps needed to run a feedback with a mainloop, starting, pausing, stopping, quiting, etc.

While running it’s internal mainloop it calls tick repeatedly. Additionally it calls either play_tick or pause_tick repeatedly afterwards, depending if the Feedback is paused or not.

init()
Called at the beginning of the Feedback's lifecycle.

More specifically: in Feedback.on_init().

pause_tick()
Called repeatedly in the mainloop if the Feedback is paused.

play_tick()
Called repeatedly in the mainloop if the Feedback is not paused.

post_mainloop()
Called after leaving the mainloop, e.g. after stop or quit.

pre_mainloop()
Called before entering the mainloop, e.g. after on_play.

tick()
Called repeatedly in the mainloop no matter if the Feedback is paused or not.

Module author: Bastian Venthur <venthur@cs.tu-berlin.de>

CHAPTER

THREE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

F

L

INDEX

O

P

Q

R

S

T

U

X