Pynapse Manual

Python Programming in Synapse

Updated 2025-02-27

Tucker-Davis Technologies
11930 Research Circle
Alachua, FL 32615 USA
Phone: +1.386.462.9622
Fax: +1.386.462.5365

Notices

The information contained in this document is provided "as is," and is subject to being changed, without notice. TDT shall not be liable for errors or damages in connection with the furnishing, use, or performance of this document or of any information contained herein.

The latest versions of TDT documents are always online at https://www.tdt.com/docs/

Table of Contents

Overview

> **Important**
>
> Pynapse underwent a significant revision in v96. This documentation is for v96 and above. For Pynapse v95 documentation, see Pynapse Manual for v95 Synapse.

Pynapse is a gizmo for tightly integrating Python coding into your Synapse experiment.

Many users write external code in Python (or MATLAB) that runs alongside their Synapse experiments. These programs are used for overall experiment control, stimulus delivery, behavioral control, and online analysis - things that are either novel paradigms that don't exist in the current gizmo set or can't easily be programmed to run directly on the real-time hardware.

There are several challenges faced by these users and Pynapse is designed to address these issues with an intuitive and powerful interface.

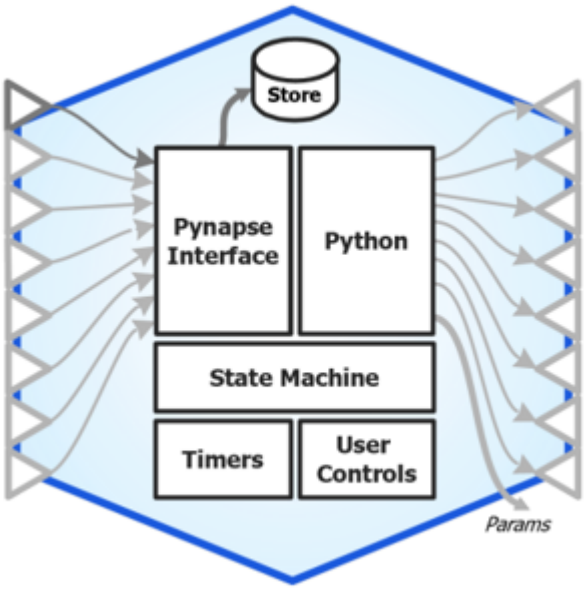
Benefits of Pynapse

Pynapse is more than a great embedded Python editor within Synapse. You get:

- Great Python editor with all the bells and whistles (highlighting, code completion, and more)
- Easy-to-learn, structured programming framework
- Fully automatic Synapse synchronization. Your Python code is saved and version controlled with your experiment
- Powerful hardware (iCon or RZ I/O) and software seamlessly integrated
- Runtime live code monitoring
- Automatic code flowcharting
- Built-in trial and session controls

- Track experiment progress and plot results

Pynapse Gizmo



Pynapse Gizmo Block Diagram

All of this is built into the Pynapse gizmo. Use the Python installation provided in Synapse (or bring in your own) and drop the Pynapse gizmo into your Synapse experiment.

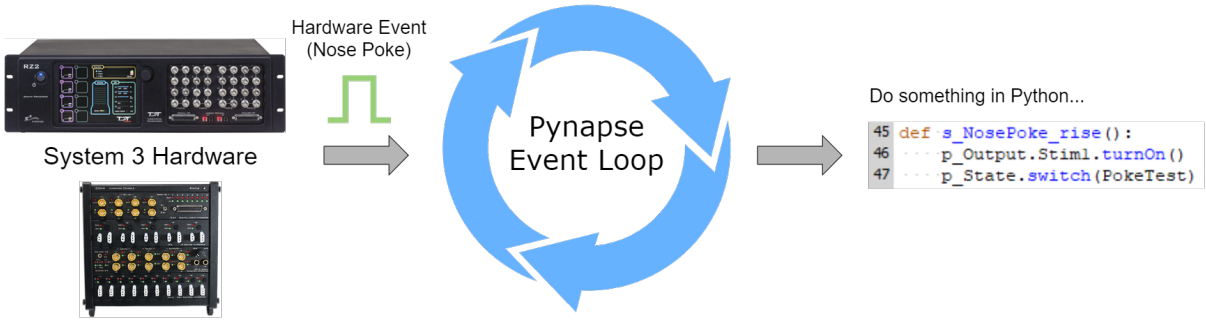
The circuit that runs on the real-time hardware has all the features that anyone writing custom Python code to interact with the hardware. Pynapse runs an optimized polling loop that synchronizes Python to Synapse, faster than existing methods. The State Machine architecture in Pynapse yields tight programs that are easy to read and easy to debug.

Event Loop

A tight polling loop is continuously running and monitoring hardware events defined in the experiment.

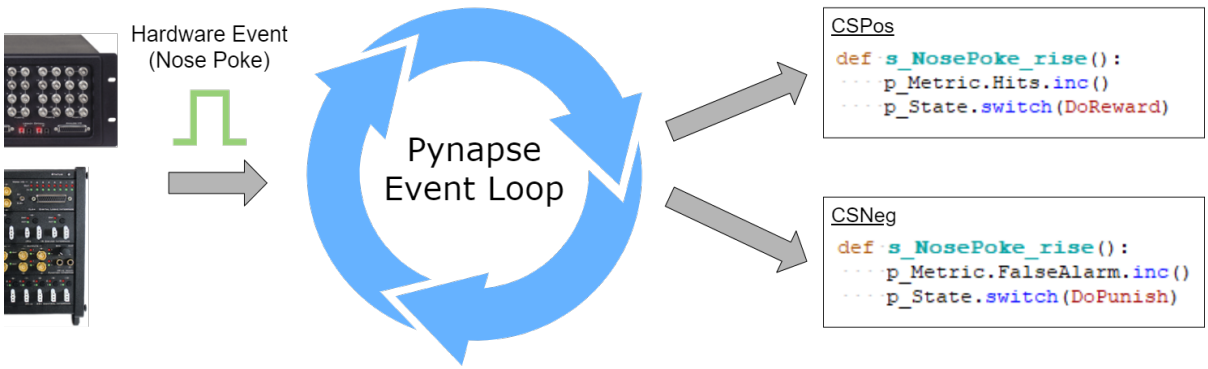
1. Hardware event is detected by the Pynapse event loop.
2. Call is made into Python to execute a method written by the user
3. Call is logged and timestamped
4. Events are sent back to the hardware

All of this happens in milliseconds.



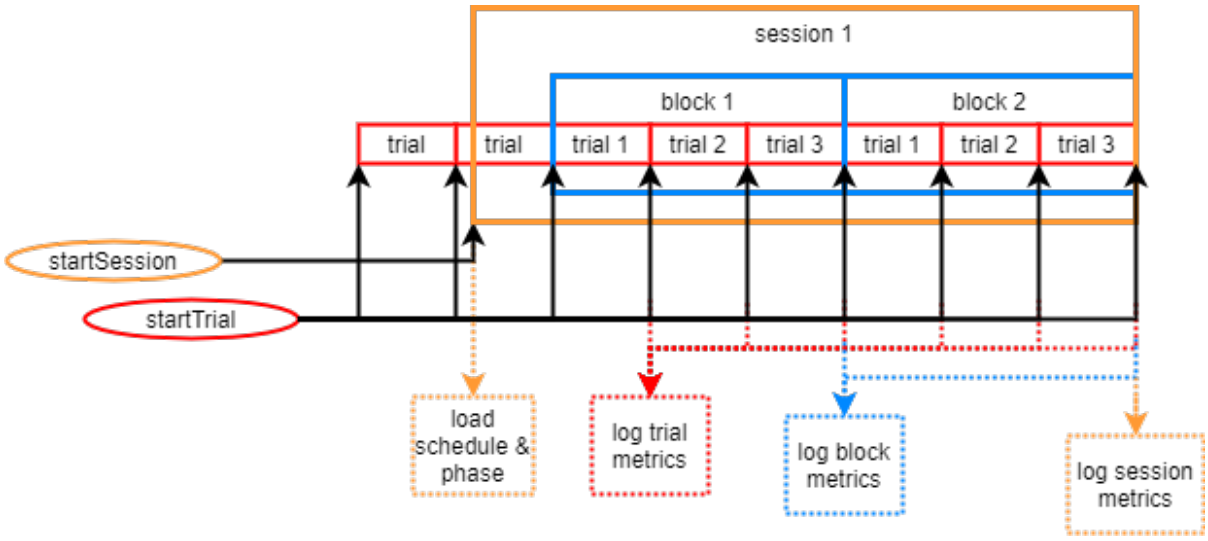
Python State Machine

The Python code can be organized into 'States'. Pynapse keeps track of which state it is in, and hardware events will only trigger Python calls defined within that state. State changes are also controlled by the Python code, and automatically timestamped and stored with the rest of your data for easy analysis.



Session Manager

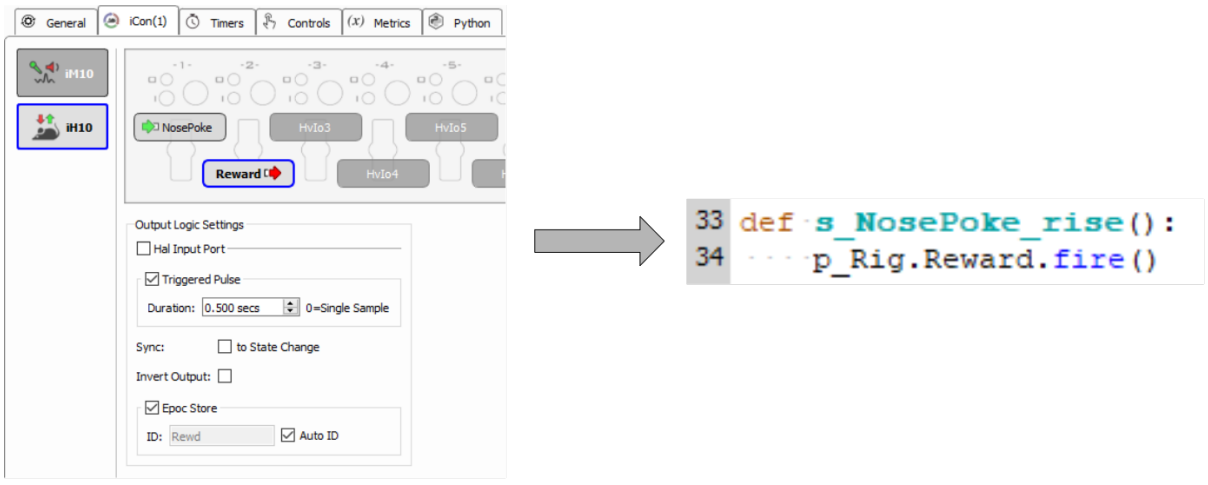
State changes make up trials. Trials can be organized into blocks, and sessions. Define metrics that are logged and plotted at any of these intervals - per trial, per block, per session, or any time they change. Simply set up the number of trials and blocks you want to run, call `startTrial` inside the Python state machine to initiate a trial, and the rest is taken care of automatically.



Take it another step further by organizing groups of experiment settings into Phases, then feed the session manager a schedule of the how many trials/block for each phase. For more open-ended experiments that don't have a fixed number of trials / blocks, you can control the entire trial/block/session flow manually from your Python script. See Sessions for more information.

iCon Integration

Pynapse runs in two different I/O modes. You can use the gizmo inputs and outputs to connect to RZ input/output links, or you can integrate an iCon module directly into Pynapse. Configure all iCon inputs and outputs in Pynapse, and access them in your Python code.



See iCon Inputs and iCon Outputs for more information.

Main Assets of Pynapse Gizmo

Pynapse has the following fundamental asset classes built into it that are accessible in the Python code.

Asset	Description
iCon Inputs	Connect to iCon input events
iCon Outputs	Drive iCon logic outputs under Python control.
States	Switch states, set timeouts, or capture state change events
Timers	Generate custom timer to trigger stimuli or time experiment events
Controls	Provide experiment variable controls with standard user interface elements
Sessions	Manage the flow of the experiment by splitting your sessions into blocks and trials. Run sets of controls in phases and trigger metric updates.
Metrics	Log and plot experiment metrics per trial, block, or session. Share global variables across your Python code.
Logs	Log metrics, control values, and session information during the experiment
UDP	Send a network packet with metrics or custom text during the experiment to a client application
Synapse Control	Capture Synapse state changes, and use SynapseAPI to control other gizmos from within Pynapse
Gizmo Inputs	Connect to gizmo inputs. Can integrate with RZ inputs directly.
Gizmo Outputs	Drive logic and waveform gizmo outputs under Python control. Can integrate with RZ outputs directly.

Common Applications

Program Control

- Start/stop Synapse or other programs based on conditional triggers
- Run Synapse for a set duration

Behavioral Control

- Implement complex behavioral paradigms over trials that control:
  - Cues
  - Waiting periods
  - Input decisions

    - Reward output
    - and more

Signal Analysis and Display

- Collect signals in a triggered buffer or through the API
- Perform calculations, such as:
    - Presentation averaging, spike counting, or FFT
- Plot results using Python plotting libraries (such as Matplotlib)

Stimulus Presentation

- Generate simple or complex stimuli to present during triggered conditions using Pynapse output control or built-in buffers

Requirements

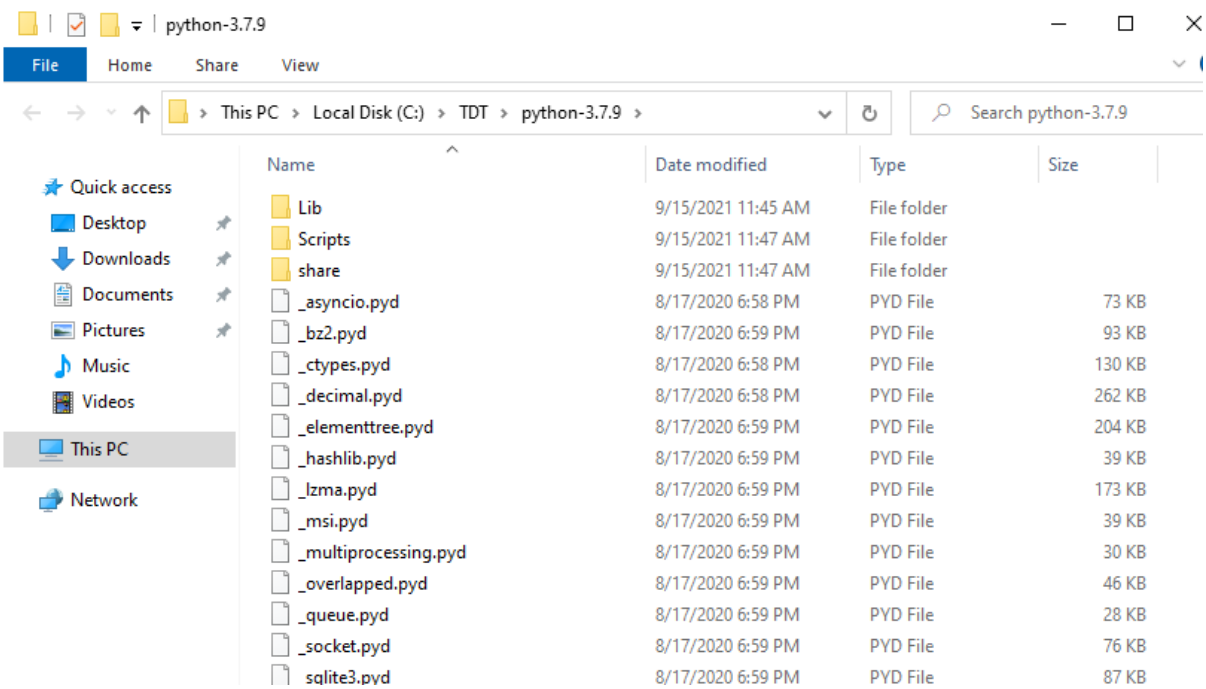
Installing Python

Pynapse requires an installation of Python. A pre-built Python 3.7.9 folder installs to `C:\TDT\python-3.7.9` with Synapse and is automatically configured in Pynapse. It has everything you need to run Pynapse out of the box.

Custom Plotting and User Interface

If you want to use **matplotlib** and/or **PyQt5** to develop your own custom plotting and user interface that extends beyond Pynapse's built-in plots and user controls, follow these instructions:

1. Download the pre-built Python folder from TDT's website.
2. Delete the existing `C:\TDT\python-3.7.9`, and extract the downloaded zip contents into `C:\TDT\python-3.7.9`



Install Other Packages

If you simply want to add a package or two to the pre-installed python folder provided by TDT, then open a command prompt, navigate to the python folder, and run pip install. For example, this installs the scipy package:

```
cd C:\TDT\python-3.7.9
python -m pip install scipy
```

Other Installation Methods

Anaconda Python

Anaconda includes many scientific packages pre-installed for you. Choose this if you want to add these modules to your Python environment or if you are doing any data analysis with Python on this computer. For the Anaconda Python installation method click here.

Standard Python

Advanced users who don't prefer Anaconda can install Python from the web instead. For the standard Python installation method click here.

Quick Start Example

In this example we want to turn a light on with the capture of a button press, and then turn the light off when the button is released. We will have a single input for our button ('Button') and a single output for the light ('Light'). If you were to code this in Python, it might look something like this:

```
import time

while True:

    print('WaitState entered')

    # turn off output
    Light.off()

    # wait until button is pressed
    while Button.false():
        time.sleep(.01)

    print('OnState entered')

    # turn on output
    Light.on()

    # wait until button is false
    while Button.true():
        time.sleep(.01)

    # go back to beginning
```

In Pynapse, instead of using a while loop, we define things called States that can call upon specified methods that we call Slot Methods and Asset Methods when certain hardware events occur. Pynapse knows the current State and is constantly polling the inputs directly from the hardware in a tight loop. When a hardware event occurs that has a matching Slot Method in the current State, that Slot Method gets triggered. The result of this is simplicity in how your state machine is coded - instead of using embedded while loops and conditional statements, you can simply use States and integrated Slot Methods and Asset Methods to issue commands and move from one state to another depending on what events have occurred on the hardware or outputs of code calculations.

Here is a table of important definitions of terms you will see throughout this example and in the Pynapse manual:

Term	Definition
state	Specially defined Python class that has `#StateId = ?` at the end of the class definition
method	Any function defined inside a class using the `def` keyword
slot method	Special method that Pynapse calls in response to events. Slot method names always begin with a `s_` prefix
asset	Special Pynapse classes that interact with inputs, outputs, states, controls, globals, and timers. Assets always begins with a `p_` prefix. For reference, the list of all slot methods that these assets can trigger, and methods you can use to interact with the assets in Python, can be found in the Assets Reference section of this manual.
function	General name used for any function defined outside of a class with the `def` keyword

For the quick start example, we are going to demonstrate how to perform our Light on/off task in two ways: the first is going to use the initial 'Always' State in Pynapse; the second is going to show you how to use multiple States to switch between active pieces of code and perform certain tasks based on captured hardware events.

Using the Always State

The special State called 'Always' runs on every polling loop regardless of what the Pynapse active State is. In our example, we can take advantage of this special State by just focusing on writing code that triggers based on hardware events and not worrying about the Pynapse State machine. Since the 'Always' State is present in the Pynapse Source by default, all we write are Slot Methods and Asset Methods:

```
# Pynapse Source #

class Always:   #StateID = 0

    def s_Button_rise():
        p_Output.Light.turnOn()
        print('Light is on!')

    def s_Button_fall():
        p_Output.Light.turnOff()
        print('Light is off!')
```

A high-level translation of this code would read as follows:

In the Always State, if the 'Button' input true (button is pressed), then turn the 'Light' output on and print "Light is on!"; if the 'Button' input is false (button is released), then turn the 'Light' output off and print "Light is off!".

Here is the same translation using Pynapse terminology:

As mentioned, Pynapse is constantly polling the hardware for events that trigger methods inside of the current active State (Always). In this case, we've defined two Slot Methods 's_Button_rise' and 's_Button_fall' which are part of the Input Assets. The 's_Button_rise' Slot Method triggers when the button is pressed (Button input changes to true) and the 's_Button_fall' triggers when the button is released (Button input changes from true to false). When the button press is detected, Pynapse internally executes the Output Pynapse Asset Method (Light.turnOn), which toggles the output logic signal from low to high. When the button is released, the Light.turnOff Asset Method is executed.

> **Note**
>
> If the button was released first (e.g you were pressing down the button as you went to run-time then let go) the 's_Button_fall' Slot Method would trigger first. These Slot Methods are independent functions that are beholden only to detected hardware events - they do not influence each other.

> **Important**
>
> All Pynapse Slot Methods start with `s_` and all Pynapse Assets start with `p_`. Remembering these two prefixes makes it easy to use code completion inside the Pynapse Code Editor to see all the available Slots and Assets right inside the editor and quickly find what you are looking for.

After we test this code (right-click 'Main' → 'Test') and go to run-time, you will see the Light output toggle on and off with the pressing or release of the Button input.

Using Multiple States

This example was entirely coded in the Always State. Now, we'll take advantage of Pynapse's built-in State Machine. Using multiple Pynapse States has major advantages at runtime, especially as your paradigm increases in complexity. You get a visual indicator of what State you are in for behavior monitoring and the State changes are timestamped and recorded in the data tank synchronized with the rest of your data. There are some other debugging features you get at run-time as well, discussed in more detail in Run-Time and Debugging.

Here we define three states: Always, WaitState and LightOn. When moving to run-time (Standby, Preview, or Record mode) Pynapse enters the Always State and the 's_Mode_recprev' Slot Method is triggered. This is a Synapse Control Slot Method that detects when Synapse has changed mode from Idle to Preview or Record. Once 's_Mode_recprev' is triggered the 'p_State.switch(WaitState)' Asset Method is executed. This key asset tells Pynapse to move to a new State.

```
# Pynapse Source #

class Always:   #StateID = 0

    def s_Mode_recprev():
        p_State.switch(WaitState)
```

Next, Pynapse enters WaitState and immediately runs a specially named Slot Method called 's_State_Enter'. In this example, the s_State_Enter Slot Method turns off the Output (Light) by executing the 'p_Output.Light.turnOff' Output Asset and prints our familiar string.

Now, just like in the first example that only used the Always State, we monitor hardware events and wait for the Input (Button) to become true, wherein another switch Asset Method is executed and we move to our third State (LightOn).

> **Note**
>
> While WaitState is our active State a release of the button will not do anything since there are no Slot Methods in WaitState that detect falling-edge hardware events.

```
class WaitState:   #StateID = ?

    def s_State_enter():
        p_Output.Light.turnOff()
        print('Light is off!')

    def s_Button_rise():
        p_State.switch(LightOn)
```

Finally, we enter the LightOn State. As we enter, the 's_State_enter' Slot Method is triggered and the 'p_Output.Light.turnOn' Asset Method is executed. As you can see, coordinating output events to State changes is a simple way to write conditional events in the Pynapse State Machine. Similarly to the WaitState, Pynapse will continue polling until the 's_Button_fall' Slot

Method is triggered by a button release, wherein we switch back to WaitState and the Output (Light) is turned off.

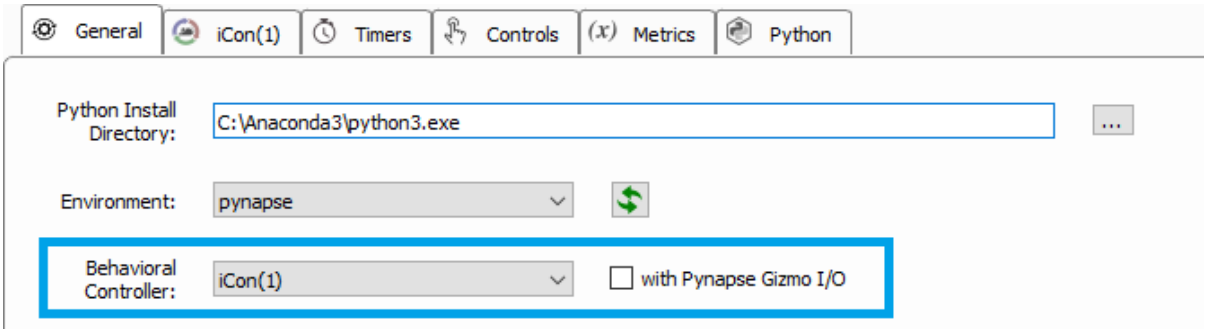
```
class LightOn:   #StateID = ?

    def s_State_enter():
        p_Output.Light.turnOn()
        print('Light is on!')

    def s_Button_fall():
        p_State.switch(WaitState)
```

Important

Notice there were no `while` loops used in these examples. In Pynapse you should almost never write `while` loops. Its polling loop handles this for you. All of your methods should return immediately so the polling loop can continue executing.

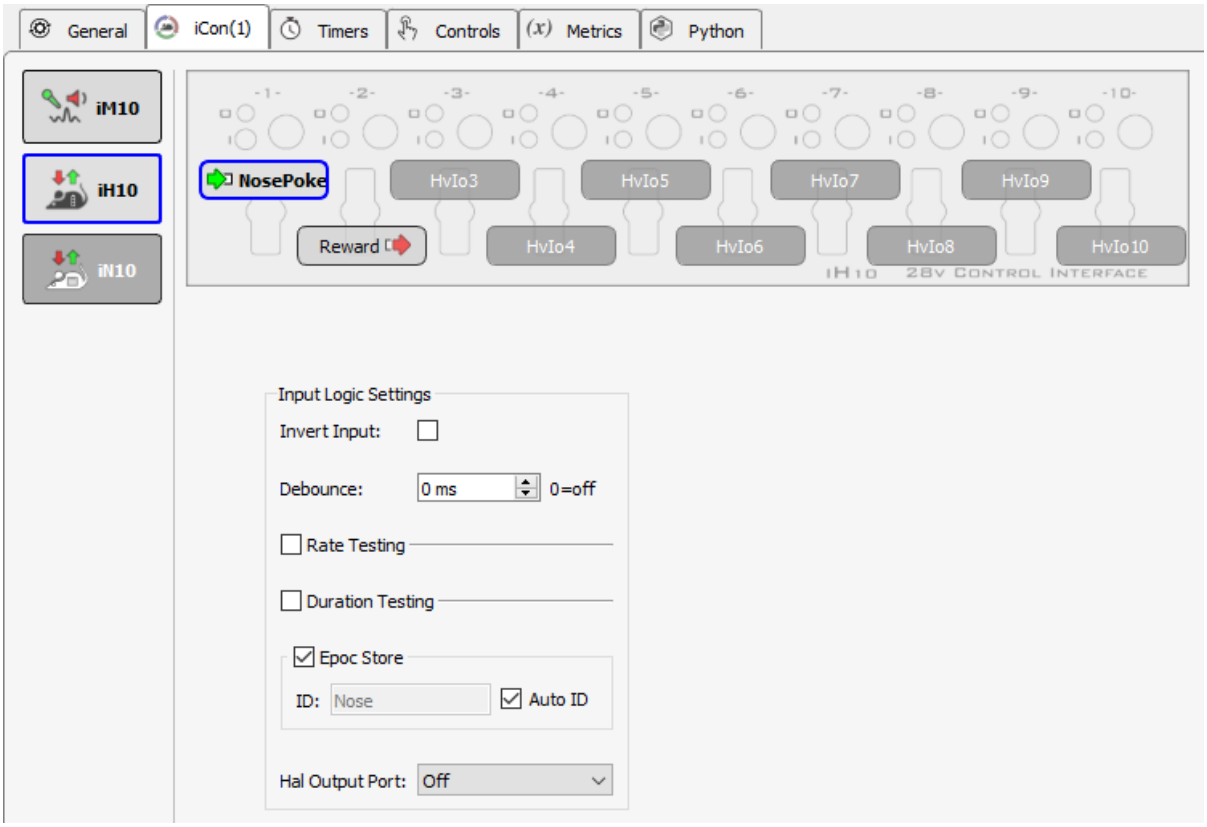
iCon Inputs

When an iCon is attached to the Pynapse Behavioral Controller in the General Tab, an additional iCon tab appears. This gives you a unified interface that lets you configure the iCon inputs/outputs directly within Pynapse and integrates them in the Python code editor.



iCon Tab

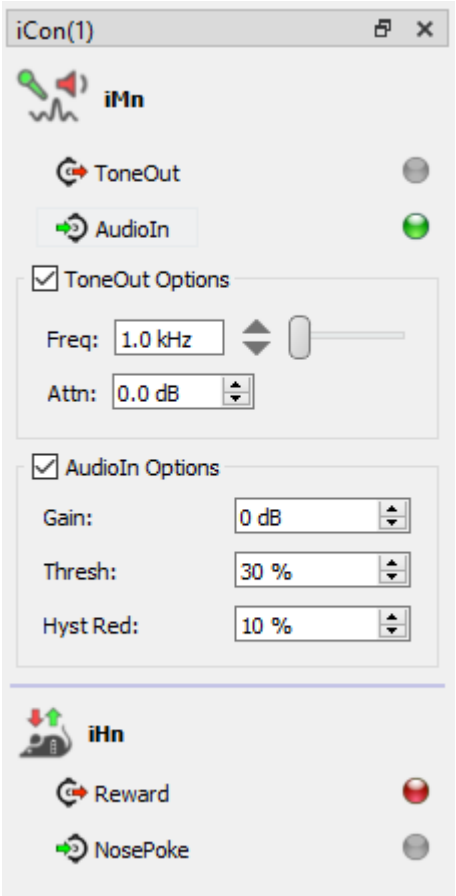
Configure the iCon inputs/outputs in the iCon tab. See Logic Input Processor in the Synapse Manual for information on setting up the iCon inputs and pre-processing them so Pynapse can capture event changes.



iCon Tab

Run-time Interface

The run-time interface has a button for each input and an LED indicator that shows the current state of the input. For the iMn modules, additional options that are available during design-time can also be modified during run-time.



iCon Run-time Interface

Click on the input name to manually trigger it. Hold down CTRL and click an input name to 'mute' it. This prevents the input from triggering Pynapse events.

Slot Methods for Responding to Input States

These input slots capture status information about the inputs. They are available as method definitions inside Pynapse states for each input. Write a method with this name to react to the corresponding event.

Slot name	Operation	Event
s_Input1_rise()	Status	input changed to true
s_Input1_fall()	Status	input changed to false
s_Input1_active()	Duration	input passed the 'Time to Active' duration test (see Duration Testing)
s_Input1_pass()	Duration	input passed the 'Time to Pass' duration test (see Duration Testing)
s_Input1_fail()	Duration	input failed the 'Time to Pass' duration test, after passing 'Time to Active' (see Duration Testing)

Note

The name of each slot method ('Input1' above) gets replaced with the name of your actual input, so if you name the input 'NosePoke' then s_NosePoke_rise() is an available slot.

Example

Move through behavioral states based on status of MyInput

```
class PreTrial:
    def s_MyInput_rise():
        p_State.switch(StartTrial)

class StartTrial:    # StateID = ?
    def s_MyInput_active():
        p_State.switch(ActiveState)

    def s_MyInput_fall():
        p_State.switch(PreTrial)

class ActiveState:    # StateID = ?
    def s_MyInput_pass():
        p_State.switch(PassState)

    def s_MyInput_fail():
        p_State.switch(FailState)
```

Methods

All input methods have the form `p_Rig.{INPUT_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you. The name of each method gets replaced with the name of your actual output, so if you name the input 'NosePoke' then `p_Rig.NosePoke.isOn()` is an available method.

Duration Settings

setActTime

```
p_Rig.MyInput.setActTime(acttime_sec)
```

Override the duration test 'Time to Active' setting. This is only available if 'Duration Testing' is enabled on the input.

Inputs	Type	Description
acttime_sec	float	Time to Active, in seconds

Example

Modify the timing test based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, increase the time to active by 50 ms.
    if p_Metric.success.read() > 5:
        p_Metric.active_time.inc(delta=0.05)
        p_Rig.MyInput.setActTime(p_Metric.active_time.read())
```

setPassTime

```
p_Rig.MyInput.setPassTime(passtime_sec)
```

Override the duration test 'Time to Pass' setting. This is only available if 'Duration Testing' is enabled on the input.

Inputs	Type	Description
passtime_sec	float	Time to Pass, in seconds

Example

Modify the timing test based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, increase the time to pass by 50 ms.
    if p_Metric.success.read() > 5:
        p_Metric.pass_time.inc(delta=0.05)
        p_Rig.MyInput.setPassTime(p_Metric.pass_time.read())
```

setRateThresh

```
p_Rig.MyInput.setRateThresh(ratethr_hz)
```

Override the rate testing 'Rate Threshold' setting. This is only available if 'Rate Testing' is enabled on the input.

Inputs	Type	Description
ratethr_hz	float	Rate testing threshold, in Hertz

Example

Modify the rate threshold based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, increase the rate threshold by 1 Hz.
    if p_Metric.success.read() > 5:
        p_Metric.rate_thresh.inc()
        p_Rig.MyInput.setRateThresh(p_Metric.rate_thresh.read())
```

Manual Control

Manual turn inputs on, off, or pulse during runtime. Useful for debugging.

manualOn

```
p_Rig.MyInput.manualPulse()
```

Manually turn on the input.

Example

Turn on the input when entering a state.

```
def s_State_enter():
    p_Rig.MyInput.manualOn()
```

manualOff

```
p_Rig.MyInput.manualPulse()
```

Manually turn off the input.

Example

Turn off the input when exiting a state.

```
def s_State_exit():
    p_Rig.MyInput.manualOff()
```

manualPulse

```
p_Rig.MyInput.manualPulse()
```

Manually pulse the input.

Example

Pulse the input when entering a state.

```
def s_State_enter():
    p_Rig.MyInput.manualPulse()
```

setMute

```
p_Rig.MyInput.setMute(muted)
```

Mute the input so it can't trigger, or unmute it.

Inputs	Type	Description
muted	bool	Change the input mute status (True or False)

Status

Get information on the current state of the input.

isOn

```
p_Rig.MyInput.isOn()
```

Returns true if the input is currently true.

Example

When entering a state, check if an input is already true.

```
def s_state_enter():
    if p_Rig.MyInput.isOn():
        print('MyInput is on')
    else:
        print('MyInput is off')
```

isOff

```
p_Rig.MyInput.isOff()
```

Returns true if the input is currently false.

Example

When entering a state, check the status of the input.

```
def s_state_enter():
    if p_Rig.MyInput.isOff():
        print('MyInput is off')
    else:
        print('MyInput is on')
```

getStatusBits

```
p_Rig.MyInput.getStatusBits()
```

Read the current state of an input as a bitwise integer value. Bit order is:

`Fail | Pass | Active | Fall | Rise | True`

Used by the Pynapse polling loop.

iMn Input Settings

The iMn modules have analog inputs that are converted to logic signals. These functions override the analog-to-logic conversion settings at runtime. See iMn Input Processor for more information.

setProcLowPass

```
p_Rig.MyInput.setProcLowPass(hz)
```

Set the Lowpass Frequency, in Hertz. This is only available if Processing mode is Complex and 'Frequency Range' is not 'Unlimited'.

Inputs	Type	Description
hz	float	Set the lowpass frequency, in Hertz

setProcHighPass

```
p_Rig.MyInput.setProcHighPass(hz)
```

Set the Highpass Frequency, in Hertz. This is only available if Processing mode is Complex and 'Frequency Range' is not 'Unlimited'.

Inputs	Type	Description
hz	float	Set the highpass frequency, in Hertz

setProcGain

```
p_Rig.MyInput.setProcGain(v)
```

Set the input gain on the signal, in dB. This is only available if Processing mode is Simple or Complex. See iMn Input Processor for more information.

Inputs	Type	Description
v	float	Set the input gain, in dB

setProcThresh

```
p_Rig.MyInput.setProcThresh(v)
```

Set the input gain on the signal, in dB. This is only available if Processing mode is Simple or Complex. See iMn Input Processor for more information.

Inputs	Type	Description
v	float	Set the input gain, in dB

setProcHistReduce

```
p_Rig.MyInput.setProcHistReduce(v)
```

Set the input gain on the signal, in dB. This is only available if Processing mode is Simple or Complex. See iMn Input Processor for more information.

Inputs	Type	Description
v	float	Set the input gain, in dB

setProcSmooth

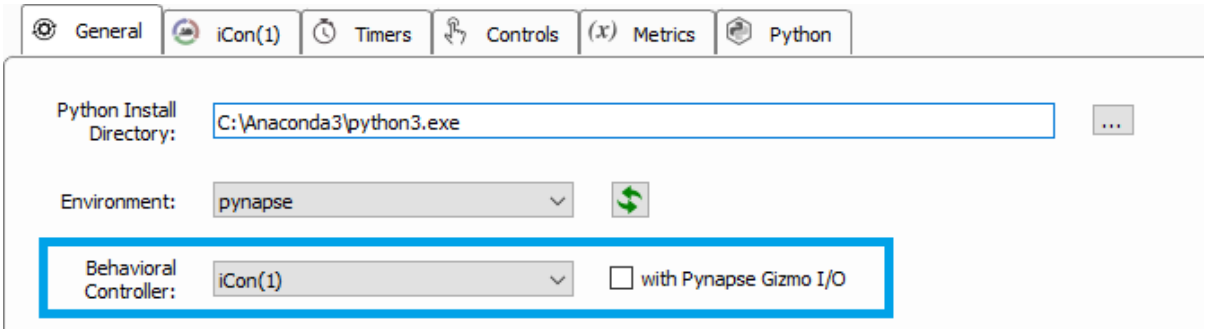
```
p_Rig.MyInput.setProcSmooth(ms)
```

Change the smoothing filter applied to the input signal before it goes through the logic conversion. This is only available if Processing mode is Complex. See iMn Input Processor for more information.

Inputs	Type	Description
ms	float	Time constant of the low-pass smoothing filter, in ms (0=off)

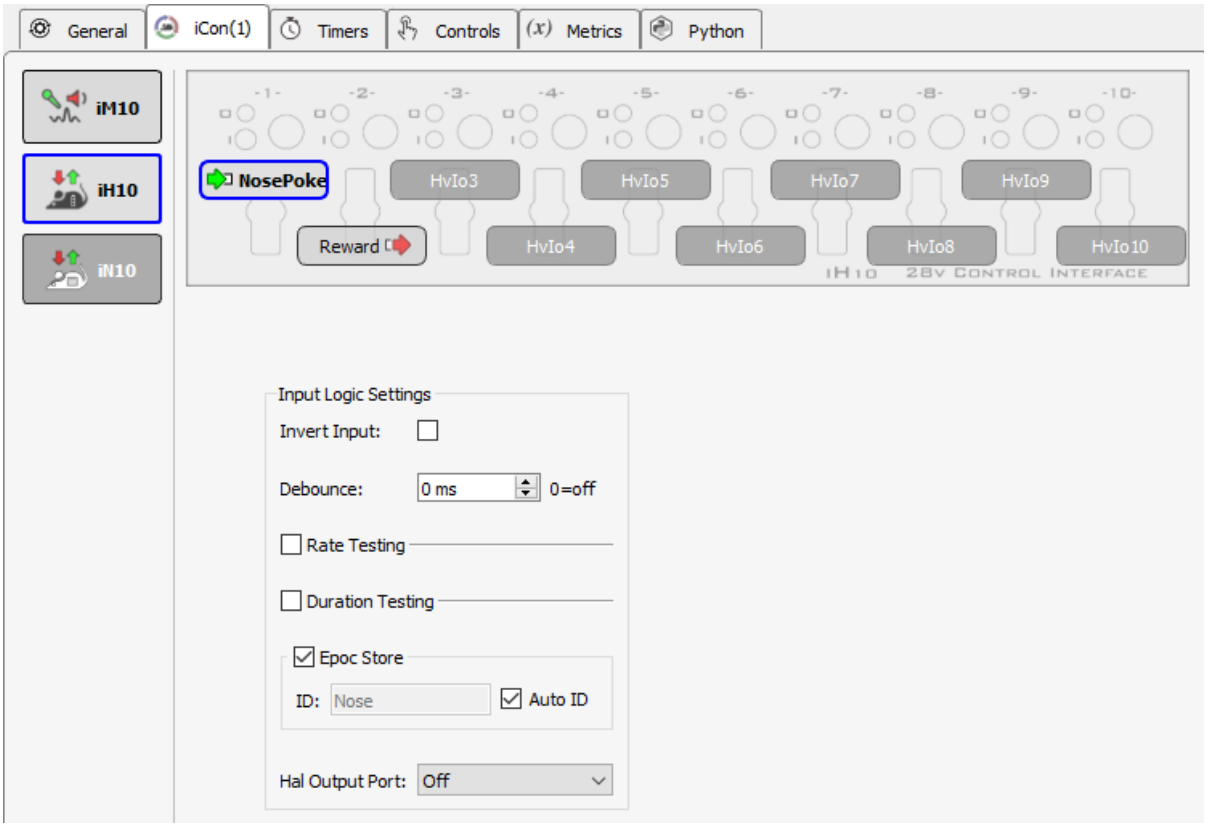
iCon Outputs

When an iCon is attached to the Pynapse Behavioral Controller in the General Tab, an additional iCon tab appears. This gives you a unified interface that lets you configure the iCon inputs/outputs directly within Pynapse and integrates them in the Python code editor.



iCon Tab

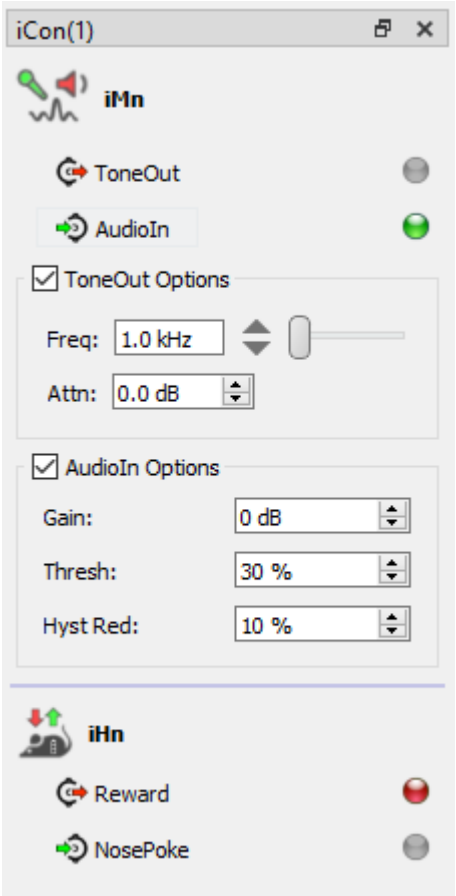
Configure the iCon inputs/outputs in the iCon tab. See Logic Output Processor in the Synapse Manual for information on setting up the iCon outputs how to convert Pynapse function calls to physical hardware events.



iCon Tab

Run-time Interface

The run-time interface has a button for each output and an LED indicator that shows the current state of the output. For the iMn and iS9, additional options that were available during design-time can also be modified during run-time.



iCon Run-time Interface

Click an output to manually toggle its state (on/off). Hold down CTRL and click an output to 'mute' it. This prevents Pynapse from triggering the output.

Output Methods

All output methods have the form `p_Rig.{OUTPUT_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you. The name of each method gets replaced with the name of your actual output, so if you name the output 'Reward' then `p_Rig.Reward.fire()` is an available method.

Manual Control

Manual turn outputs on, off, or fires a pulse waveform during runtime. Useful for stimulus/reward presentation.

fire

```
p_Rig.MyOutput.fire()
```

This is only available if Triggered Pulse is selected.

Quickly pulse the output. If **Duration** is non-zero, the output will stay high for that set duration. Set **Duration** to zero to trigger a single sample pulse.

Example

Trigger an output when the input goes high.

```
class Always:   #StateID = 0

    def s_MyInput_pass():
        p_Rig.MyOutput.fire()
```

turnOn

```
p_Rig.MyOutput.turnOn()
```

Turn the output on indefinitely. This is only available if Triggered Pulse is not selected.

Example

Link an input status to an output.

```
class Always:   #StateID = 0

    def s_MyInput_rise():
        p_Rig.MyOutput.turnOn()

    def s_MyInput_fall():
        p_Rig.MyOutput.turnOff()
```

turnOff

```
p_Rig.MyOutput.turnOff()
```

Turn the output off. This is only available if Triggered Pulse is not selected.

Example

Link an input status to an output.

```
class Always:   #StateID = 0

    def s_MyInput_rise():
        p_Rig.MyOutput.turnOn()

    def s_MyInput_fall():
        p_Rig.MyOutput.turnOff()
```

setMute

```
p_Rig.MyOutput.setMute(muted)
```

Mute the output so it can't trigger, or unmute it.

Inputs	Type	Description
muted	bool	Change the output mute status (True/False or 1/0)

Duration Settings

setDuration

```
p_Rig.MyOutput.setDuration(dur_sec)
```

Override the output **Duration** setting. This is only available if Triggered Pulse is enabled and **Duration** is greater than 0.

Inputs	Type	Description
dur_sec	float	Duration of the output pulse when triggered with `fire`, in seconds

Example

Modify the pulse shape and output value based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, decrease the output pulse time by 50 ms.
    if p_Metric.success.read() > 5:
        p_Metric.pulse_dur.dec(delta=0.05)
        p_Rig.MyOutput.setDuration(p_Metric.pulse_dur.read())
```

Status

Get information on the current state of the output.

isOn

```
p_Rig.MyOutput.isOn()
```

Returns true if the output is currently true.

Example

When entering a state, check if an output is already true.

```
def s_state_enter():
    if p_Rig.MyOutput.isOn():
        print('MyOutput is on')
    else:
        print('MyOutput is off')
```

isOff

```
p_Rig.MyOutput.isOff()
```

Returns true if the output is currently false.

Example

When entering a state, check the status of the output.

```
def s_state_enter():
    if p_Rig.MyOutput.isOff():
        print('MyOutput is off')
    else:
        print('MyOutput is on')
```

iMn Output Settings

The iMn has analog outputs. These functions override the analog output settings at runtime. See iMn Analog Outputs for more information.

setAtten

```
p_Rig.MyOutput.setAtten(v)
```

Set the output Attenuation, in dB. This is only available if Waveform Shape is User, Tone, White Noise, Pink Noise, Square, Clock, or PWM.

Inputs	Type	Description
v	float	Set the output attenuation, in dB

setFreq

```
p_Rig.MyOutput.setFreq(v)
```

Set the output Frequency, in Hertz. This is only available if Waveform Shape is Tone, Square, or Clock.

Inputs	Type	Description
v	float	Set the output frequency, in Hz

setVolt

```
p_Rig.MyOutput.setVolt(v)
```

Set the output voltage, in Volts. This is only available if Waveform Shape is DC Voltage.

Inputs	Type	Description
v	float	Set the output voltage, in V

setDutyCycle

```
p_Rig.MyOutput.setDutyCycle(v)
```

Set the duty cycle percentage, from 0 to 100. This is only available if Waveform Shape is PWM.

Inputs	Type	Description
v	float	Set the output period, in ms

Deprecated after v96

setPeriod

```
p_Rig.MyOutput.setPeriod(v)
```

Set the output period, in ms. This is only available if Waveform Shape is PWM.

Inputs	Type	Description
v	float	Set the output period, in ms

setWidth

```
p_Rig.MyOutput.setWidth(v)
```

Set the output width, in ms. This is only available if Waveform Shape is PWM.

Inputs	Type	Description
v	float	Set the output width, in ms

iS9 Output Settings

The iS9 sends a stimulation current. These functions override the output settings at runtime. See iS9 Stim Outputs for more information.

setStimCurrent

```
p_Rig.MyOutput.setStimCurrent(v)
```

Set the output current, in mA. This value ranges from 0.1 to 2.5 mA.

Inputs	Type	Description
v	float	Set the output current, in mA

States

Pynapse has an internal state machine that logs all events and keeps track of the current state. New triggers coming in are filtered through this state machine, so that only the slot methods associated with the current state can run.

There is a special 'Always' state - slot methods in the Always state can trigger on any polling loop. This is a useful state to add user mode controls (start/pause/stop).

States are 'classes' in the Python code that have the special comment `#StateID = ?` at the end of the class definition. The number defined here is the id number associated with this state. This value will be timestamped and stored with the data in the data tank. If the parser finds a `?`, it will automatically assign a number for you. Otherwise enter an integer to lock the StateID in place, for example `#StateID = 555`.

See Working with StateIDs for more information.

Slot Methods for Responding to State Changes

These state slots capture state machine changes. They are available as method definitions inside Pynapse states, including the Always state. Write a method with this name to react to the corresponding event.

Slot name	Event
s_State_change	Triggers on any state change
s_State_enter	Triggers once when the state begins
s_State_exit	Triggers once when the state ends
s_State_timeout	Triggers when the state timeout is reached

Example

Turn on an output only while in a particular state.

```
class StartTrial:    # StateID = ?
    # turn on MyOutput when entering state
    def s_State_enter():
        p_Output.MyOutput.turnOn()

    # when MyInput passes 'Time to Active', switch to ActiveState
    def s_MyInput_active():
        p_State.switch(ActiveState)

    # turn off MyOutput when exiting state
    def s_State_exit():
        p_Output.MyOutput.turnOff()

class ActiveState:    # StateID = ?
    # when MyInput passes 'Time to Pass', switch to PassState
    def s_MyInput_pass():
        p_State.switch('PassState')

    def s_MyInput1_fail():
        p_State.switch(FailState)
```

In this example, use s_State_change() to track order of state execution. Suppose there are many states that exit to TargetState. If the state that exited to TargetState is TargetOldState, we want to do something.

```
class Always:    # StateID = 0
    def s_State_change(newstateidx, oldstateidx):
        print('new state', newstateidx, 'old state', oldstateidx)
        if newstateidx == TargetState and oldstateidx == TargetOldState:
            print('do something')
```

State Timeouts

The Pynapse state machine has a built-in Timer that is used as a timeout within the current state that moves to another state if it fires. Timeouts are usually set in the s_State_enter() slot method but can be set anywhere in the State. Timeouts can also be canceled.

Example

If the user fails to press a button (MyInput) within five seconds, we want to move to a NoTrial state and wait there for ten seconds before starting a new trial.

```
class StartTrial:    # StateID = ?

    # if no input is received after 5 seconds, switch to NoTrial state
    def s_State_enter():
        p_State.setTimeout(5, NoTrial)

    def s_MyInput_rise():
        p_State.switch(TrialState)

class NoTrial:    # StateID = ?
    # wait 10 seconds, return to StartTrial
    def s_State_enter():
        p_State.setTimeout(10, StartTrial)

class TrialState:    # StateID = ?
    # turn on an output
    def s_State_enter():
        p_Output.MyOutput.turnOn()
```

Methods

All state methods have the form `p_State.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

State Control

switch

```
p_State.switch(newstate)
```

Tell Pynapse to move to a new state. All states are python 'classes'. The input to `switch` can be either the class or the string class name you want to switch to. See the example below

Inputs	Type	Description
newstate	class or string	Name of new class to switch to

Note

The function that contains the `switch` command does not exit immediately after switching the internal state. This can have unintended consequences, particularly if you are using the **Sync to State Change** option for outputs or timers. Best practice is to use the `switch` command last, right before the function exits.

See Synchronizing Events for information.

Example

Switch between states when MyInput goes high.

```
class StartTrial:    # StateID = ?

    def s_MyInput_active():
        p_State.switch(ActiveState)

class ActiveState:    # StateID = ?
    def s_MyInput_pass():
        # you can also switch states with a string name
        p_State.switch('PassState')

    def s_MyInput_fail():
        p_State.switch(FailState)
```

setTimeout

```
p_State.setTimeout(secs, stateOnTimeout)
```

Switch to a default state after a certain period of time.

Inputs	Type	Description
secs	float	Timeout duration in seconds
stateOnTimeout	class or string	New of the class to switch to

Important

There can only be one active timeout per state. If you need to set a new timeout within the state, use the cancelTimeout method first.

Example

Toggle between the FirstState and SecondState until MyInput rises in FirstState.

```
class FirstState:    # StateID = ?

    # if no input is received in 5 seconds, switch to SecondState
    def s_State_enter():
        p_State.setTimeout(5, SecondState)

    def s_MyInput_rise():
        p_State.switch(EndState)

class SecondState:    # StateID = ?

    # wait 5 seconds, return to FirstState
    def s_State_enter():
        p_State.setTimeout(5, FirstState)

class EndState:    # StateID = ?
    def s_State_enter():
        print('done')
```

cancelTimeout

```
p_State.cancelTimeout()
```

Cancel the current timeout. Can be called anywhere within the state.

Example

Give the subject 15 seconds to press MyInput 10 times. If successful, cancel the state timeout and give the subject unlimited time to reach 20 presses before moving to the success state.

```
class TrialState:    # StateID = ?
    def s_State_enter():
        # reset counter
        p_Global.count.write(0)

        # if target isn't reached in 15 seconds switch to DefaultState
        p_State.setTimeout(15, DefaultState)

    def s_MyInput_rise():
        # increment counter
        p_Global.count.inc()

        # if we reached our first target, cancel timeout
        if p_Global.count.read() == 10:
            p_State.cancelTimeout()
        elif p_Global.count.read() == 20:
            p_State.switch(SuccessState)

class DefaultState:    # StateID = ?
    def s_State_enter():
        print('default')

class SuccessState:    # StateID = ?
    def s_State_enter():
        print('success')
```

Status

isCurrent

```
p_State.isCurrent(stname)
```

Check if the current state is the given name. This is useful if you have a lot of States defined but want to do similar actions in multiple states for a given slot method. You can move the logic into the Always state and avoid repeating yourself. See the example below.

Or if you want to

Inputs	Type	Description
stname	class or string	Name of state to check

Example

In a long list of states, we want to turn the MyOutput on in just two of them.

```
class Always:   #StateID = 0
    def s_MyInput1_rise():
        if p_State.isCurrent(State8) or p_State.isCurrent(State20):
            p_Output.MyOutput.turnOn()
```

In this second example, the target state is dynamically set by a global variable. When MyInput2 turns on, the slot method is captured in the Always state and only continues (turns on Output2) if the current state matches the target state. This target state can be set on the user interface or somewhere else in the code using the Globals asset. This could also be tied to a Control asset.

```
    class Always:   #StateID = 0
        def s_MyInput2_rise():
            if p_State.isCurrent(p_Global.target_state.read()):
                print('current state is the target state set in the user interface')
                p_Output.MyOutput2.turnOn()
```

isNotCurrent

```
p_State.isNotCurrent(stname)
```

Check if the current state is not given name. This is useful if you have a lot of States defined but don't want to include the same identical slot method in all of them except a small number of states. You can include this logic check within the Always state. See the example below.

Inputs	Type	Description
stname	class or string	Name of state to check

Example

When MyInput turns on, turn on MyOutput in all states unless we're in the DontStim state.

```
class Always:   #StateID = 0
    def s_MyInput_rise():
        if p_State.isNotCurrent(DontStim):
            p_Output.MyOutput.turnOn()
```

Timers

Timers are used independently of states to control program flow or stimulus presentation.

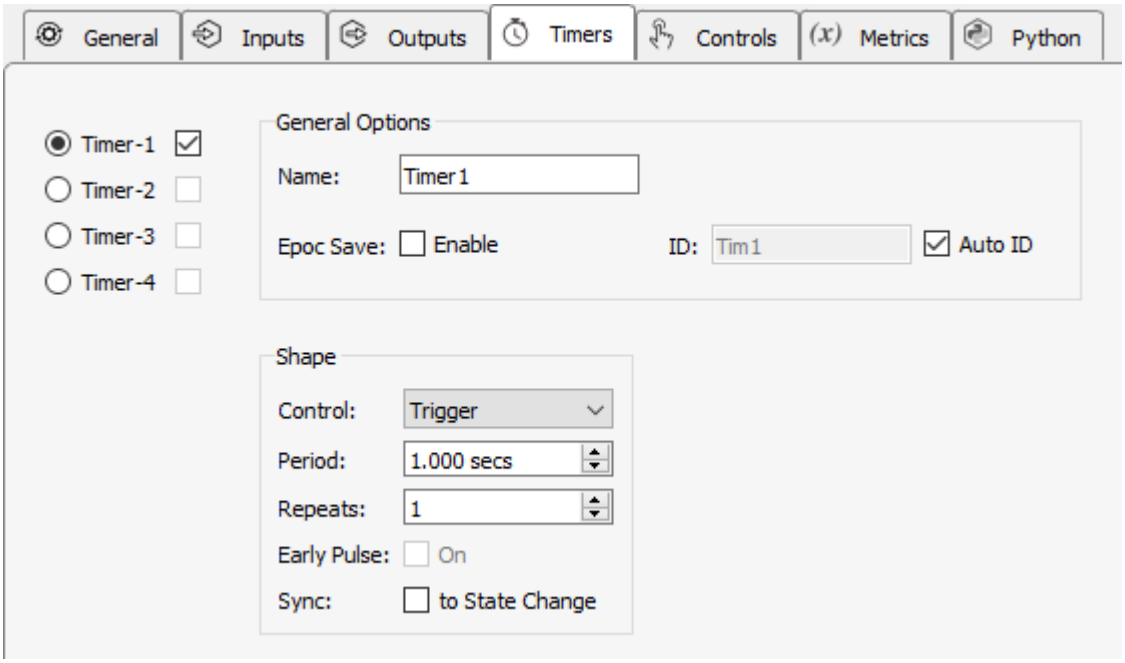
The period and number of repeats can be modified at runtime within the Python code.

Control Modes

There are two **Control** types, Trigger and Enable.

In **Trigger** mode, you initiate the timer with the `start` method and it runs until it has reached the set number of **Repeats** (or indefinitely if Repeats is set to -1).

In **Enable** mode, you turn the timer on and off with the `turnOn` / `turnOff` method pairs.



Timers Tab

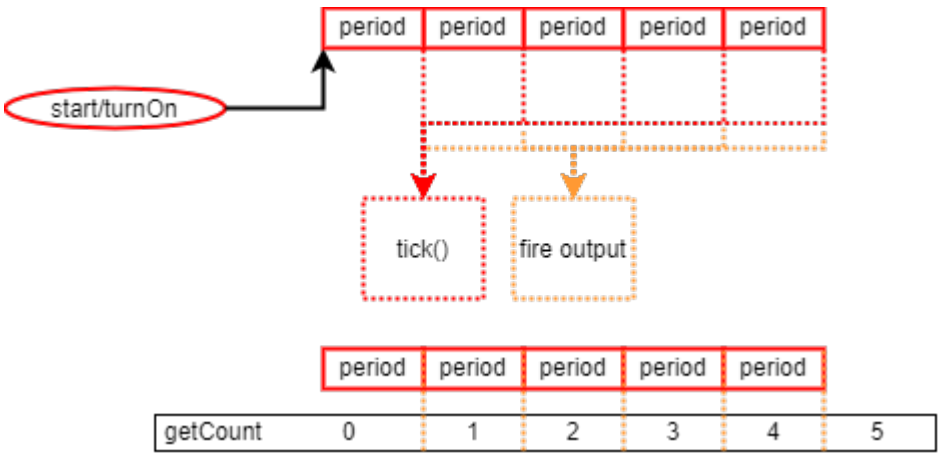
See Synchronizing Events for information on the **Sync to State Change** option.

Pulse Control

The Timer can operate in two modes, `Standard` and `Early Pulse`, depending on the state of the `Early Pulse` checkbox.

Standard

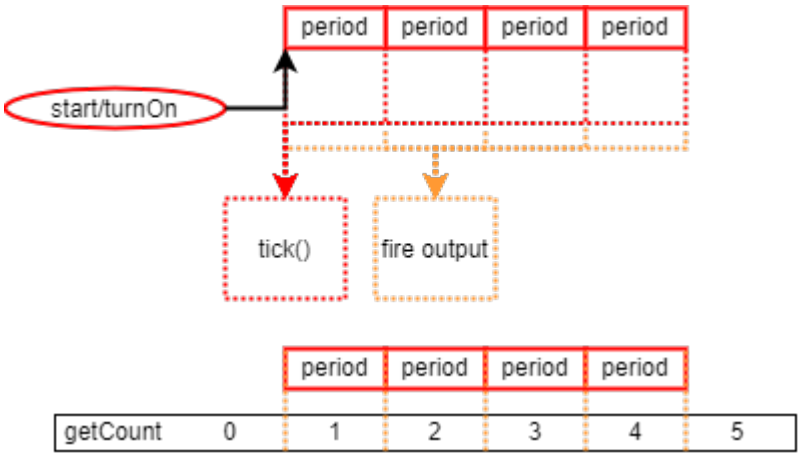
In standard mode, the `tick` event and the timer output fires after one timer period has passed. Use this to time events starting from when you first turn the timer on with the `start` or `turnOn` method.



Standard Flow Diagram

Early Pulse

In Early Pulse mode, the tick event and the timer output fires immediately after you call the `start` or `turnOn` method (or when the next state change happens if **Sync to State Change** is enabled). Use this to start timed stimulus presentations immediately.



Early Pulse Flow Diagram

> **Note**
>
> In Trigger mode, `Repeats` must have a value greater than 1 to enable the `Early Pulse` checkbox.

Slot Methods for Responding to Timer Ticks

These slot methods capture status information about the timers. They are available as method definitions inside Pynapse states for each timer. Write a method with this name to react to the corresponding event.

Slot name	Event
s_MyTimer1_tick	Fires on each tick of `MyTimer1`. See Pulse Control for the flow diagram

The timer slot method has the form `s_{TIMER_NAME}_tick`. Type `def s_` in the Pynapse Code Editor and let the code completion do the work for you.

Example

Set a timer that fires once per second for 10 seconds, and print the current timer count when it fires

```
class Always:    # StateID = 0
    def s_Mode_standby():
        p_Timer.MyTimer.setPeriod(1)
        p_Timer.MyTimer.setRepeats(10)

    def s_Mode_recprev():
        p_Timer.MyTimer.turnOn()

    def s_Timer1_tick(count):
        print(count)
```

In this example, Synapse stops the recording after 10 seconds and switches to Idle.

```
class Always:    # StateID = 0
    def s_Mode_standby():
        p_Timer.MyTimer.setPeriod(10)
        p_Timer.MyTimer.setRepeats(1)

    def s_Mode_recprev():
        p_Timer.MyTimer.turnOn()

    def s_Timer1_tick(count):
        print('done')
        syn.setModeStr('Idle')
```

Methods

All control methods have the form `p_Timer.{TIMER_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

Setup

setPeriod

```
p_Timer.MyTimer.setPeriod(period_sec)
```

Set the period between timer ticks.

Inputs	Type	Description
period_sec	number	New period, in seconds

setRepeats

```
p_Timer.MyTimer.setRepeats(reps)
```

Set the number of ticks before the timer finishes, when timer is in Trigger mode. Can be -1 to run indefinitely.

Inputs	Type	Description
reps	number	Number of times to repeat the timer tick

Control

turnOn

```
p_Timer.MyTimer.turnOn()
```

Start a timer that has **Control** mode set to Enable. This call is required to start any timer in Enable mode.

turnOff

```
p_Timer.MyTimer.turnOff()
```

Stop a timer prematurely. This only works with timers that have **Control** mode set to Enable.

start

```
p_Timer.MyTimer.start()
```

Start a timer that has **Control** mode set to Trigger. This call is required to start any timer in Trigger mode.

Status

Get information on the current state of the timer.

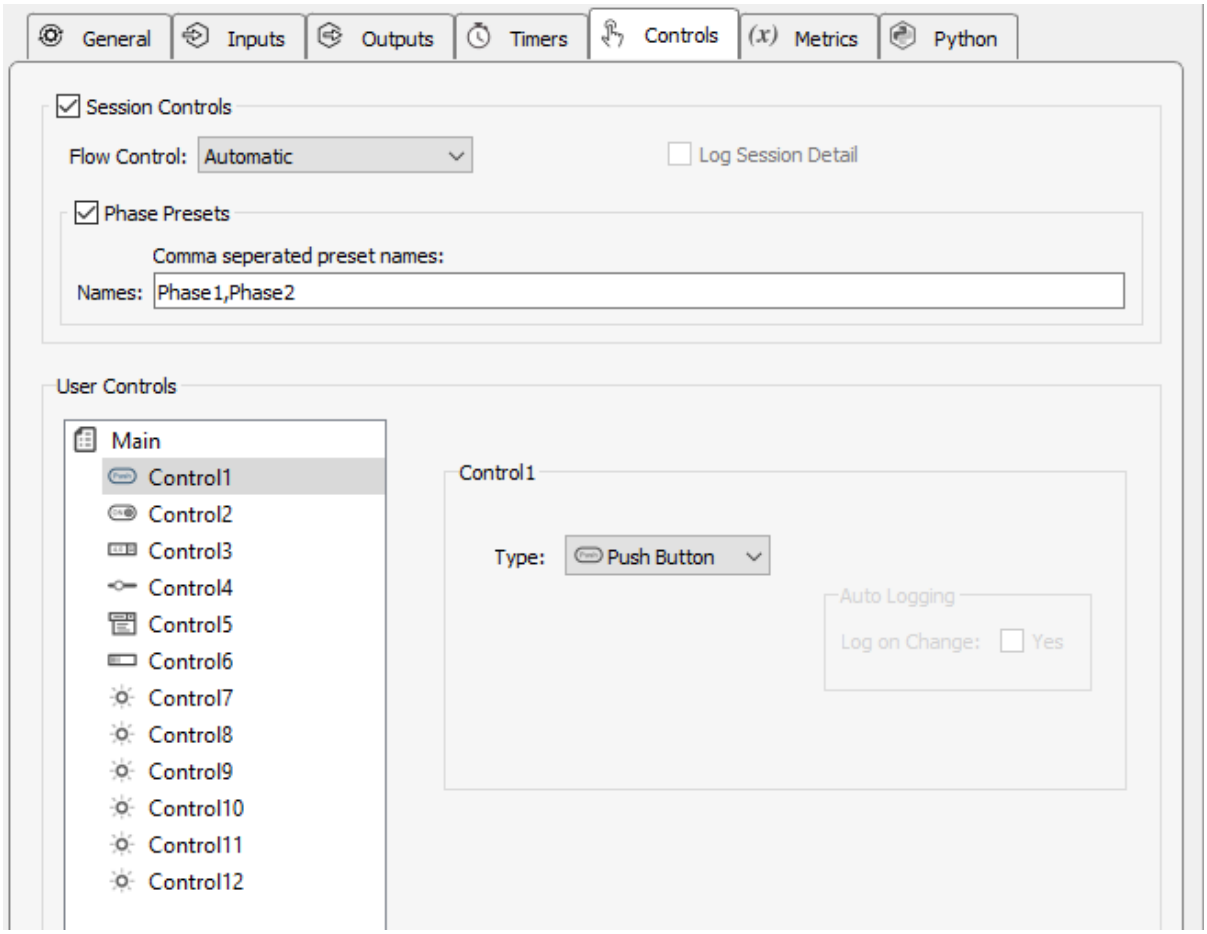
getCount

```
n_ticks = p_Timer.MyTimer.getCount()
```

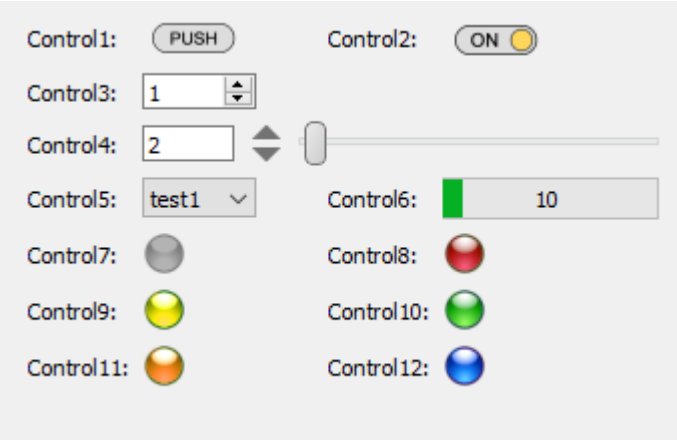
Returns the number of times the timer has fired. See Pulse Control for a flow chart example.

Controls

You can create several kinds of run-time widgets and read/ write the values of the widget during the experiment. Controls will issue a trigger when their value is changed. This event is captured in the Pynapse event loop. You can also read the value of the controls inside any Python method.

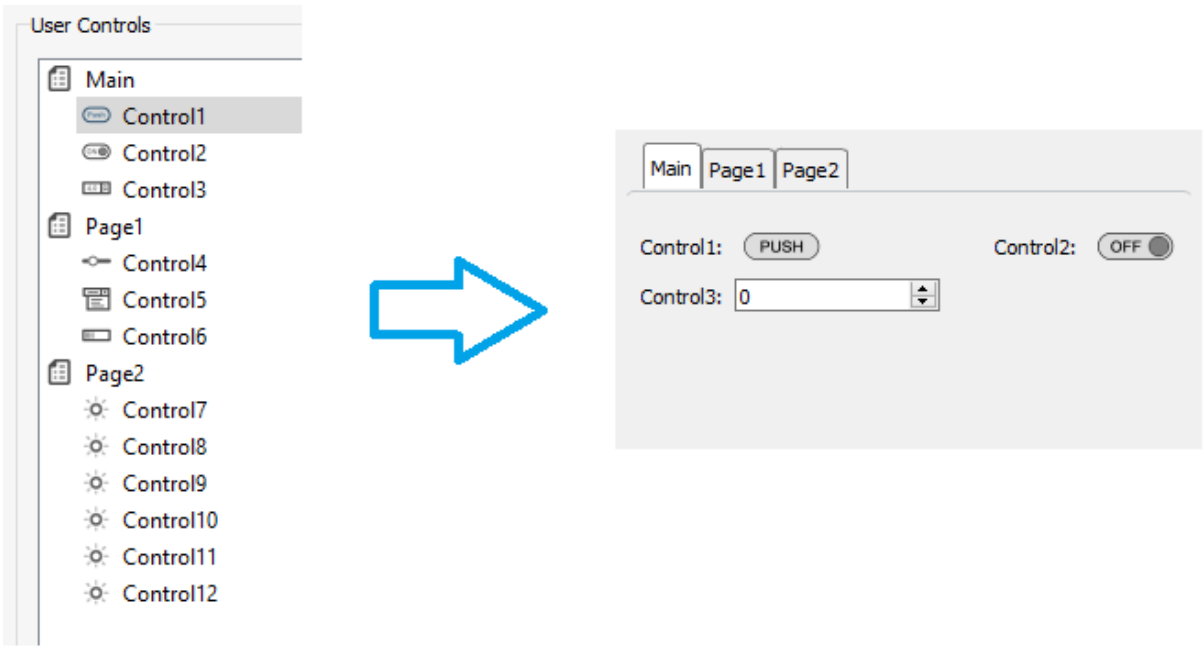


Controls Tab



Runtime Tab Showing Example Controls

You can add up to 50 custom controls. Right-click on the 'Main' page to add new controls. Drag and drop the controls in the tree to set the order they are displayed at runtime. Right-click on 'Main' to add a new page of controls. Controls will be organized into tabs at runtime.

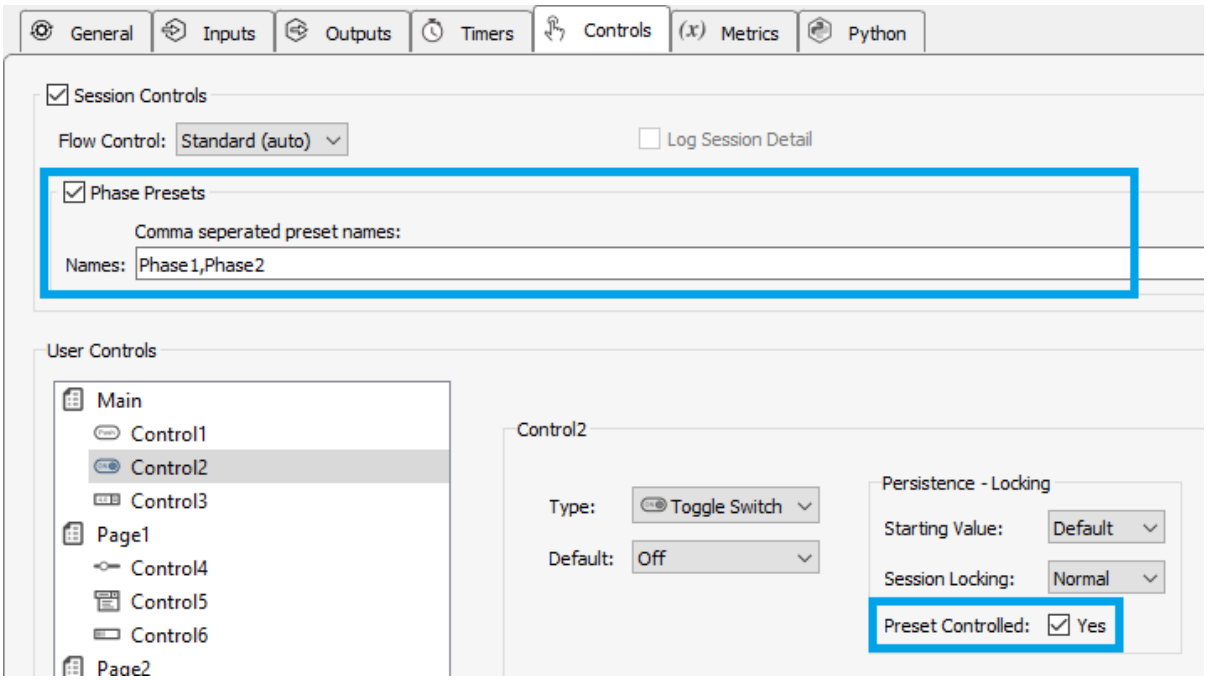


Runtime Control Tab Example

Phase Presets

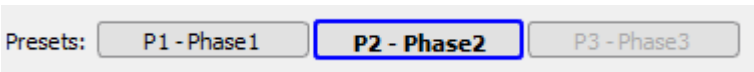
User-adjustable controls (Toggle Switch, Spin Control, Slider Control, and Combo Box) can be assigned preset values for different phases of the experiment, so you can quickly change several controls with the click of a button. When combined with Session Controls you can tell the Pynapse scheduler how many trials / blocks of each phase you want to run and it automatically handles it for you. The Session Scheduler takes this one step further and allows you to schedule the sequence of phase preset conditions to run in order.

Check the **Phase Presets** box to enable this feature and make a comma-separated list with your own custom phase name. Check the **Preset Controlled** option on any control you want to add to the presets.



Assign controls to phases

At run-time, the phase presets appear as buttons. Setup the control values that you want, then right-click on one of the phase buttons and select 'Store To Preset' to assign those values to that button. The button text changes to black to indicate preset values have been assigned, and the button outline changes to blue to indicate it is currently selected.



Phase buttons: assigned, selected, and undefined

Right-click on a phase button for more options:

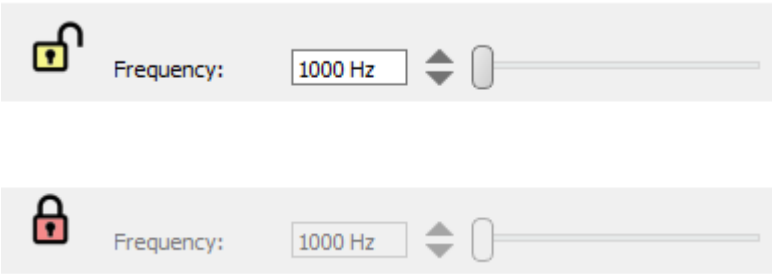
Option	Description
Clear Preset	Clear the selected preset, leaving it undefined
Store to Preset	Overwrite the selected phase values with the current control settings
Load from Preset	Overwrite the current control values with the selected phase values

Locking

Choose when to lock controls during the experiment.

Note

The Lock icon next to the icon at runtime must be locked in order for this setting to take effect. The Lock icon is unlocked by default in `Preview` mode, and locked by default in `Record` mode.



Slider control in unlocked and locked state

Option	Description
Normal	Control value is locked when the Lock icon is locked and Pynapse is in an active Session, so it can't be adjusted by the user during critical moments of the experiment
Never	Control value can always be adjusted by the user at any time during the experiment, regardless of Lock icon state
Always	Control value can't be changed at any time the Lock icon is locked

Slot Methods for Responding to Control Changes

This control slot method captures status information about the controls. It is available as method definitions inside Pynapse states for each control. Write a method with this name to react to the corresponding event.

Slot name	Event
s_MyControl_change	MyControl value changed

All custom control slot methods `s_{CONTROL_NAME}_change`. Type `def s_` in the Pynapse Code Editor and let the code completion do the work for you.

Example

Print a slider value when it is changed at runtime.

```
class Always:    # StateID = 0

    # capture any control value change with this
    def s_MyControl_change(value):
        print('new control value', value)
```

Methods

All control methods have the form `p_Control.{CONTROL_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

Status

read

```
value = p_Control.MyControl.read()
```

Read the current value of the control. For combo box controls, the value is the index into the list of items in the combo box.

Example

Set the next stimulation based on a slider value controlled by the user at runtime.

```
class PrepStim:   #StateID = 0
    def s_State_enter():
        # get next stim ready
        wave_freq = p_Control.MyControl.read()
        p_Param.p_Param.WaveFreq_write(wave_freq)
```

Control

write

```
p_Control.MyControl.write(val)
```

Write a new value to the control. For combo box controls, the value is the index into the list of items in the combo box. For Led Indicators, the value is the index into the list of colors.

Inputs	Type	Description
val	number	New value of the control

Example

Increment a progress bar.

```
class EndTrial:   #StateID = 0
    def s_State_enter():

        # increment completed trials counter
        p_Metric.completed_trials.inc()

        # update progress bar
        progress = 100 * p_Metric.completed_trials.read() / p_Metric.desired_trials.read()
        p_Control.MyProgressBar.write(progress)
```

lock

```
p_Control.MyControl.lock()
```

Lock the control to prevent modification.

Example

Lock control during a portion of the experiment

```
class StartTrial:   #StateID = 0
    def s_State_enter():
        p_Control.MyControl.lock()
```

unlock

```
p_Control.MyControl.unlock()
```

Unlock the control to allow modification.

Example

Unlock control during a portion of the experiment

```
class EndTrial:   #StateID = 0
    def s_State_enter():
        p_Metric.completed_trials.inc()
        p_Control.MyControl.unlock()
```

setRange

```
p_Control.MyControl.setRange(minv, maxv)
```

Set the value range of the control between minv and maxv. This is only valid for Spin Control, Slider Control, and Progress Bar controls.

Inputs	Type	Description
minv	number	New minimum value for this control
maxv	number	New maximum value for this control

setLabel

```
p_Control.MyControl.setLabel(txt)
```

Sets the text label of the control.

Inputs	Type	Description
txt	string	New label text

hide

```
p_Control.MyControl.hide()
```

Hides the control on the user interface.

show

```
p_Control.MyControl.show()
```

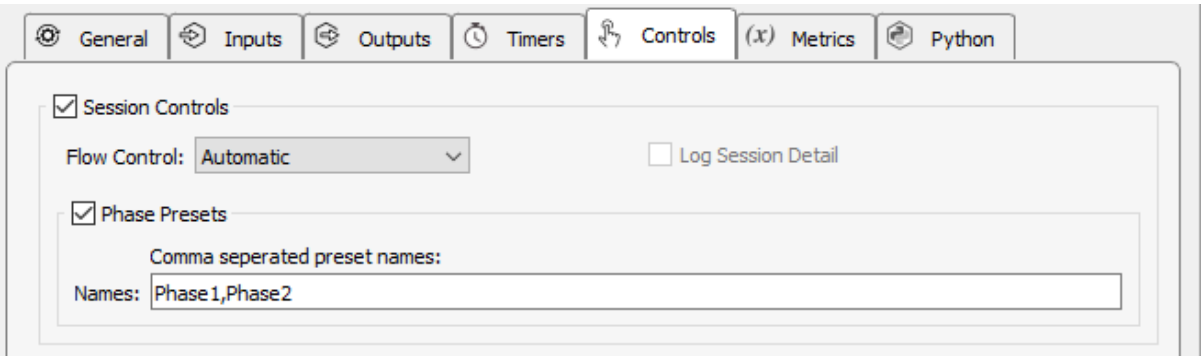
Shows the control on the user interface.

Sessions

Session Mode Controls

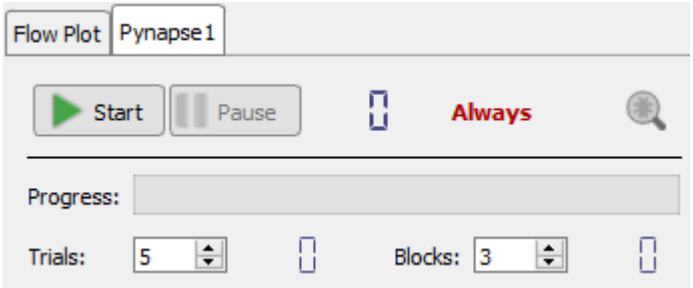
Session Mode Controls allow you to schedule the number of trials and blocks to run. It triggers the trial, block, and session Metrics to save, display, and plot. It also includes a Scheduler which can automatically present different phases of the experiment with minimal user interaction.

Enable Session Controls in the Controls tab.



Session Control Options

The Session Mode Controls add a run-time interface to start, stop, pause, resume the session. Scheduler controls also appear at run-time. A session counter, block counter, and trial counter are also on the interface.



Session Runtime Controls

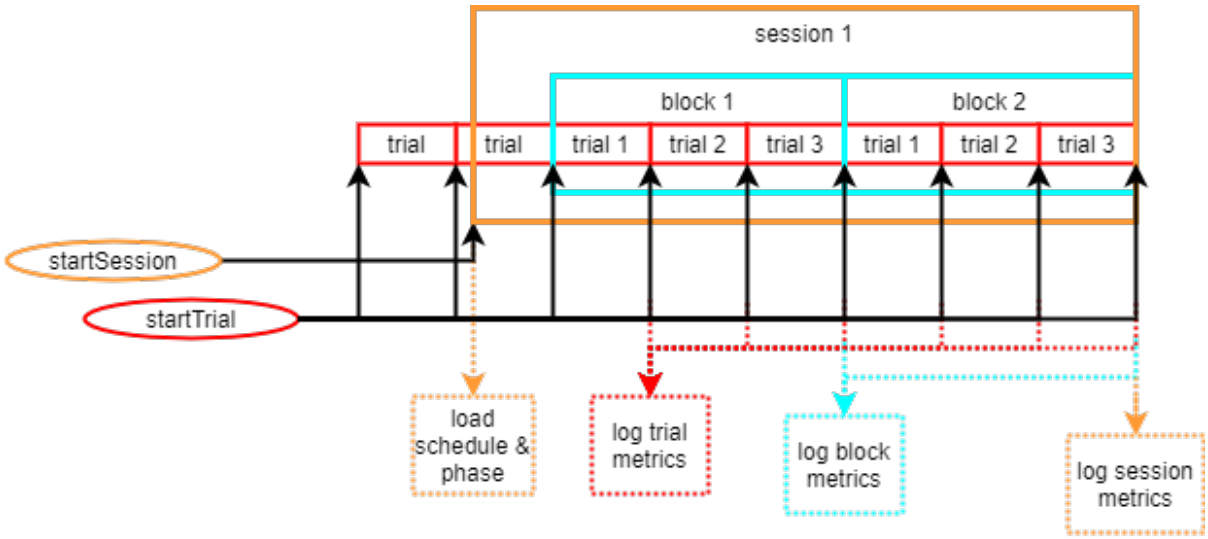
See Slot Methods for Responding to Session Changes for an example of how these might be used in your Python code.

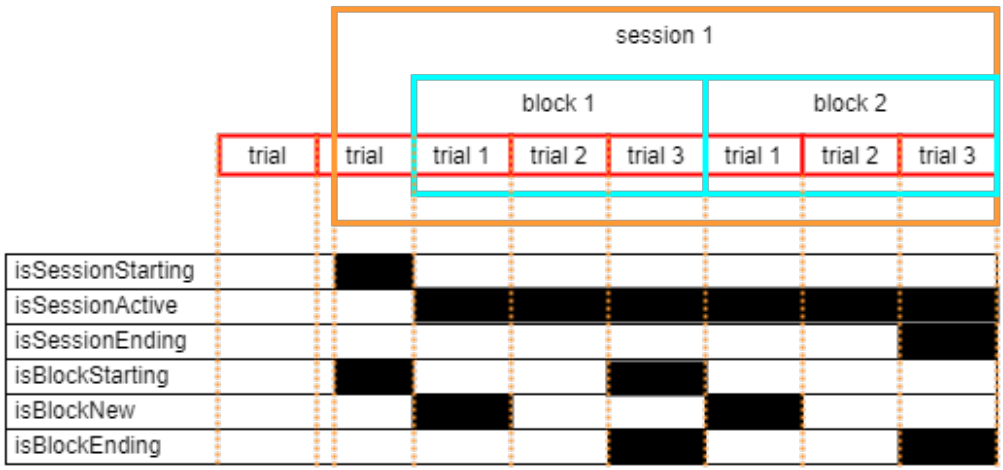
Flow Control

The three Flow Control methods are described below. The diagrams show the session flow based on when the python Trial Control methods (colored ovals) are called. The tables show when the Status information methods return true (black in the timeline).

Automatic

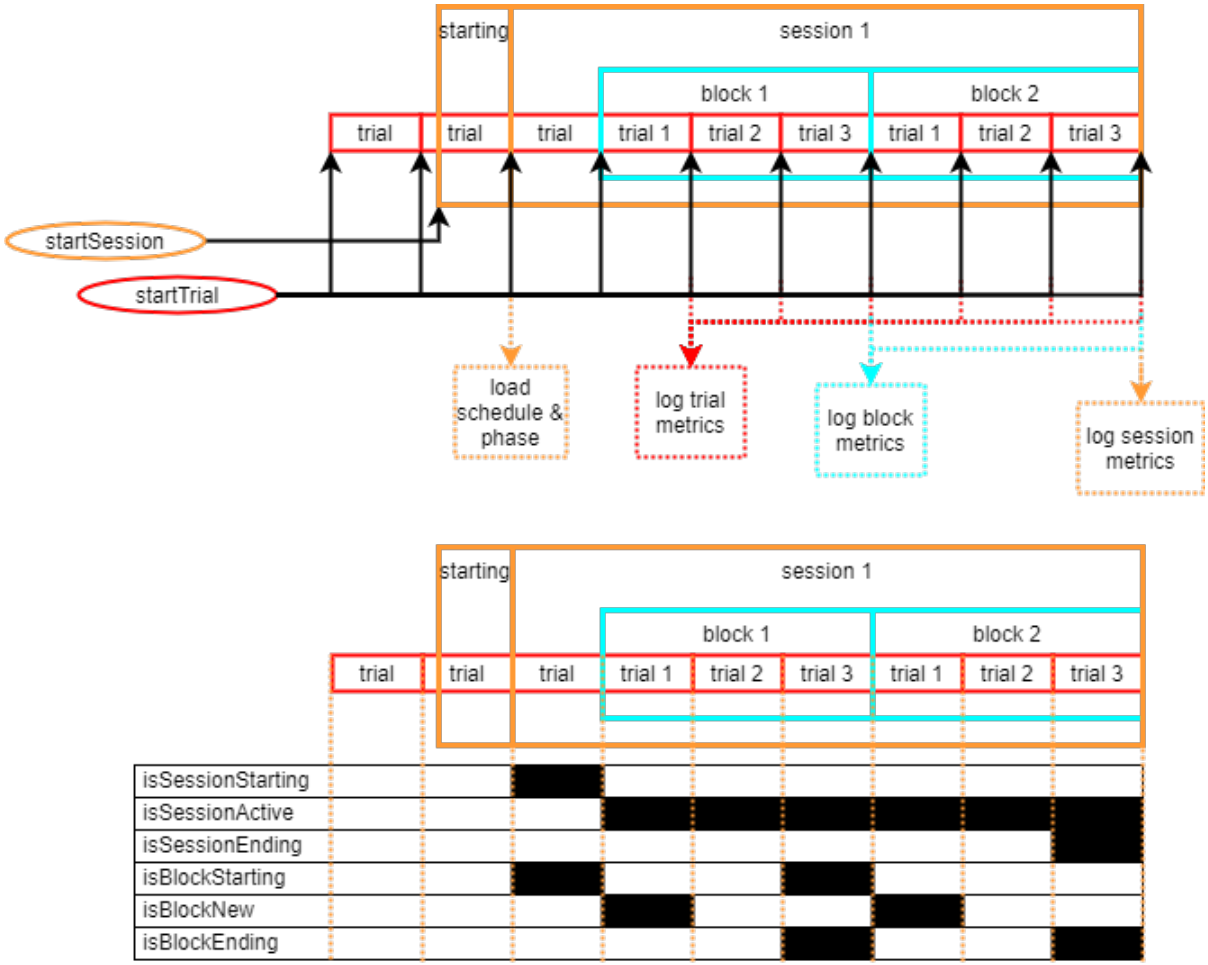
In **Automatic** mode, the session starts immediately when you click the 'Start' button. You only need to call `p_Session.startTrial()` in your Python code during an active session and the scheduler will automatically increment the trial and block counters for you, based on the number of trials that have already occurred.





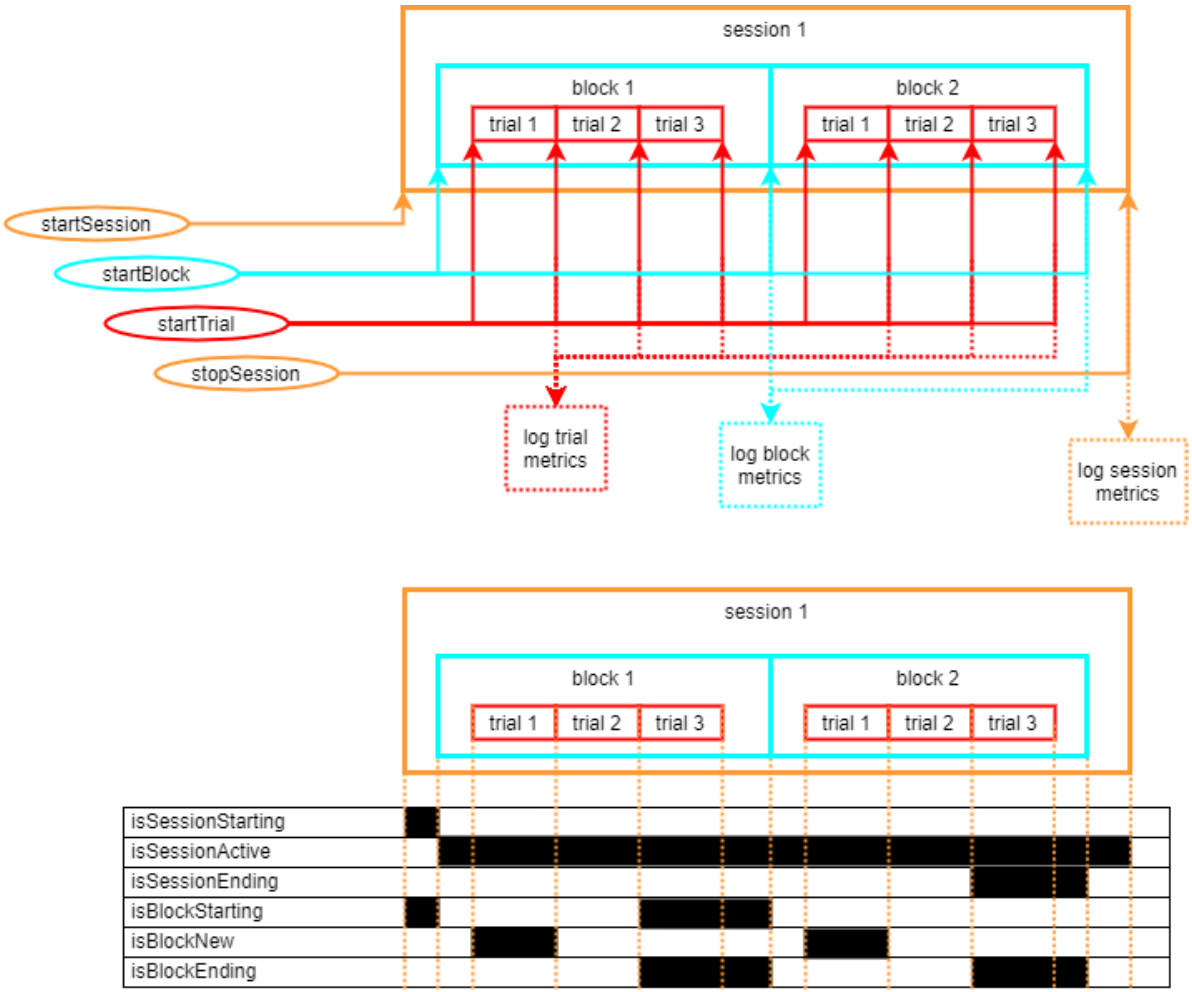
Automatic with SyncDelay

In **Automatic with SyncDelay** mode, the phase is loaded when the current trial completes and the session starts after one full trial has finished after that. You only need to call `p_Session.startTrial()` in your Python code during an active session and the scheduler will automatically increment the trial and block counters for you, based on the number of trials that have already occurred.

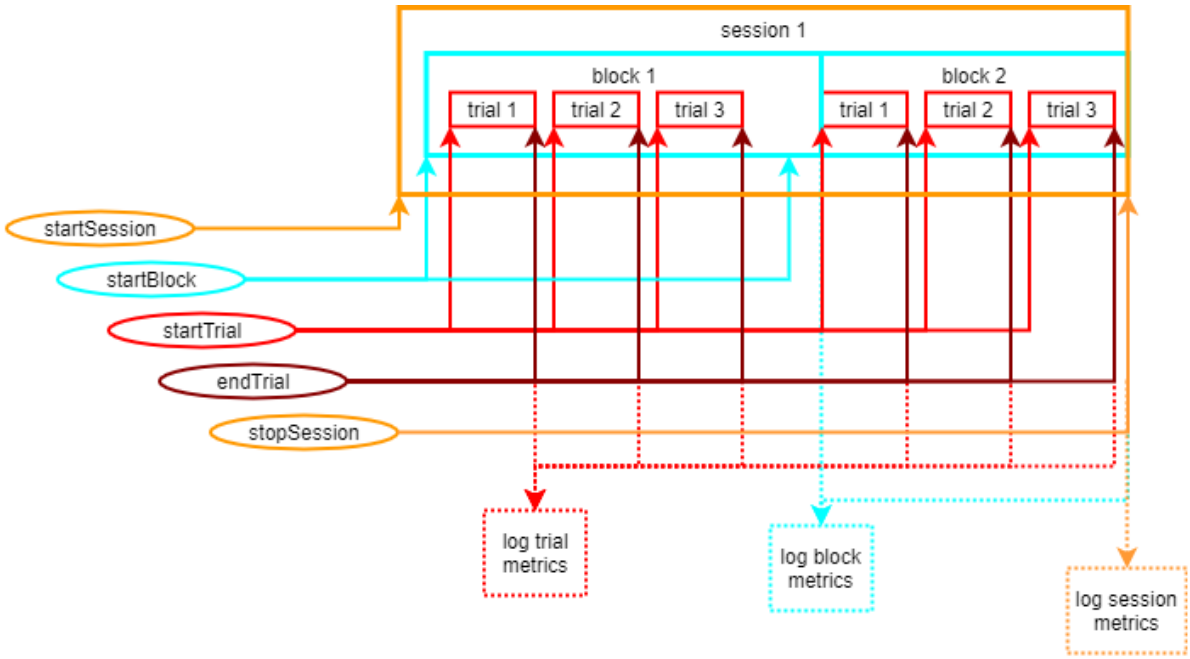


Manual

In **Manual** mode, you manually call `p_Session.startBlock()` and `p_Session.startTrial()` in the Python code during an active session to advance the trial and block counters. You can use the mode controls at the top of the runtime interface or call `p_Session.startSession()` and `p_Session.stopSession()` in the Python code.



You can also optionally call `p_Session.endTrial()` to add a gap between trials. This call ends the trial and logs the trial metrics before starting the next trial. You can use this time to make a decision before the next trial begin.



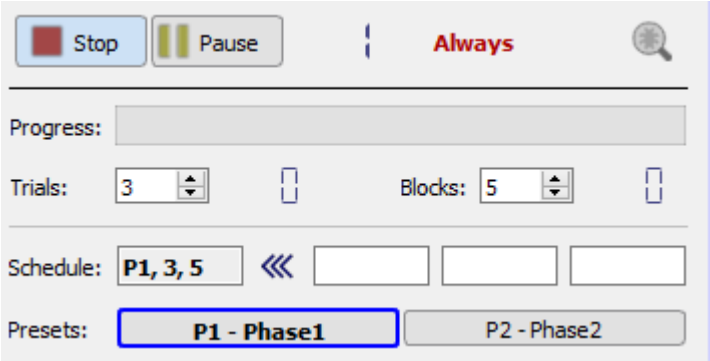
Note

isBlockStarting and **isBlockEnding** turn on when the trial count is reached.

isSessionEnding turns on when the block count and trial count are reached.

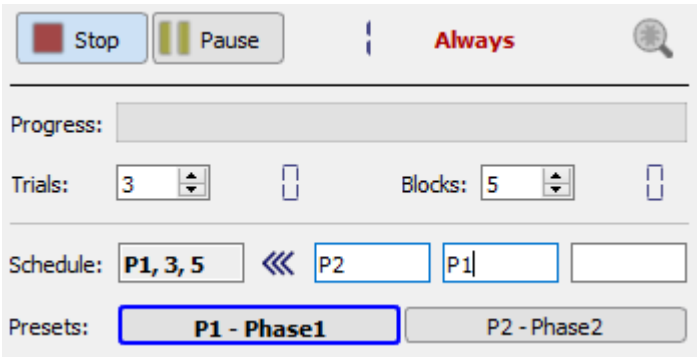
Scheduler

If you are using one of the 'Automatic' flow control methods and are using Phase Presets, then a run-time Scheduler manages the experiment flow. It is programmed with a list of phases and the number of trials and blocks to run for each phase.



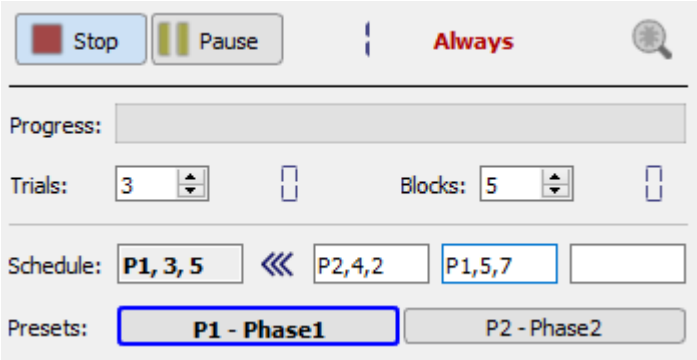
Session Scheduler Controls

Enter the Phase ID (e.g. P1, P2) into the input boxes to set the scheduler. When the specified number of blocks and trials for the current phase has finished, the scheduler will automatically advance to the next phase in the list.



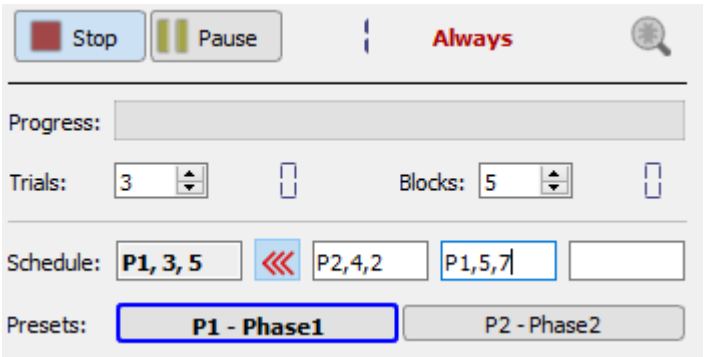
Session Scheduler Phase List

Enter the Phase ID, number of trials, and number of blocks as a comma-separated list to override the default trial and block count.



Session Scheduler Trial and Block Specification

Click the arrow button to end the current phase and start the next phase immediately on the next trial. The arrow button will turn red. Click it again before the next trial starts to undo this operation.



Session Scheduler Override

Slot Methods for Responding to Session Changes

These control slot methods capture status information about the session. They are available as method definitions inside Pynapse states. Write a method with this name to react to the corresponding event.

Slot name	Event
s_Session_start	Start button pressed, or `p_Session.startSession()` called
s_Session_pause	Pause button pressed, or `p_Session.pauseSession()` called
s_Session_resume	Resume button pressed, or `p_Session.resumeSession()` called
s_Session_stop	Stop button pressed, or `p_Session.stopSession()` called

Important

The `s_Session` methods are only available if **Session Mode Controls** is enabled in the Controls tab.

Example

Switch to a starting state when user clicks the 'Start' button on the Pynapse tab at runtime.

```
class Always:    # StateID = 0
    def s_Session_start():
        p_Metric.count.write(0)
        p_State.switch(PreTrial)

    def s_Session_pause():
        p_Metric.count.write(0)
        p_State.switch(Resting)

    def s_Session_resume():
        p_State.switch(PreTrial)

    def s_Session_stop():
        print(p_Metric.ntrials.read(), 'trials completed')
        p_State.switch(EndTrials)
```

Methods

All state methods have the form `p_Session.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

Session Control

startSession

Mimics the behavior of clicking the 'Start' button.

pauseSession

```
p_Session.pauseSession()
```

Mimics the behavior of clicking the 'Pause' button.

resumeSession

```
p_Session.resumeSession()
```

Mimics the behavior of clicking the 'Resume' button.

stopSession

```
p_Session.stopSession()
```

Mimics the behavior of clicking the 'Stop' button.

disabManSessionControl

```
p_Session.disabManSessionControl()
```

Disables the Start/Stop/Pause/Resume buttons on the run-time interface.

enabManSessionControl

```
p_Session.enabManSessionControl()
```

Enables the Start/Stop/Pause/Resume buttons on the run-time interface.

Trial Control

setTrialMax

```
p_Session.setTrialMax(tmax)
```

Write a new maximum trial number. This should be an integer.

Inputs	Type	Description
tmax	integer	New value assigned to trial max

setBlockMax

```
p_Session.setBlockMax(bmax)
```

Write a new maximum block number. This should be an integer.

Inputs	Type	Description
bmax	integer	New value assigned to block max

startTrial

```
p_Session.startTrial()
```

Begin the next trial in the session.

startBlock

```
p_Session.startBlock()
```

Begin the next block in the session.

endTrial

```
p_Session.endTrial()
```

End the current trial in the session.

Status

getTrialMax

```
p_Session.getTrialMax()
```

Read the maximum trial number (integer).

getBlockMax

```
p_Session.getBlockMax()
```

Read the maximum block number (integer).

curTrial

```
p_Session.curTrial()
```

Read the current trial number (integer).

curBlock

```
p_Session.curBlock()
```

Read the current block number (integer).

curSession

```
p_Session.curSession()
```

Read the current session number (integer).

isBlockStarting

```
p_Session.isBlockStarting()
```

See Flow Control to see when this returns true during the session.

isBlockEnding

```
p_Session.isBlockEnding()
```

See Flow Control to see when this returns true during the session.

isBlockNew

```
p_Session.isBlockNew()
```

See Flow Control to see when this returns true during the session.

isSessionStarting

```
p_Session.isSessionStarting()
```

See Flow Control to see when this returns true during the session.

isSessionEnding

```
p_Session.isSessionEnding()
```

See Flow Control to see when this returns true during the session.

isSessionActive

```
p_Session.isSessionActive()
```

See Flow Control to see when this returns true during the session.

isBlockActive

```
p_Session.isBlockActive()
```

See Flow Control to see when this returns true during the session.

isTrialActive

```
p_Session.isTrialActive()
```

See Flow Control to see when this returns true during the session.

Timers

markTime

```
p_Session.markTime(idx=1)
```

Start a custom timer (up to four can be used).

Inputs	Type	Description
idx	integer	Timer index (1 to 4)

sinceRecordStart

Read the elapsed time since the recording began.

```
ts = p_Session.sinceRecordStart()
```

sinceSessionStart

Read the elapsed time since the current session began.

```
ts = p_Session.sinceSessionStart()
```

sinceBlockStart

Read the elapsed time since the current block began.

```
ts = p_Session.sinceBlockStart()
```

sinceTrialStart

Read the elapsed time since the current trial began.

```
ts = p_Session.sinceTrialStart()
```

sinceTrialEnd

Read the elapsed time since the last trial ended.

```
ts = p_Session.sinceTrialEnd()
```

sinceMark

```
ts = p_Session.sinceMark(idx=1)
```

Read a custom timer that was started with `markTime`.

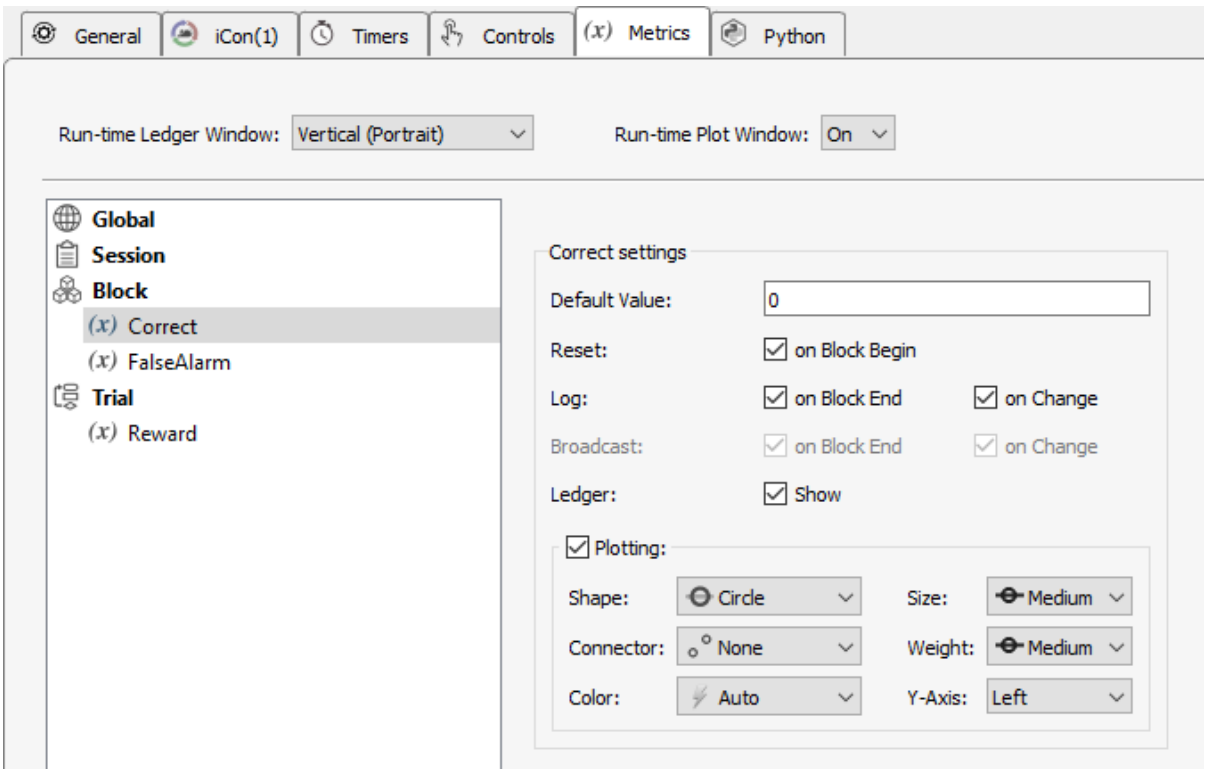
Inputs	Type	Description
idx	integer	Timer index (1 to 4)

Metrics

Metrics are global variables you can read/write in your Python logic code. They can also be added to the runtime interface for visual display, logged at certain time points in the trial, and plotted.

A metric can be any python object. This is similar to using the `global` keyword in Python, except that by storing it as a Pynapse asset you can use the methods below to read and modify it, and all of the logging and display/plotting is taken care of for you.

Give each variable a name and a default value in the Metrics Tab. If Session Controls are enabled, you can assign a metric to trials, blocks, or sessions. You can add up to 50 metrics.



Metrics Tab

Option	Description
Reset*	Reset the metric to 0 after the session event
Log	Determine if and when to log the metric, if **User Log File** is enabled in the General Tab. See Metric Logging for more information.
Ledger	Show string representation in the runtime UI, if **Run-time Ledger Window** is enabled. See Ledger below.
Broadcast	Include the metric in the broadcast UDP packet, if **UDP Broadcast** is enabled on the General Tab. See UDP for more information.
Plotting	Choose the plot shape and color, if **Run-time Plot Window** is enabled. All metrics in the same scope are plotted together. See Plotting below.

Important

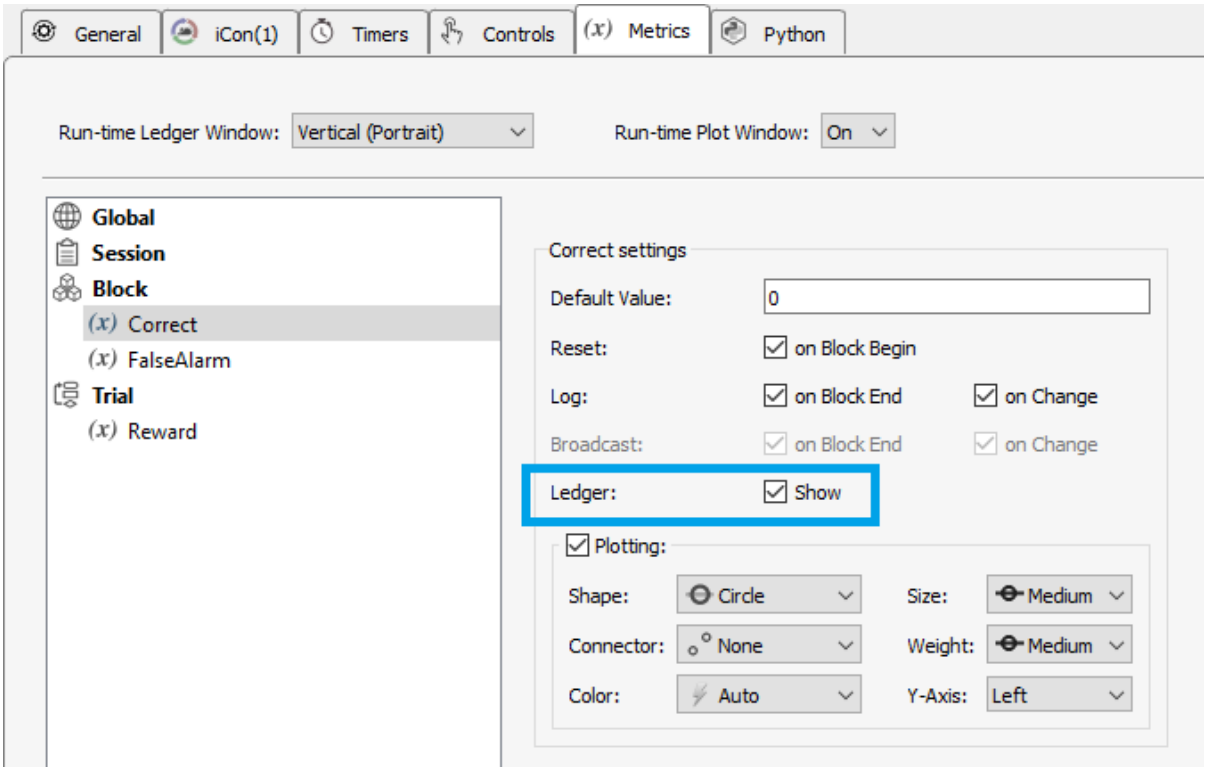
*The metric resets 'on Trial Begin' after the function that calls `startTrial` exits. In the example below, the MinPress metric will be 0 after `s_State_enter` finishes and won't have the desired value. When manually setting metrics, turn off the 'Reset' option.

```
class StartTrial:    # StateID = ?
    def s_State_enter():
        p_Session.startTrial()
        p_Metric.MinPress.write(random.randint(5,10))
```

Run-time Interface

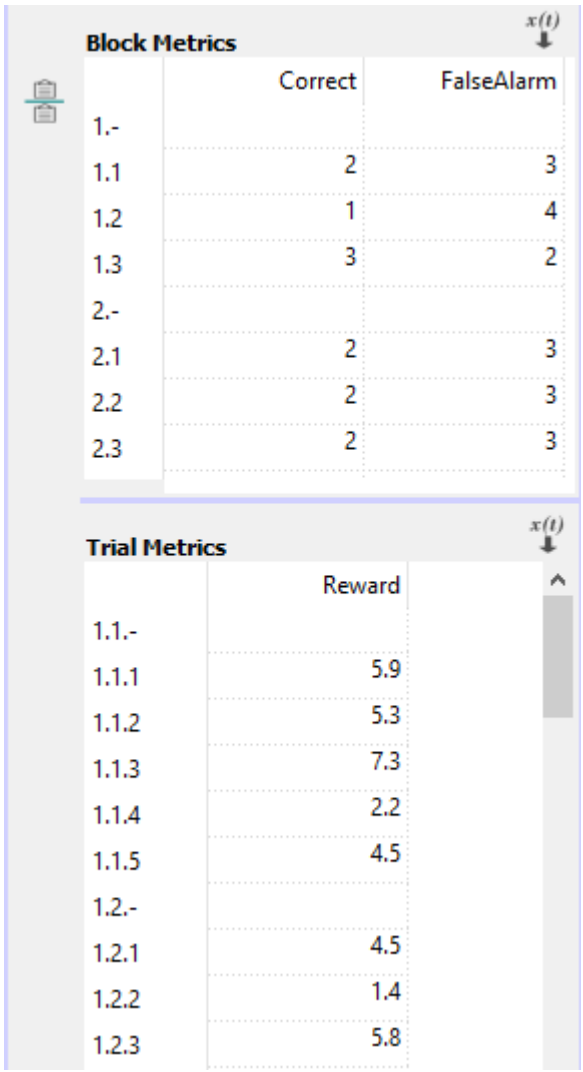
The Metrics run-time interface can have two elements, the ledger and plot windows. Metrics are organized by scope - Session, Block, Trial, or Global.

Ledger



Metrics Ledger Option

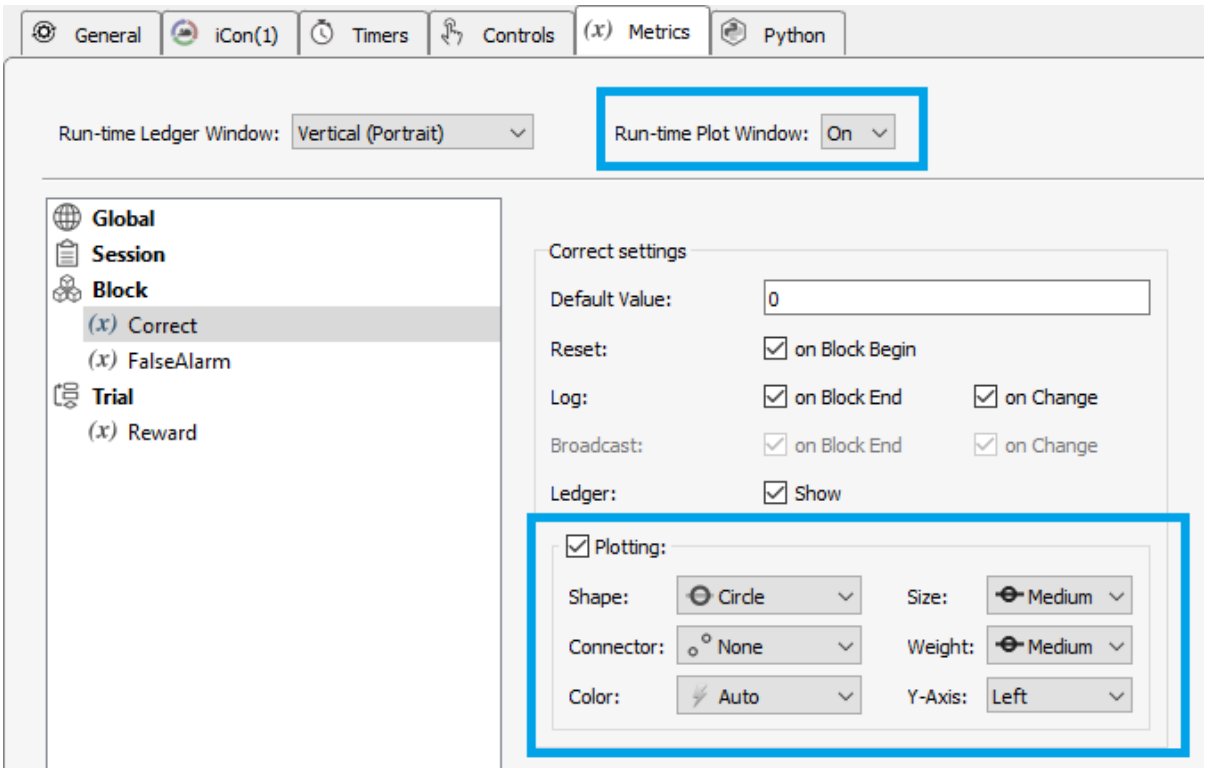
Metrics with the `Ledger` option selected appear on the ledger. The numbers on the ledger indicate the `{SESSION}.{BLOCK}.{TRIAL}` for that entry. New blocks / trials are indicated with a `-`. The **Run-time Ledger Window** setting organizes the metric panes either vertically or horizontally by default. The layout can be changed at run-time.



Metrics Ledger Interface

Button	Description
	Organize metrics into columns
	Organize metrics into rows
	Organize metric panes horizontally
	Organize metric panes vertically
	Show all Session, Block, and Trial metrics in the Global list as well. There must be at least one Global metric in the ledger to see this button

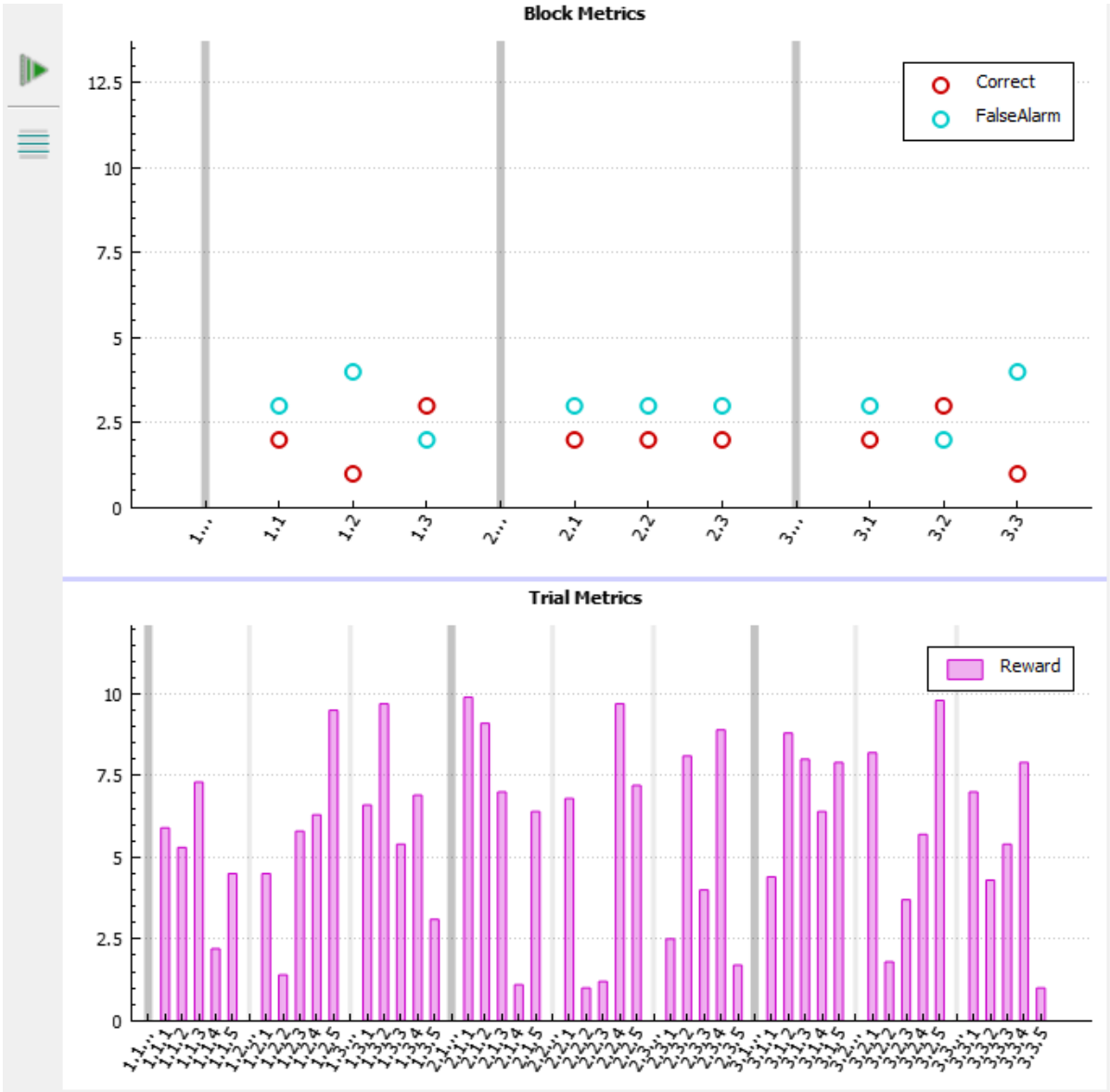
Plotting



Metrics Plot Option

Enable **Run-time Plot Window** and choose the plot shape and color. Metrics are organized in the plots by their scope. Darker gray vertical bars represent new sessions. Lighter gray vertical bars represent new blocks.

If you have multiple metrics in the same scope and need two axes to represent the data in a meaningful way, use the `Y-axis` Left/Right setting to plot metrics on two different axes on the same plot. Each metric has an option to choose which of the two axes to use.



Metrics Plot Interface

Button	Description
	Organize plot panes horizontally, with Global at the bottom
	Organize plot panes vertically
	Stop plots from updating, for review
	Resume updating plots

Adjusting the plots

Hover near the axis you want to change and use the mouse to change the zoom and scale. The mouse cursor will change to a vertical or horizontal axis with a finger pointed in the direction of the axis as you get near.

Left-click and drag to change the axis offset. Hold CTRL + left-click and drag, or move the mouse wheel, to change axis zoom. Here is the full list of cursors and actions:

Cursor	Mouse Action	Description
	None	Bottom axis is currently targeted
	Left-click + drag	Change bottom axis offset
	CTRL + left-click + drag, or mouse wheel	Zoom bottom axis
	None	Left axis is currently targeted
	Left-click + drag	Change left axis offset
	CTRL + left-click + drag, or mouse wheel	Zoom left axis
	None	Right axis is currently targeted
	Left-click + drag	Change right axis offset
	CTRL + left-click + drag, or mouse wheel	Zoom right axis

Methods

All metric methods have the form `p_Metric.{METRIC_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

Status

read

```
value = p_Metric.varname.read()
```

Read the current value of a metric variable.

Control

write

```
p_Metric.varname.write(value)
```

Write a new value to the metric variable. This can be any python object.

Inputs	Type	Description
value	python object	New value assigned to global variable

inc

```
p_Metric.varname.inc(delta=1)
```

Increment the metric value by delta (default is 1).

Inputs	Type	Description
delta	number	Amount to increase variable value (default=1)

dec

```
p_Metric.varname.dec(delta=1)
```

Decrement the metric value by delta (default is 1).

Inputs	Type	Description
delta	number	Amount to decrease variable value (default=1)

scale

```
p_Metric.varname.scale(sf)
```

Scales the metric value by sf.

Inputs	Type	Description
sf	number	Amount to scale variable value

round

```
p_Metric.varname.round(ndec)
```

Rounds the metric to ndec decimal places.

Inputs	Type	Description
ndec	integer	Decimal places

Data Conversion

toFloat

```
p_Metric.varname.toFloat()
```

Convert the metric value to a floating point number for math operations

toInt

```
p_Metric.varname.toInt()
```

Convert the metric value to an integer number for math operations

toString

```
p_Metric.varname.toFloat()
```

Convert the metric value to a string.

toPretty

```
p_Metric.varname.toPretty()
```

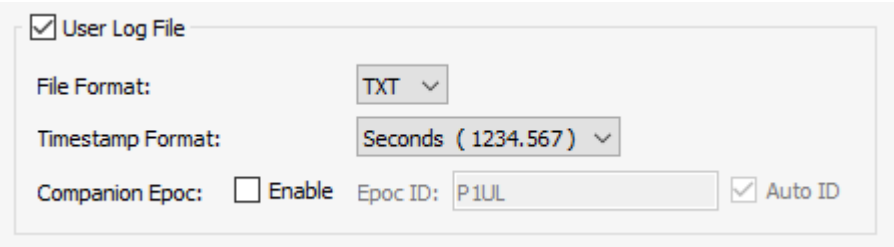
Convert the metric value to a string containing its name and value. Useful for displaying to the console

Example

```
p_Metric.varname.write(1)

# prints 'varname=1'
print(p_Metric.varname.toPretty())
```

Logs

Log files can be saved alongside the experiment files. Enable logs on the General Tab.



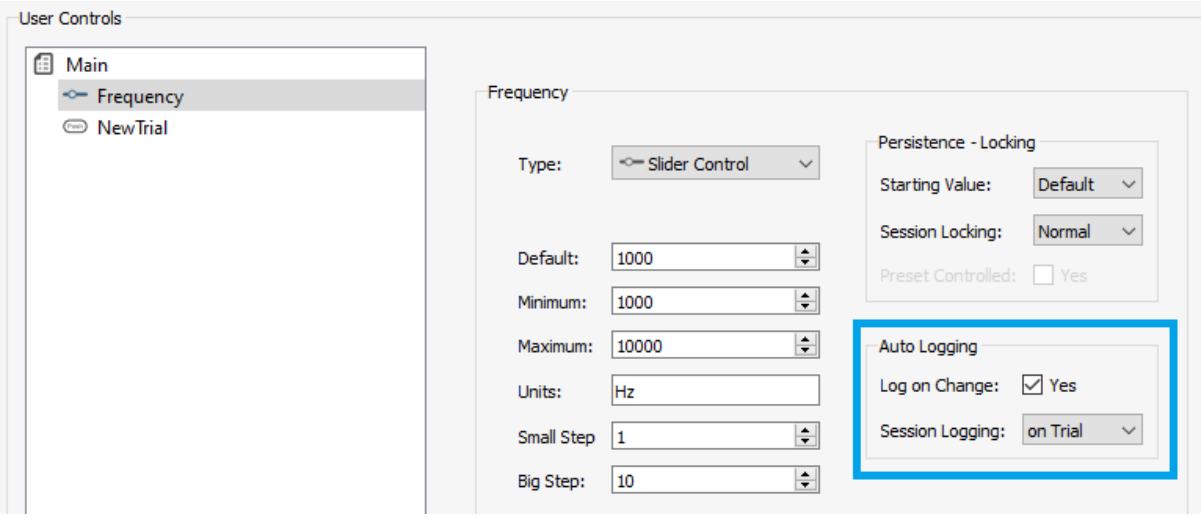
Log Options

Control values, Metric values, Session information, and custom text can be written to the log. The log files are saved in the data block folder with the name `{GIZMO_NAME}_user_log.{FILE_FORMAT}`. The file format can be a tab-delimited TXT file, or a comma-separated CSV file.

All log entries except for raw user text contain a log number and timestamp. Enable `Companion Epoc` to also capture the log number and timestamp in the data block as an epoc event.

Control Logging

Control values can be automatically logged when they change value. If Session Controls are enabled, the values can also be logged when a new Session, a new Block, or a new Trial starts. They can also be logged manually with the `writeControlValue` method.



Control Log Options

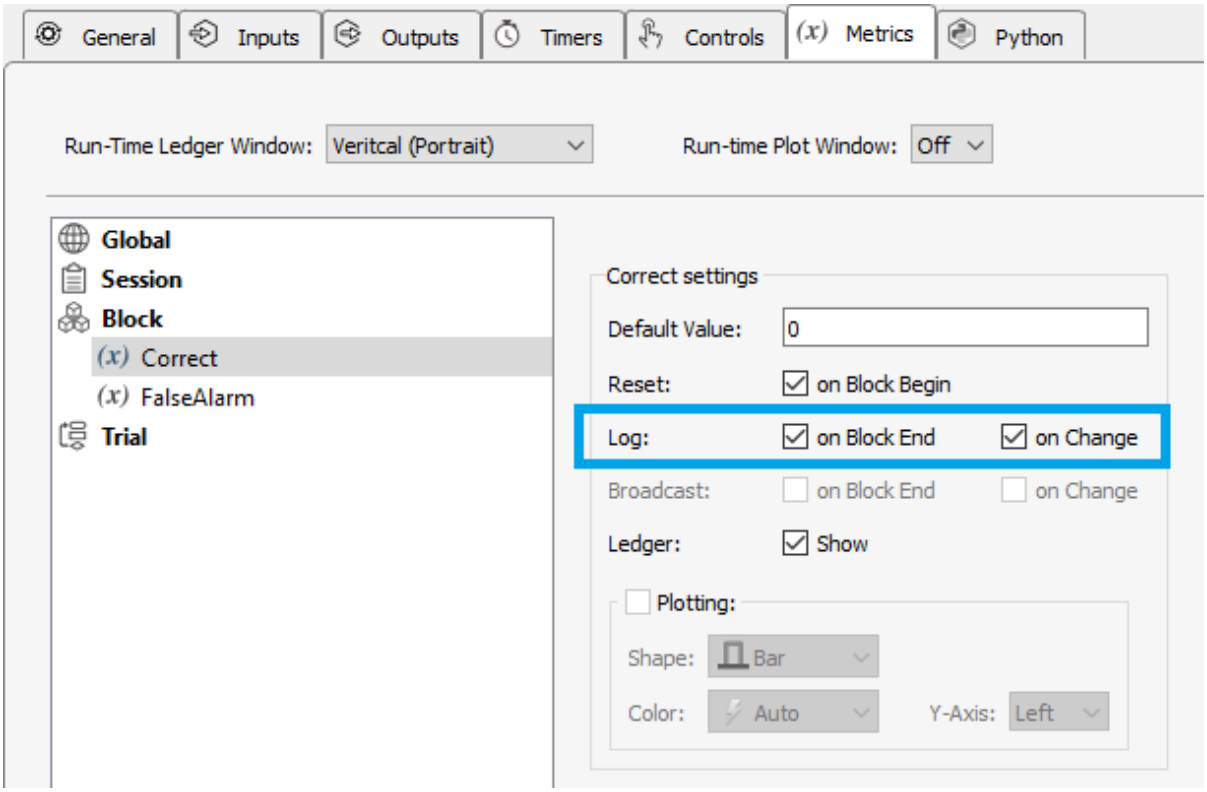
Example output line in the log file

```
22  4.013   CONTROL:    Frequency   1000
```

This is an ID number, timestamp, type, control name, and value.

Metric Logging

Metric values can be automatically logged when they change value. If Session Controls are enabled and the metric is categorized as a Session, Block, or Trial metric, then the metric value can also be logged at the end of that time period. They can also be logged manually with the `writeMetricValue` method.



Metrics Log Options

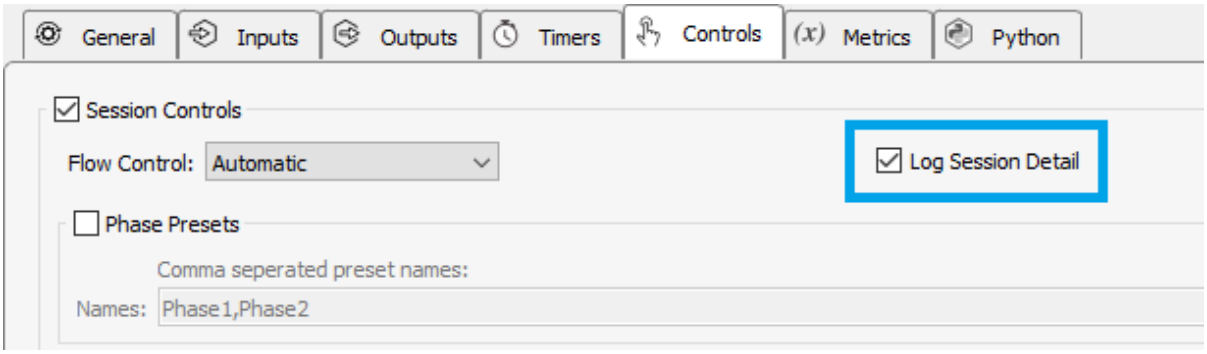
Example output line in the log file

```
6   2.131   METRIC: Correct 3
```

This is an ID number, timestamp, type, metric name, and value.

Session Logging

Session information (when new sessions, blocks, and trials start) can be logged automatically. This includes the start of a session, block, or trial, or when a session resumes.



Session Log Options

Block and trial start logs contain the block/trial number. Session start logs contain the name of the phase and the number of trials / blocks defined at the start of the session.

Example output line in the log file

```
1   0.641   SESSION:    Start   P*  5   2
2   1.505   BLOCK:      Start   1
4   1.507   TRIAL:      Start   1
5   1.629   TRIAL:      Start   2
6   2.145   TRIAL:      Start   3
7   2.288   TRIAL:      Start   4
8   2.461   TRIAL:      Start   5
9   2.602   BLOCK:      Start   2
11  2.604   TRIAL:      Start   1
12  2.762   TRIAL:      Start   2
13  2.949   TRIAL:      Start   3
14  3.122   TRIAL:      Start   4
15  3.294   TRIAL:      Start   5
```

For sessions, this is an ID number, timestamp, event name, event type, and phase information. For blocks and trials, this is an ID number, timestamp, event name, event type, and counter value.

Custom Text Logging

Use `writeSessionEntry` to add timestamped notes to the log file.

Example output line in the log file

```
12  1.265   USER:   my user notes
```

This is an ID number, timestamp, type, and string value.

Use `writeRawText` to add raw text to the file with no numbers or timestamps.

Example output line in the log file

```
raw text with no formatting
```

This is just the string value

Methods

All log methods have the form `p_Log.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

writeControlValue

```
p_Log.writeControlValue(cname)
```

Write a timestamped control value to the log file.

Inputs	Type	Description
cname	string	name of control to write to log file

writeMetricValue

```
p_Log.writeMetricValue(mname)
```

Write a timestamped metric value to the log file.

Inputs	Type	Description
mname	string	name of metric to write to log file

writeRawText

```
p_Log.writeRawText(strg)
```

Write raw text to the log file. Make sure to add the newline character (`\n`) at the end of the string to advance the log file to the next line.

Inputs	Type	Description
strg	string	raw text to write to log file

writeSessionEntry

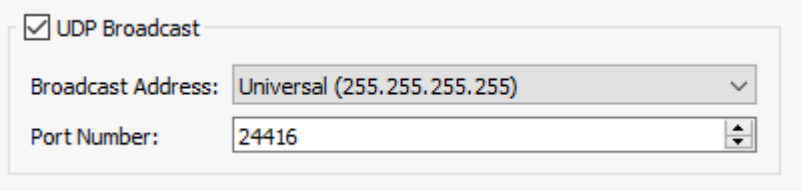
```
p_Log.writeSessionEntry(strg)
```

Write a custom timestamped entry to the log file.

Inputs	Type	Description
strg	string	raw text to write to log file

UDP

UDP Broadcast allows you to send information from Pynapse to client software. Control values, Metric values, Session-related text, and custom text can be sent on the network. Enable UDP Broadcast on the General Tab.



UDP Broadcast Options

Client classes for MATLAB and Python are available in the MATLAB and Python SDKs. See Programming Guide.

All UDP broadcast entries except for raw user text contain the session, block, and trial number.

Control Packet

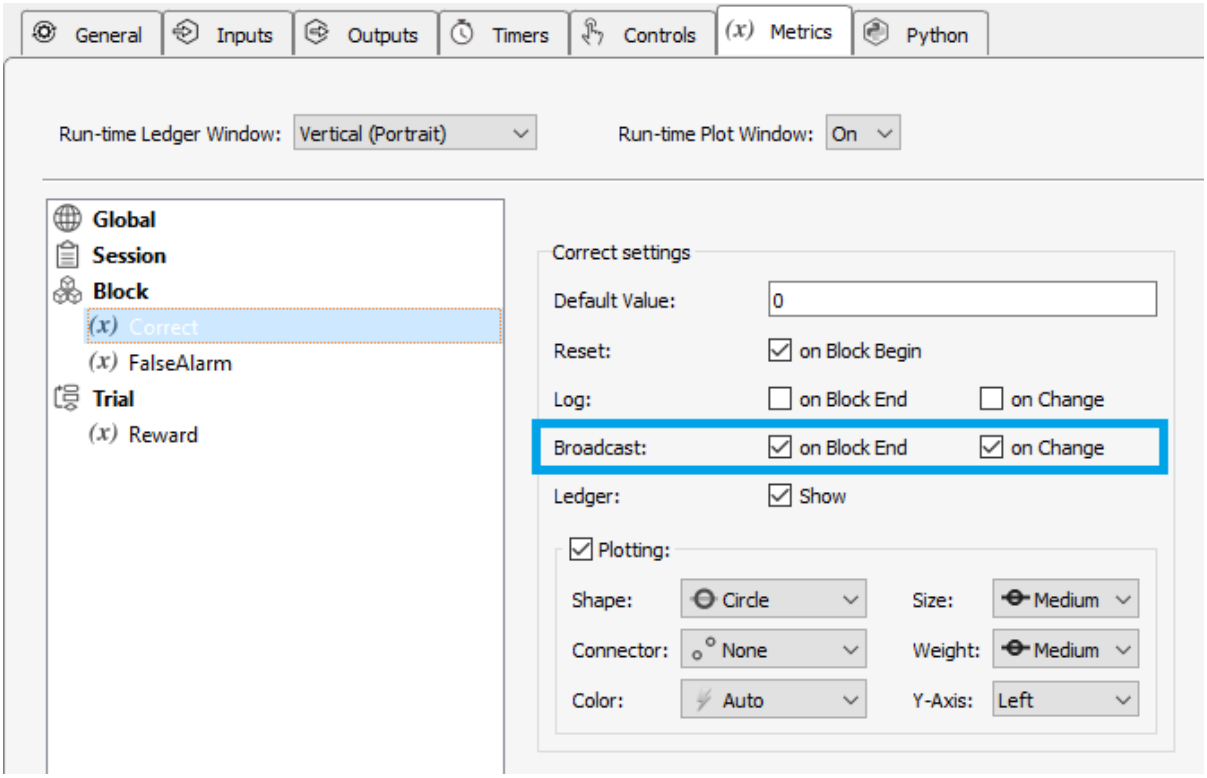
Control values can be broadcast manually with the `sendControlValue` method.

Example output

```
[1.1.2] Frequency=1000
```

Metric Packet

Metric values can be automatically broadcast when they change value. If Session Controls are enabled and the metric is categorized as a Session, Block, or Trial metric, then the metric value can also be broadcast at the end of that time period. They can also be broadcast manually with the `sendMetricValue` method.



Metrics UDP Options

Example output

```
[1.3.4] Correct=2
```

Custom Text Packet

Use `sendSessionEntry` to send text with the current session, block, and trial number.

Example output

```
[2.1.2] my custom text
```

Use `sendRawText` to send raw text (with or without a timestamp since the start of the recording).

Example output

```
# with timestamp
00:09.91 raw text with no formatting

# without timestamp
raw text with no formatting
```

Methods

All UDP methods have the form `p_Udp.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

sendControlValue

```
p_Udp.sendControlValue(cname)
```

Broadcast a control value to the network prefixed with the current session, block, and trial number.

Inputs	Type	Description
cname	string	name of control to write to log file

sendMetricValue

```
p_Udp.sendMetricValue(mname)
```

Broadcast a metric value to the network prefixed with the current session, block, and trial number.

Inputs	Type	Description
mname	string	name of metric to write to log file

sendRawText

```
p_Udp.sendRawText(strg, withTimeStamp=False)
```

Broadcast a custom string to the network.

Inputs	Type	Description
strg	string	raw text to write to broadcast
withTimeStamp	bool	set True to include timestamp in the broadcast packet

sendSessionEntry

```
p_Udp.sendSessionEntry(strg)
```

Broadcast a custom string to the network prefixed with the current session, block, and trial number.

Inputs	Type	Description
strg	string	raw text to write to log file

Programming Guide

MATLAB

You can download the latest MATLAB SDK files here.

The PynapseUDP class installs to:

```
C:\TDT\TDTMatlabSDK\TDTSDK\UDP
```

Example scripts install to:

```
C:\TDT\TDTMatlabSDK\Examples\UDP
```

Reading from Pynapse UDP

```
% instance of class that reads Pynapse UDP
u = PynapseUDP();

while 1
    % block until next UDP packet received
    u = u.read();

    % print it
    disp(u.data)
end
```

Python

The Python PynapseUDP class interfaces with Pynapse. It is available in the tdt pypi package (pip install tdt).

Reading from Pynapse UDP

```
import tdt

udp = tdt.PynapseUDP()

while True:
    udp.recv()
    if udp.data is not None:
        print(udp.data)

udp.server.server_close()
```

Synapse Control

Pynapse has built in slots for Synapse mode change events. These are useful in the 'Always' state to initialize variables or buffers for stimulation before the recording begins. It also has a built-in instance of SynapseAPI to control other gizmo parameters from Pynapse.

Slot Methods for Responding to Synapse Mode Changes

These slots capture Synapse system mode change events. They are available as method definitions inside Pynapse states. Write a method with this name to react to the corresponding mode change event.

Slot name	Event
s_Mode_change(newmode)	Triggers on any mode change
s_Mode_idle	Mode changed to idle
s_Mode_standby	Mode changed to standby
s_Mode_preview	Mode changed to preview
s_Mode_record	Mode changed to record
s_Mode_recprev	Mode changed to preview or record

Important

If you use `p_State.switch()` inside `s_Mode_standby()`, this overrides the 'Initial State' setting on the General Tab.

Example

Preload a stimulus output buffer before the experiment runs

```
import numpy as np

class Always:   #StateID = 0
    def s_Mode_standby():
        import random
        p_Output.MyOutput.setBuffer(np.random.random(1000).tolist())
```

When experiment starts, switch to PreTrial state as the default starting state.

```
class Always:   #StateID = 0
    def s_Mode_standby():
        p_State.switch(PreTrial)
```

SynapseAPI

Pynapse also exposes an instance of the SynapseAPI class as the variable `syn` in the source code editor. Type `syn.` in the Code Editor and code completion shows you all of the available method calls. For the complete list of SynapseAPI methods and how to use them, see SynapseAPI Manual.

Important

SynapseAPI calls goes through sockets, and that adds some extra delay. The SynapseAPI calls are also affected by what is happening in the Synapse window. For example, if you do something that is graphically intensive such as resizing the windows during a recording, you can see a big (>100 ms) lag before the call gets through. It shouldn't be relied on for time critical events.

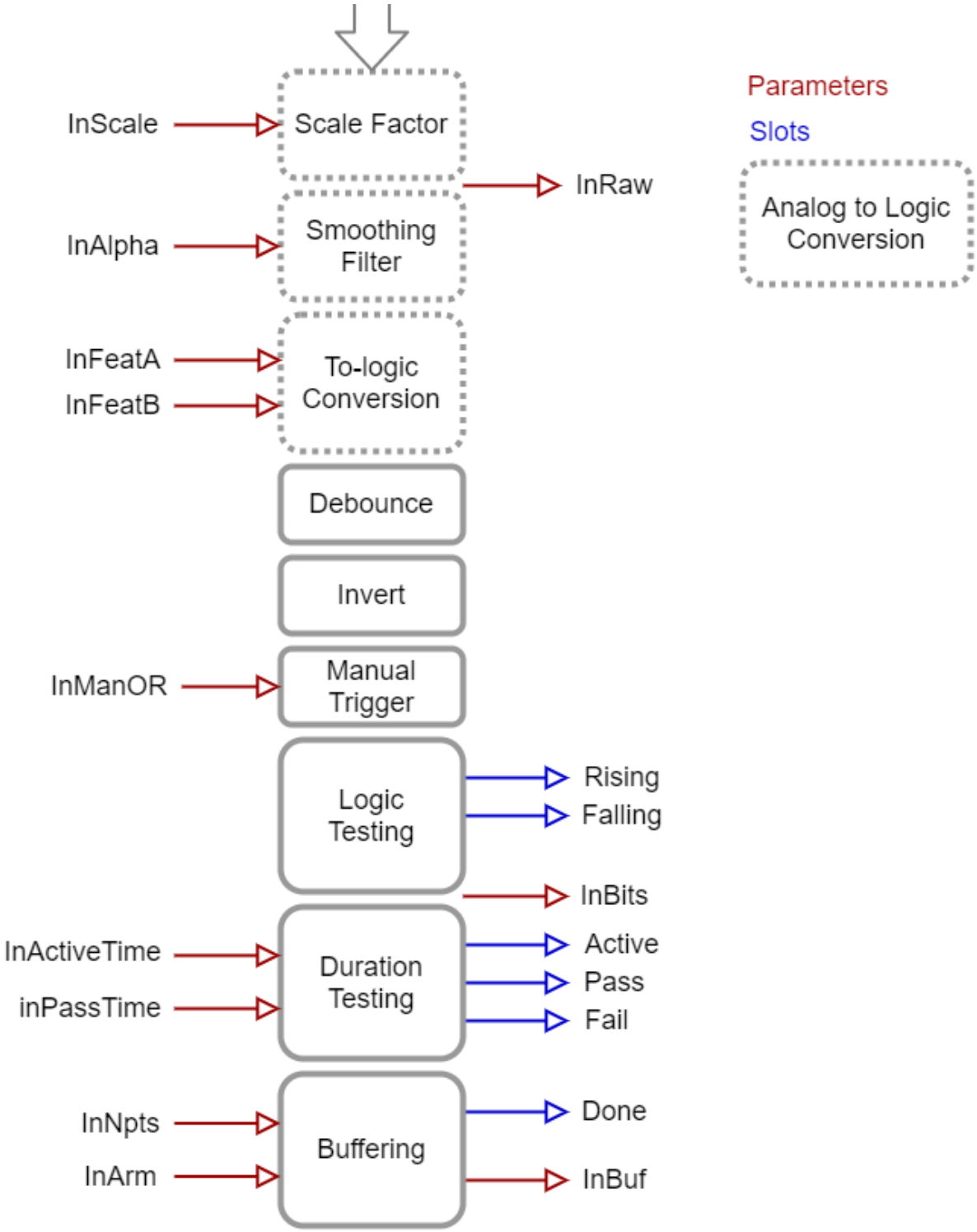
Example

User puts system into Standby mode, then when trigger is received Pynapse switches system to Record mode.

```
class Always:   #StateID = 0
    def s_MyInput_rise():
        syn.setModeStr('Record')
```

Gizmo Inputs

You can have up to 8 gizmo inputs into the Pynapse gizmo. These inputs can be digital signals (logic TTL) or analog signals (float or integer) that go through a Logic Conversion, such as thresholding.



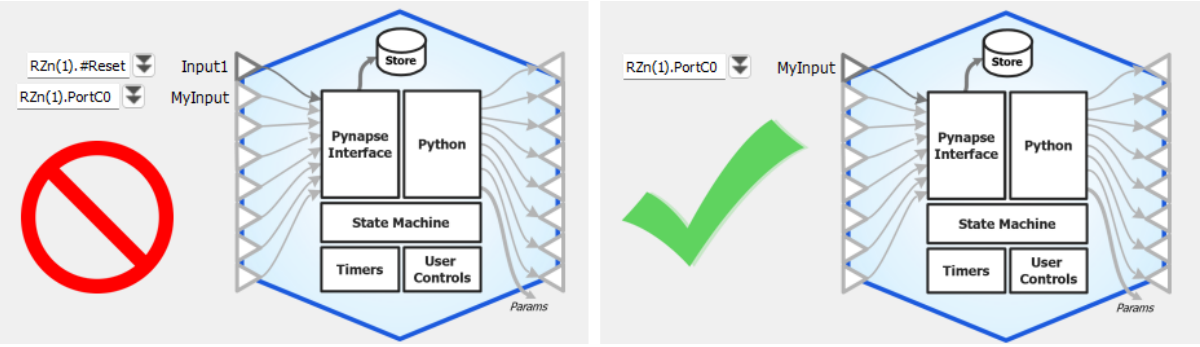
Input Processor

Set the **Name** of the input to something that makes sense for your experiment, e.g. 'NosePoke'. This will be used throughout the Python code.

You can optionally save epoch timestamp events for each input. An integer code for the event type is stored with the timestamp. See Epoc Storage below for more information.

Important

By default, Input1 is enabled and connected to the "#Reset" signal as shown below so that Synapse compiles it correctly. If you make your own gizmo inputs, replace #Reset and start with the first input.



Logic Conversion for Number Signals

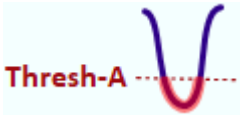
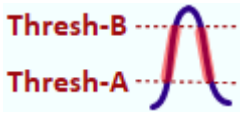
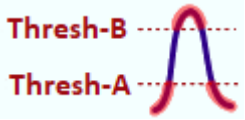
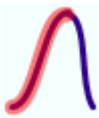
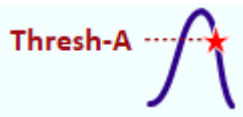
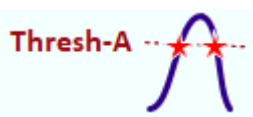
Number input signals pass through a logic conversion so they can trigger on/off events in Pynapse.

Pre-scale is a scalar multiplied by the signal.

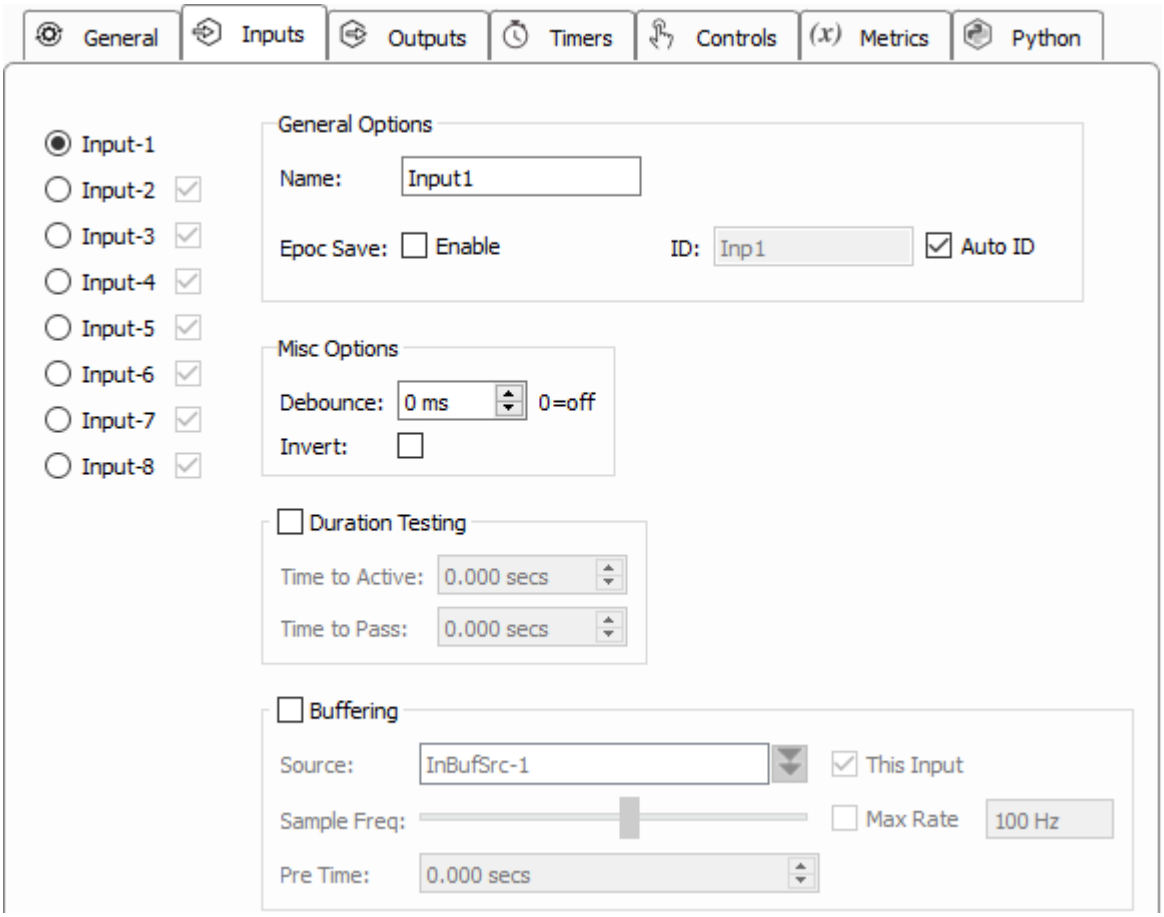
Smoothing is a low pass filter that removes jitter on the analog signal before logic conversion. This helps avoid a situation where the signal is quickly bouncing around the test threshold.

By default, the **Test** is **Above**, which is a simple threshold detection method to convert the number signal input into a logic signal when it goes beyond the **Thresh-A** value.

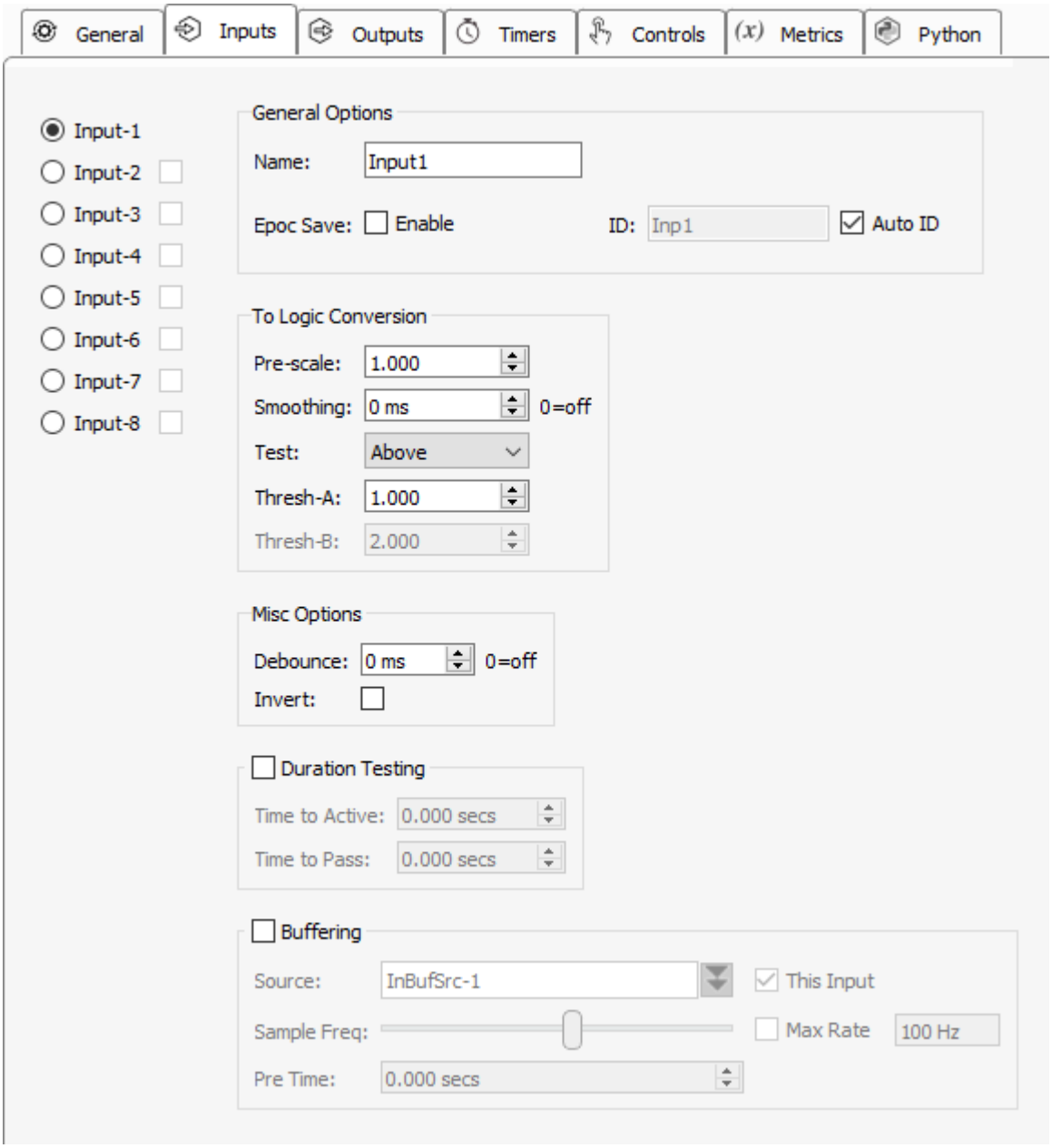
Epoc events are triggered on the 'rise' event of all of these tests. If the test can be true for more than 2 samples ('Strobe') then a timestamp for the 'fall' event is also stored. See Epoc Storage below for full epoc code information.

Test Type		Description	Duration
Above		Signal is above Thresh-A	Strobe
Below		Signal is below Thresh-A	Strobe
Between		Signal is between Thresh-A and Thresh-B	Strobe
Outside		Signal is outside Thresh-A and Thresh-B	Strobe
Rising		Signal is increasing in value	Strobe
Falling		Signal is decreasing in value	Strobe
Peak		Signal forms a local peak	Trigger
Valley		Signal forms a local valley	Trigger
Tip		Signal forms a local peak or valley	Trigger
Rise Thru		Signal rises through Thresh-A from below	Trigger
Fall Thru		Signal falls through Thresh-A from above	Trigger
Pass Thru		Signal passes through Thresh-A from above or below	Trigger

Inputs Tab - Logic Options



Inputs Tab - Number Options



Invert and **Debounce** are typically used if the input is coming directly from a digital input on the RZ processor. **Debounce** is the amount of time the input has to settle before its new state is used. This is useful for lever presses or hardware button presses which can 'bounce' on contact and trigger several rapid artificial events before making solid contact.

Slot Methods for Responding to Input States

These input slots capture status information about the inputs. They are available as method definitions inside Pynapse states for each input. Write a method with this name to react to the corresponding event.

Slot name	Operation	Event
s_Input1_rise()	Status	input changed to true
s_Input1_fall()	Status	input changed to false
s_Input1_active()	Duration	input passed the 'Time to Active' duration test (see Duration Testing)
s_Input1_pass()	Duration	input passed the 'Time to Pass' duration test (see Duration Testing)
s_Input1_fail()	Duration	input failed the 'Time to Pass' duration test, after passing 'Time to Active' (see Duration Testing)
s_Input1_done()	Buffer	input buffer is full and ready to be read (see Buffering)

Note

'Input1' is the default name of the first input. The name of each slot method gets replaced with the name of your actual input, so if you name the input 'NosePoke' then s_NosePoke_rise() is an available slot

Example

Move through behavioral states based on status of MyInput

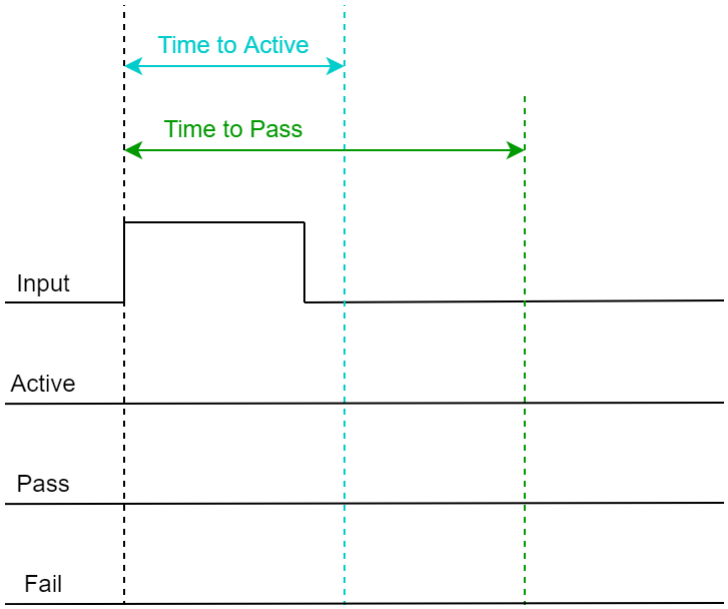
```
class PreTrial:
    def s_MyInput_rise():
        p_State.switch(StartTrial)

class StartTrial:    # StateID = ?
    def s_MyInput_active():
        p_State.switch(ActiveState)

    def s_MyInput_fall():
        p_State.switch(PreTrial)

class ActiveState:    # StateID = ?
    def s_MyInput_pass():
        p_State.switch(PassState)

    def s_MyInput_fail():
        p_State.switch(FailState)
```

Duration Testing

The inputs can use built-in duration testing. In this example, the button has to be pressed for 600 ms to get to the 'Active' state, and another 400 ms for it to 'Pass'. This timing happens on the hardware and the `active`, `pass` and `fail` slots are triggered in Pynapse.

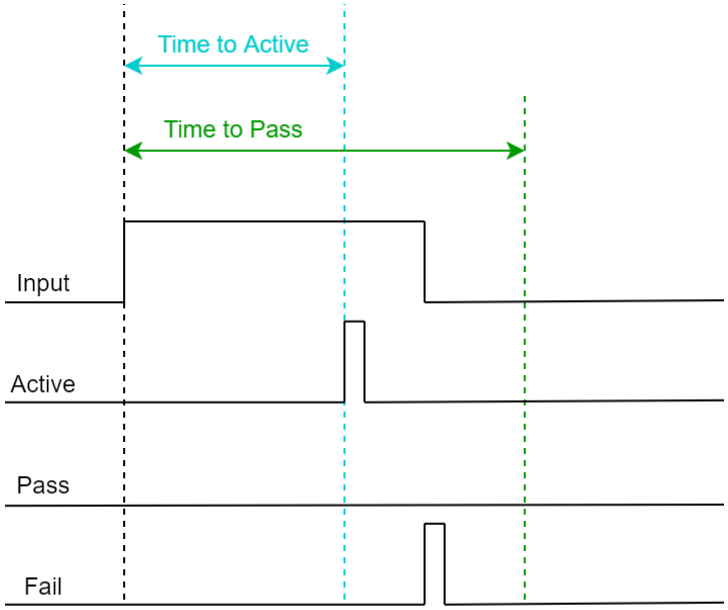
No Trial

Time to Active not reached by Input.
No trial is initiated.



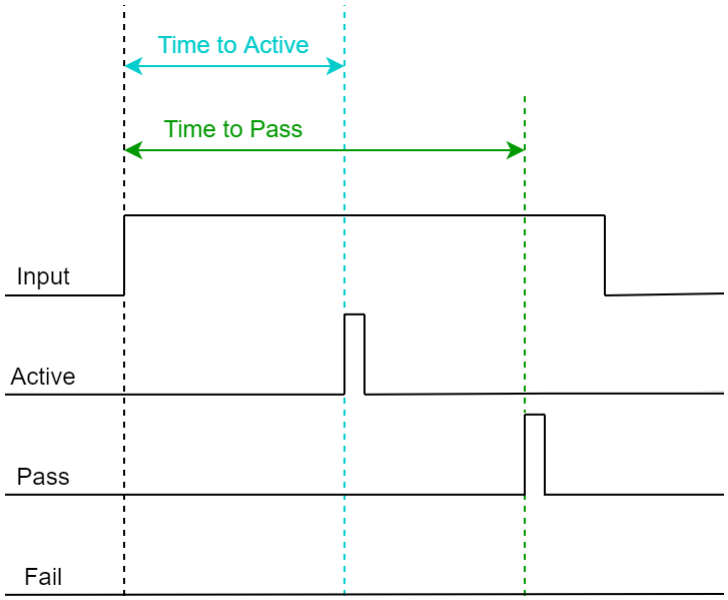
Fail

Time to Active reached by Input, so 'Active' trigger fires.
Time to Pass was not reached by Input, so the 'Fail' trigger fires when Input is released.



Pass

Time to Active reached by Input, so 'Active' trigger fires.
Time to Pass also reached by Input, so 'Pass' trigger fires.



Epoc Storage

Epoc events are triggered on the 'rise' event of the input and a timestamp and value of 3 is stored in the data tank. If the input is true for more than 2 samples then the 'fall' event is also timestamped and stored, with a value of 4.

The full state of the input, including duration test results, is captured in the integer code:

b6	b5	b4	b3	b2	b1	b0
Done	Fail	Pass	Active	Fall	Rise	True

Example values of the epoc event:

Event	Value	Binary Representation
Rise	3	0x0000011
Fall	4	0x0000100
Active	9	0x0001001
Pass	17	0x0010001
Fail	36	0x0100100
Done	64	0x1000000

Buffering

Buffering lets you save a small snippet of data in hardware memory and read it into Pynapse. When the input switches to `true` ('rise' trigger) the buffer is captured. When buffering is finished it fires the `done` trigger.

You can connect the buffer signal source can be any single channel signal in your experiment, including the input signal. Even though the inputs are all converted to digital signals for logic tests, you can still trigger Pynapse to buffer up the original analog signal and then read that into Python for online analysis. For example, you can do a threshold detection on a signal and save the snippet around the threshold crossing, and do something with this in Pynapse.

Example

Display a 1000 sample buffer when triggered.

```
%matplotlib

import matplotlib.pyplot as plt

class Always:   #StateID = 0

    def s_Mode_standby():
        # set up the buffer and plot
        p_Input.MyInput2.setBufferSize(1000)
        p_Input.MyInput2.armBuffer()
        plt.plot()

    def s_MyInput_done():
        # get buffer
        arr = p_Input.MyInput2.getBuffer()

        # plot buffer contents
        plt.plot(arr)
```

Methods

All input methods have the form `p_Input.{INPUT_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you.

Buffer operations

Read a triggered snippet of memory from the hardware.

setBufferSize

```
p_Input.MyInput.setBufferSize(npts)
```

Change the number of samples to store in the buffer.

Inputs	Type	Description
npts	integer	Number of samples to buffer

Important

This call must always be made if using a buffer.

Example

Initialize the buffer size before the recording starts.

```
class Always:   #StateID = 0

    def s_Mode_standby():
        # set up the buffer
        p_Input.MyInput.setBufferSize(1000)
        p_Input.MyInput.armBuffer()
```

armBuffer

```
p_Input.MyInput.armBuffer()
```

Let the buffer accept a trigger and fill with new data.

Important

This call must always be made if using a buffer.

Example

Arm the buffer when the experiment first runs.

```
class Always:   #StateID = 0

    def s_Mode_standby():
        # set up the buffer
        p_Input.MyInput.setBufferSize(1000)
        p_Input.MyInput.armBuffer()
```

disarmBuffer

```
p_Input.MyInput.disarmBuffer()
```

Stop the buffer from loading again. Use this to avoid overwriting buffer data before you've had a chance to read it with `getBuffer`.

Example

Prevent the hardware buffer from triggering/ loading new data while you read it.

```
p_Input.MyInput.disarmBuffer()
arr = p_Input.MyInput.getBuffer()
p_Input.MyInput.armBuffer()
```

getBuffer

```
arr = p_Input.MyInput.getBuffer(npts=0, offset=0)
```

Inputs	Type	Description
npts	integer	Number of samples to read (0=all)
offset	integer	Starting index in buffer to read from (0-based)
Returns		
array	number	Buffer contents as python list

Example

Capture the MyInput buffer when the 'done' trigger fires.

```
class Always:   #StateID = 0

    def s_Mode_standby():
        # set up the buffer
        p_Input.MyInput.setBufferSize(1000)
        p_Input.MyInput.armBuffer()

    def s_MyInput_done():
        # get buffer
        arr = p_Input.MyInput.getBuffer()
        print(arr)
```

Duration Settings

setActTime

```
p_Input.MyInput.setActTime(acttime_sec)
```

Override the duration test 'Time to Active' setting.

Inputs	Type	Description
acttime_sec	float	Time to Active, in seconds

Example

Modify the timing test based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, increase the time to active by 50 ms.
    if p_Metric.success.read() > 5:
        p_Metric.active_time.inc(delta=0.05)
        p_Input.MyInput.setActTime(p_Metric.active_time.read())
```

setPassTime

```
p_Input.MyInput.setPassTime(passtime_sec)
```

Override the duration test 'Time to Pass' setting.

Inputs	Type	Description
passtime_sec	float	Time to Pass, in seconds

Example

Modify the timing test based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, increase the time to pass by 50 ms.
    if p_Metric.success.read() > 5:
        p_Metric.pass_time.inc(delta=0.05)
        p_Input.MyInput.setPassTime(p_Metric.pass_time.read())
```

Manual Control

Manual turn inputs on, off, or pulse during runtime. Useful for debugging.

manualOn

```
p_Input.MyInput.manualPulse()
```

Manually turn on the input.

> **Example**
>
> Turn on the input when entering a state.
>
> ```
> def s_State_enter():
>     p_Input.MyInput.manualOn()
> ```

manualOff

```
p_Input.MyInput.manualPulse()
```

Manually turn off the input.

> **Example**
>
> Turn off the input when exiting a state.
>
> ```
> def s_State_exit():
>     p_Input.MyInput.manualOff()
> ```

manualPulse

```
p_Input.MyInput.manualPulse()
```

Manually pulse the input.

Example

Pulse the input when entering a state.

```
def s_State_enter():
    p_Input.MyInput.manualPulse()
```

Number Conversion Settings

Override the feature settings applied to the input signal for logic conversion at runtime.

setFeatureThresholds

```
p_Input.MyInput.setFeatureThresholds(thresh_A, thresh_B)
```

Modify the threshold settings for the logic conversion.

Inputs	Type	Description
thresh_A	float	Threshold (in V) for the Thresh-A parameter
thresh_B	float	Threshold (in V) for the Thresh-B parameter

Example

Modify the lever force requirement based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, increase the force required by 0.05
    if p_Metric.success.read() > 5:
        p_Metric.thresh_A.inc(delta=0.05)
        p_Input.MyInput.setFeatureThresholds(p_Metric.thresh_A.read(),
p_Metric.thresh_B.read())
```

setScale

```
p_Input.MyInput.setScale(scalefactor)
```

Change the scale factor applied to the signal before it goes through the logic conversion.

Inputs	Type	Description
scalefactor	float	Scale factor applied to signal before logic conversion

Example

Modify the scale factor based on a run-time Control.

```
class Always:   #StateID = 0

    # set input scale factor to value of the SmoothCtrl slider at runtime
    def s_SmoothCtrl_change(value):
        p_Input.MyInput.setScale(value)
```

setSmoothing

```
p_Input.MyInput.setSmoothing(tau_sec)
```

Change the smoothing filter applied to the input signal before it goes through the logic conversion.

Inputs	Type	Description
tau_sec	float	Time constant of the low-pass smoothing filter, in seconds (0=off)

Example

Modify the smoothing filter based on a run-time Control.

```
class Always:   #StateID = 0

    # set smoothing 'tau' to value of the TauCtrl slider at runtime
    def s_TauCtrl_change(value):
        p_Input.MyInput.setSmoothing(value)
```

Status

Get information on the current state of the input.

isOn

```
p_Input.MyInput.isOn()
```

Returns true if the input is currently true.

Example

When entering a state, check if an input is already true.

```
def s_state_enter():
    if p_Input.MyInput.isOn():
        print('MyInput is on')
    else:
        print('MyInput is off')
```

isOff

```
p_Input.MyInput.isOff()
```

Returns true if the input is currently false.

Example

When entering a state, check the status of the input.

```
def s_state_enter():
    if p_Input.MyInput.isOff():
        print('MyInput is off')
    else:
        print('MyInput is on')
```

getRawInput

```
p_Input.MyInput.getRawInput()
```

Read the current value of an input. If it is a number, the raw input into the Pynapse gizmo after scale factor is applied but before any feature detection.

Example

When a threshold is crossed, check the current value of the signal.

```
def s_MyInput_rise():
    print(p_Input.MyInput.getRawInput())
```

getStatusBits

```
p_Input.MyInput.getStatusBits()
```

Read the current state of an input as a bitwise integer value. Bit order is:

`Done | Fail | Pass | Active | Fall | Rise | True`

Used by the Pynapse polling loop.

Gizmo Outputs

You can have up to 8 gizmo outputs from the Pynapse gizmo. The outputs can be logic signals that are either turned on/off, triggered for a single sample, or strobed high for a fixed duration. You can also load a custom analog waveform into a buffer and trigger Pynapse to play it out.

Set the **Name** of the output to something that makes sense for your experiment, e.g. 'Reward'. This will be used throughout the Python code and to link to other gizmos.

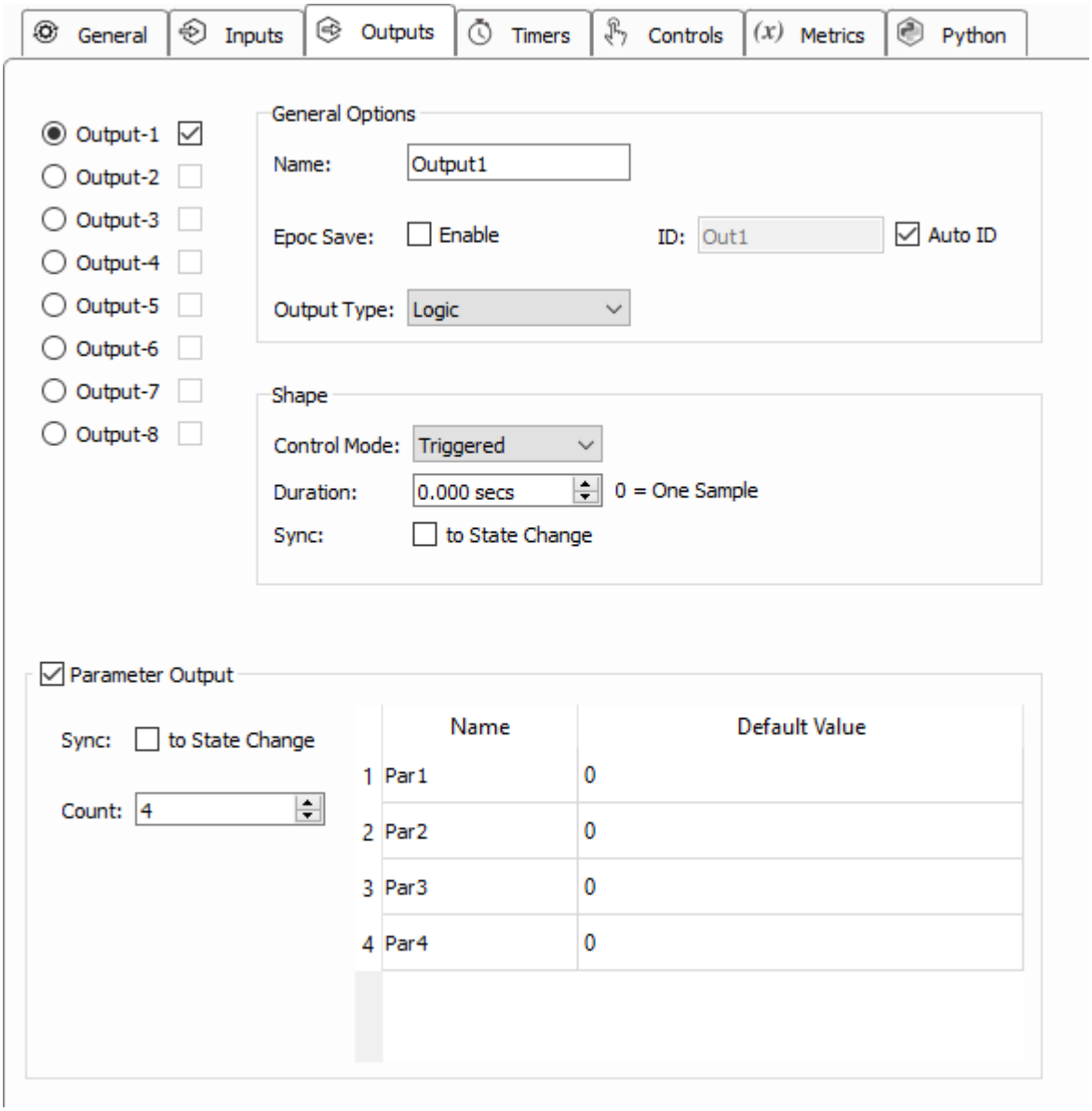
You can optionally save epoch timestamp events for each output. A timestamp is saved when the output turns on. If the output is high for more than 2 samples then the offset is stored as well.

Triggered - output stays high for a fixed amount of time (controlled by hardware). If **Duration** is 0, this is a single sample.

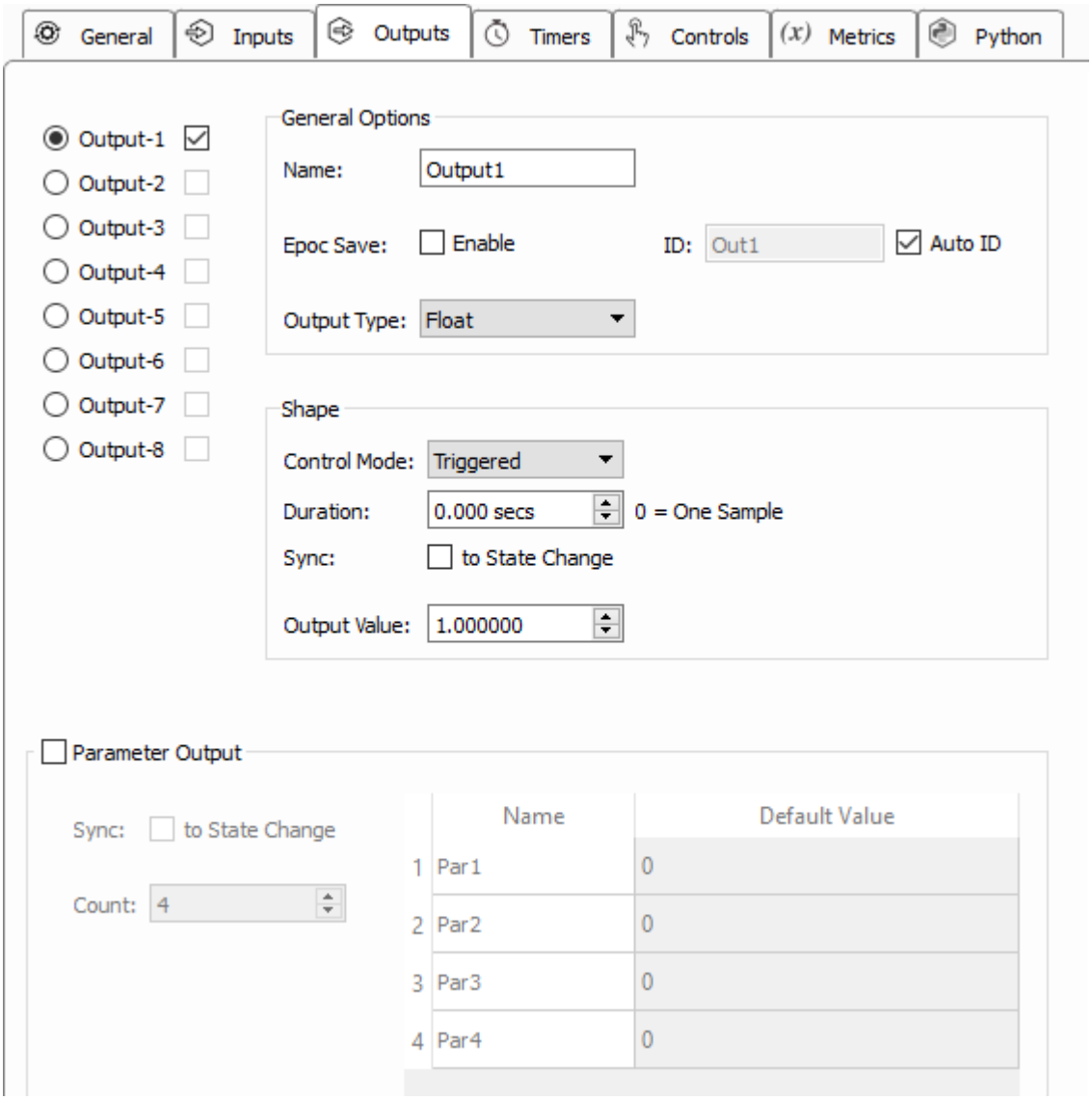
Strobed - output turns on when the turnOn() method is called and turns off when the turnOff() is called.

See Synchronizing Events for information on the **Sync to State Change** option.

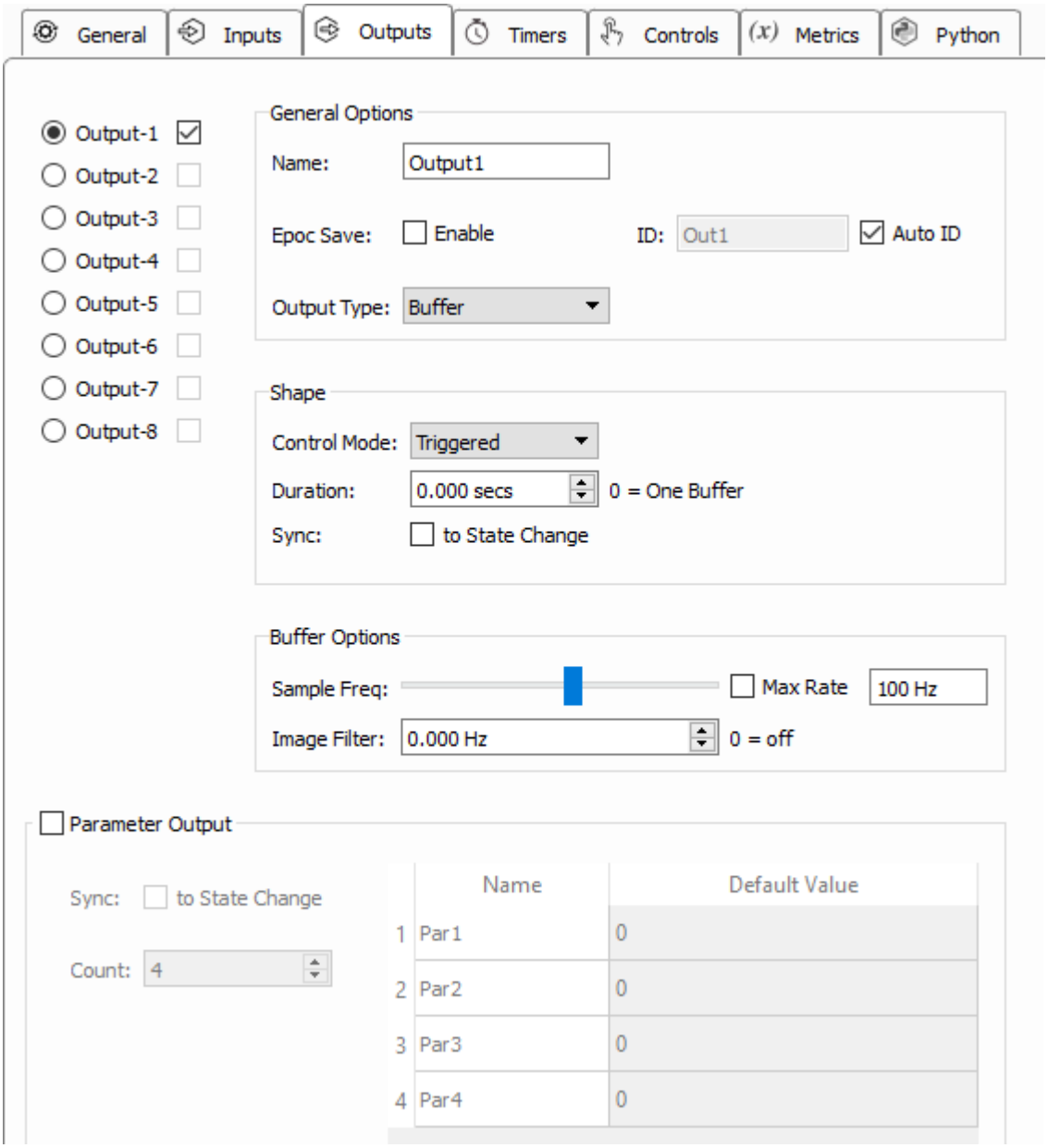
Outputs Tab - Logic Options



Outputs Tab - Number Options



Outputs Tab - Buffer Options



In the Buffer Options, there is an optional low pass filter (**Image Filter**) to remove aliased signals. If unsure, set this to ~⅓ of the output **Sample Freq**.

Buffering

Buffering lets you write a small waveform to hardware memory and trigger it for presentation. This allows you to create fully custom stimuli on the fly, either pre-loaded or adaptive in response to behavioral events.

Parameter Outputs

Pynapse has a set of Parameter outputs which can control all parameters that define a stimulation gizmo. For example, control the waveform parameters of an Audio Stimulation gizmo or an Electrical Stim Driver gizmo directly from Pynapse. This mimics the behavior of the Parameter Sequencer gizmo. Create the parameters on the fly based on subject feedback, or play from a python-generated list. See Parameter Methods below.

See Synchronizing Events for information on the **Sync to State Change** option.

> **Tip**
>
> See Using Parameters for more general information on parameters.

Output Methods

All output methods have the form `p_Output.{OUTPUT_NAME}.{METHOD}`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you. 'Output1' is the default name of the first output. The name of each method gets replaced with the name of your actual output, so if you name the output 'Reward' then `p_Output.Reward.fire()` is an available method.

Manual Control

Manual turn outputs on, off, or fires a pulse waveform during runtime. Useful for stimulus/reward presentation.

fire

```
p_Output.MyOutput.fire()
```

Quickly pulse the output. This is only available when **Control Mode** is set to `Triggered`. If **Duration** is non-zero, the output will stay high for that set duration. Set **Duration** to zero to use this output to trigger other gizmos e.g. trigger an Audio Stimulation gizmo. If **Output Type** is **Buffer**, this will play the output buffer one time.

Example

Trigger an output when the input goes high.

```
class Always:   #StateID = 0

    def s_MyInput_pass():
        p_Output.MyOutput.fire()
```

turnOn

```
p_Output.MyOutput.turnOn()
```

Turn the output on indefinitely. If the output is a buffer, it will continuously loop until turned off. This is only available when **Control Mode** is set to `Strobed`.

Example

Link an input status to an output.

```
class Always:   #StateID = 0

    def s_MyInput_rise():
        p_Output.MyOutput.turnOn()

    def s_MyInput_fall():
        p_Output.MyOutput.turnOff()
```

turnOff

```
p_Output.MyOutput.turnOff()
```

Turn the output off. This is only available when **Control Mode** is set to `Strobed`.

Example

Link an input status to an output.

```
class Always:   #StateID = 0

    def s_MyInput_rise():
        p_Output.MyOutput.turnOn()

    def s_MyInput_fall():
        p_Output.MyOutput.turnOff()
```

Duration Settings

setPulseShape

```
p_Output.MyOutput.setPulseShape(dur_sec, outval=None)
```

Override the output **Duration** (if **Control Mode** is set to **Triggered**) and the **Output Value** settings (if **Output Type** is set to **Float** or **Integer**).

Inputs	Type	Description
dur_sec	float	Duration of the output pulse when triggered with `fire`, in seconds
outval	float or integer	Output value when true

Example

Modify the pulse shape and output value based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, decrease the output pulse time by 50 ms and output
value by 1.
    if p_Metric.success.read() > 5:
        p_Metric.pulse_dur.dec(delta=0.05)
        p_Metric.output_val.dec(delta=1)
        p_Output.MyOutput.setPulseShape(p_Metric.pulse_dur.read(), p_Metric.output_val.read())
```

setDuration

```
p_Output.MyOutput.setDuration(dur_sec)
```

Override the output **Duration** (if **Control Mode** is set to **Triggered**) setting.

Inputs	Type	Description
dur_sec	float	Duration of the output pulse when triggered with `fire`, in seconds

Example

Modify the pulse shape and output value based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, decrease the output pulse time by 50 ms.
    if p_Metric.success.read() > 5:
        p_Metric.pulse_dur.dec(delta=0.05)
        p_Output.MyOutput.setDuration(p_Metric.pulse_dur.read())
```

setValue

```
p_Output.MyOutput.setValue(outval)
```

Override the output **Output Value** setting (if **Output Type** is set to **Float** or **Integer**).

Inputs	Type	Description
outval	float or integer	Output value when true

Example

Modify the pulse shape and output value based on performance.

```
def s_State_enter():
    # if more than 5 successful trials, decrease the output value by 1.
    if p_Metric.success.read() > 5:
        p_Metric.output_val.dec(delta=1)
        p_Output.MyOutput.setValue(p_Metric.output_val.read())
```

Buffer operations

Load a list of values into a memory buffer on the hardware and trigger playback.

setBuffer

```
p_Output.MyOutput.setBuffer(wave)
```

Loads a python list or NumPy array into an output buffer. Call `fire` to play the output buffer once. Call `turnOn` to play buffer on a loop until calling `turnOff`. Supports waveforms between 2 and 100,000 samples long.

Inputs	Type	Description
wave	list	List of numbers to load into output buffer

Example

Load an output buffer with 1,000 random numbers before the recording starts, and trigger it when MyInput goes true.

```
import numpy as np

class Always:   #StateID = 0

    def s_Mode_standby():
        import random
        p_Output.MyOutput.setBuffer(np.random.random(1000).tolist())

    def s_MyInput_rise():
        p_Output.MyOutput.fire()
```

Status

Get information on the current state of the output.

isOn

```
p_Output.MyOutput.isOn()
```

Returns true if the output is currently true.

Example

When entering a state, check if an output is already true.

```
def s_state_enter():
    if p_Output.MyOutput.isOn():
        print('MyOutput is on')
    else:
        print('MyOutput is off')
```

isOff

```
p_Output.MyOutput.isOff()
```

Returns true if the output is currently false.

Example

When entering a state, check the status of the output.

```
def s_state_enter():
    if p_Output.MyOutput.isOff():
        print('MyOutput is off')
    else:
        print('MyOutput is on')
```

Parameter Methods

All parameter methods have the form `p_Param.{PARAMETER_NAME}_write`. Type `p_` in the Pynapse Code Editor and let the code completion do the work for you. 'Par1' is the default name of the first parameter. The name of each `write` method gets replaced with the name of your actual parameter, so if the parameter is called 'Freq' then `p_Param.Freq_write(val)` is an available method.

Par1_write

```
p_Param.Par1_write(val)
```

Write a new value for this parameter.

Inputs	Type	Description
val	float	Floating point value to send to this parameter output

Example

Modify the wave frequency for an Audio Stimulation gizmo when a state changes.

```
class PrepStim:   #StateID = 0
    def s_State_enter():
        # get next stim ready
        wave_freq = 1000
        p_Param.p_Param.WaveFreq_write(wave_freq)
```

List_write

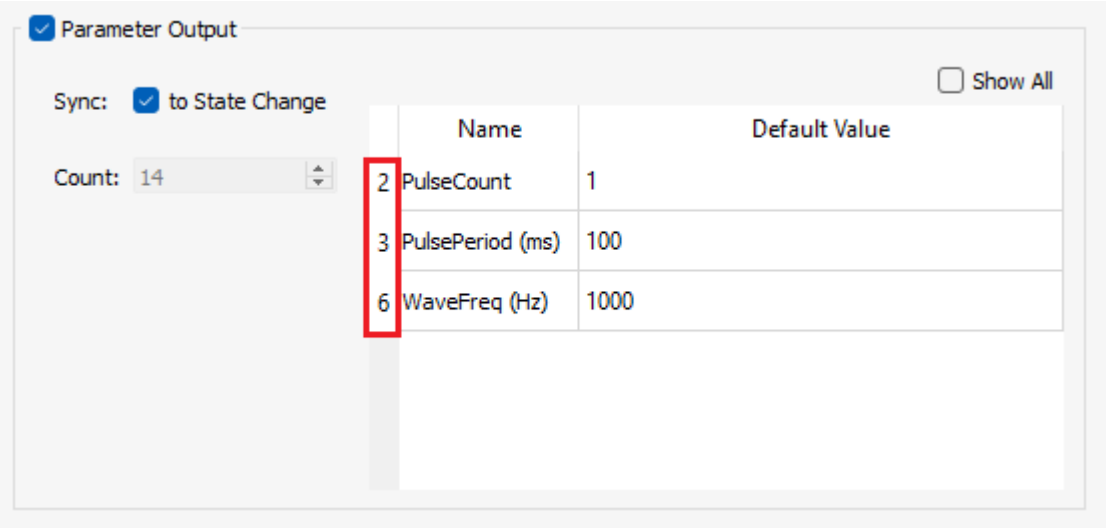
```
p_Param.List_write(vlist)
```

Write all the parameters at once using a list.

Inputs	Type	Description
vlist	list	List of floating point numbers to send to all parameter outputs

Note

In this example, the parameters that we can write are the 2nd, 3rd, and 6th values in the full parameter array. Be sure to include zeros for the parameters that aren't writable, as in the code example below.

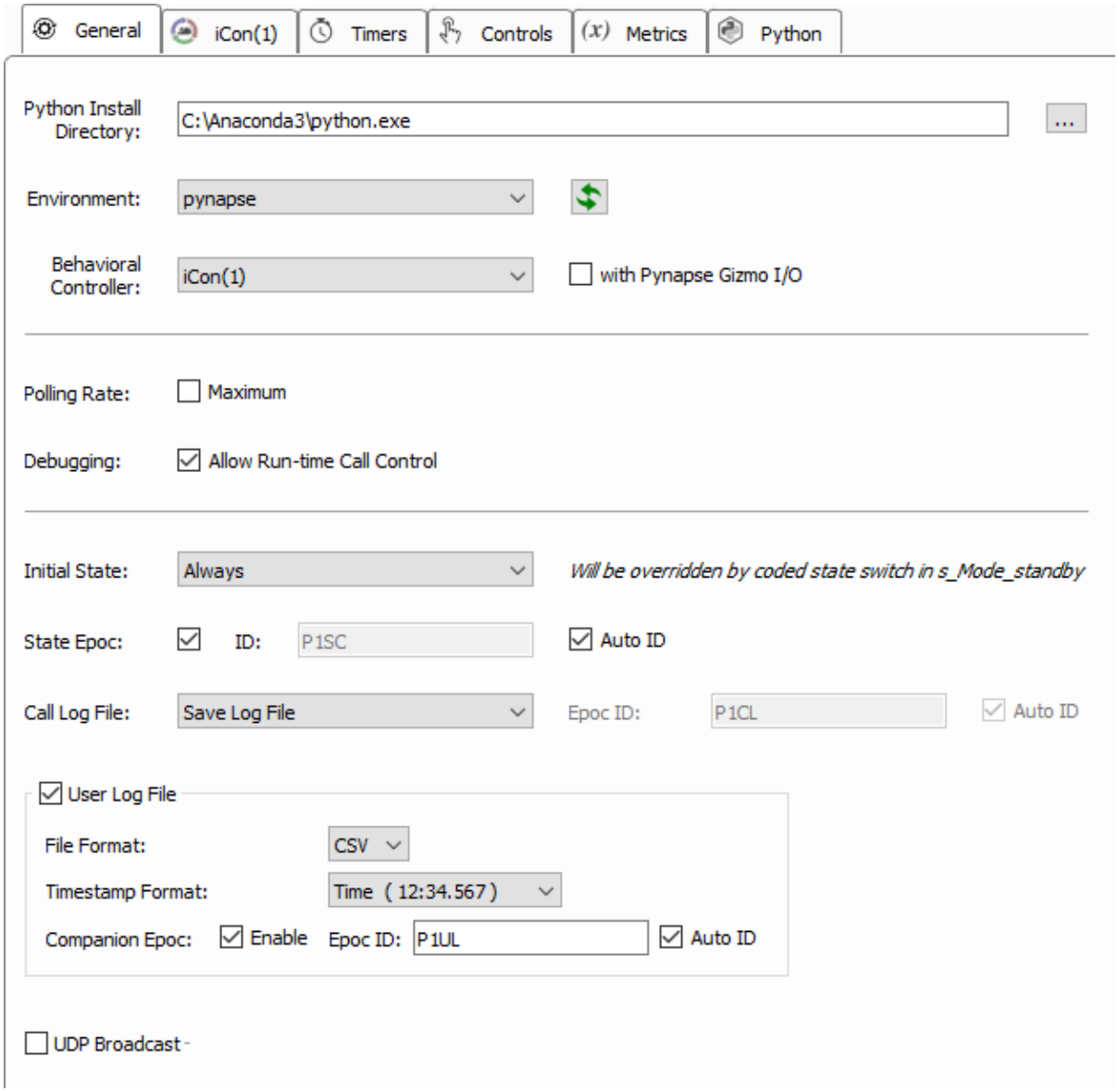


Example

Prepare a list of stimulation parameters for an Audio Stimulation gizmo when a state changes.

```
class PrepStim:   #StateID = 0
    def s_State_enter():
        # get next stim ready
        pulse_count = 3
        pulse_period = 200
        wave_freq = 500
        vals = [0, pulse_count, pulse_period, 0, 0, wave_freq]
        p_Param.List_write(vals)
```

General Tab



General Tab

Set the **Python Install Directory** and **Environment** for the Python interpreter. See Requirements for information on installing Python and setting up an environment for Pynapse. After these are defined in Pynapse, they will become the default python directory and environment used in future experiments. You can change this in the Preferences Dialog.

iCon Integration

If there is an iCon in your experiment, Pynapse will automatically attach to it and it will be selected in the **Behavioral Controller** list. This enables the iCon tab and the iCon Inputs and iCon Outputs asset classes in the Python editor and hides the Gizmo Inputs and Gizmo Outputs tabs.

If **Behavioral Controller** is set to **None**, the default Gizmo Inputs and Gizmo Outputs tabs and assets are available.

If you want to use a mix of iCon inputs/outputs and gizmo inputs/outputs in your Python code, check the **with Pynapse Gizmo I/O** box.

Polling Loop

The Pynapse event loop regularly polls the hardware for new information. The polling loop delay depends on the **Polling Rate** setting. The typical round-trip delays (read Pynapse input → set Pynapse output) are shown below.

Polling Rate Maximum	Delay
enabled	4-5 ms
disabled	~40 ms

For tighter behavioral state control, always enable Maximum Polling Rate.

> **Note**
>
> Maximum polling rate is not available when using Corpus hardware emulation

Debugging

Enable run-time debugging features so you can make manual function calls. See Run-Time and Debugging for more information. Turn this off when the experiment design is finished so you don't accidentally modify the experiment flow during run-time.

States

Initial State tells Pynapse which State to start it when the experiment first executes. If you have Python code in your Always state that triggers a state change when Synapse switches to Standby mode, this will override the state set here. See Synapse Control for more information.

State Epoc saves an epoc event with the timestamp and state number any time the state changes. This is used to correlate your e-phys and other data with behavioral states in post-processing.

Call Log File saves a file called `{GIZMO_NAME}_call_log.csv` in the block folder that contains timestamps and state information any time a slot is called. It's essentially a log of every behavioral event that happens during the recording. You can optionally save a timestamped epoc event in the data tank as well. 'Off' will disable the Call Log user interface, but the log file is still saved to disk.

User Log File

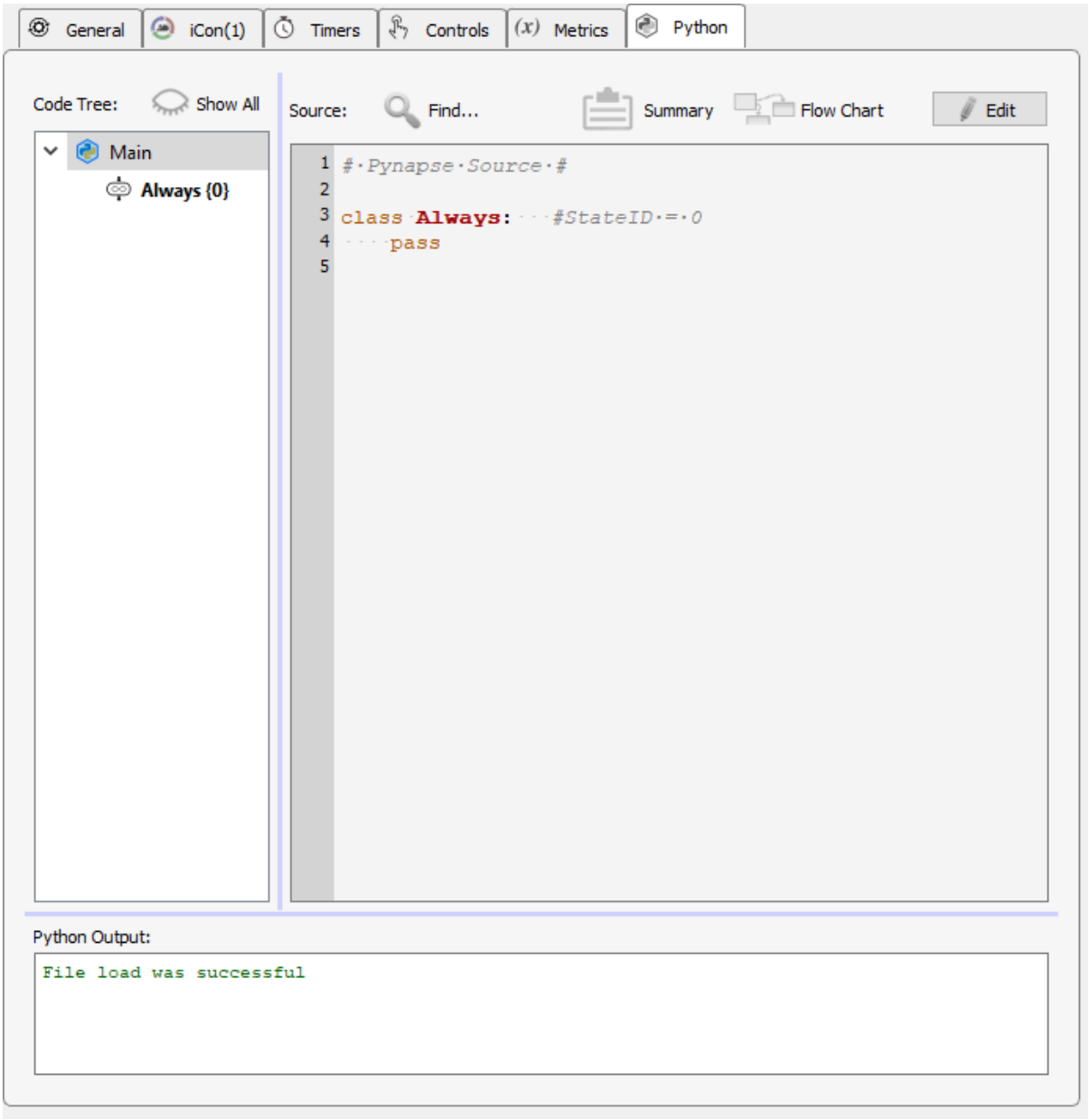
Enables automatic logging of Controls, Metrics, and Session details. Also adds a `p_Log` asset in Python to write your own entries. See Logs for more details.

UDP Broadcast

Enables a UDP broadcast packet with session details, so other applications or devices on the network can interact with the experiment. Also adds a `p_Udp` asset in Python to write your own entries. See UDP for more details.

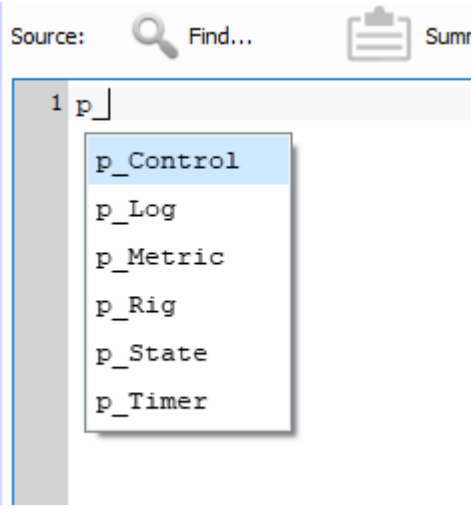
Code Editor and Parser

The built-in Python editor is where all your states and events are defined, telling the Pynapse event loop what do when events happen. Click the 'Edit' button, or double-click on the Source code, to enable editing.

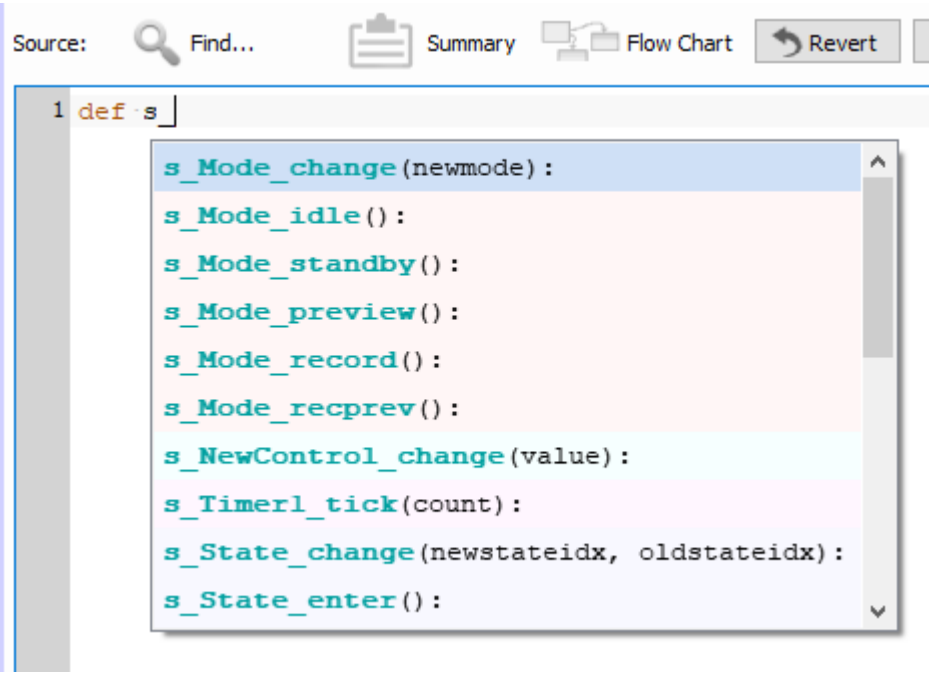


The built-in Python editor does code completion for you. Every time you press 'Commit' the parser dynamically generates a list of methods and event triggers you have access to based on the named inputs, outputs, controls, globals, and timers. All Pynapse assets start with `p_` and all slot methods start with `def s_`, so start there and the code completion will show you the available assets or slots.

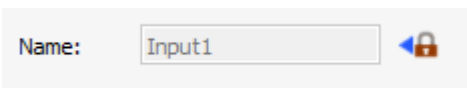
Assets Code Completion



Slots Code Completion



You can also right-click → "Help" on anything in the editor to show more complete documentation on the object under the cursor.

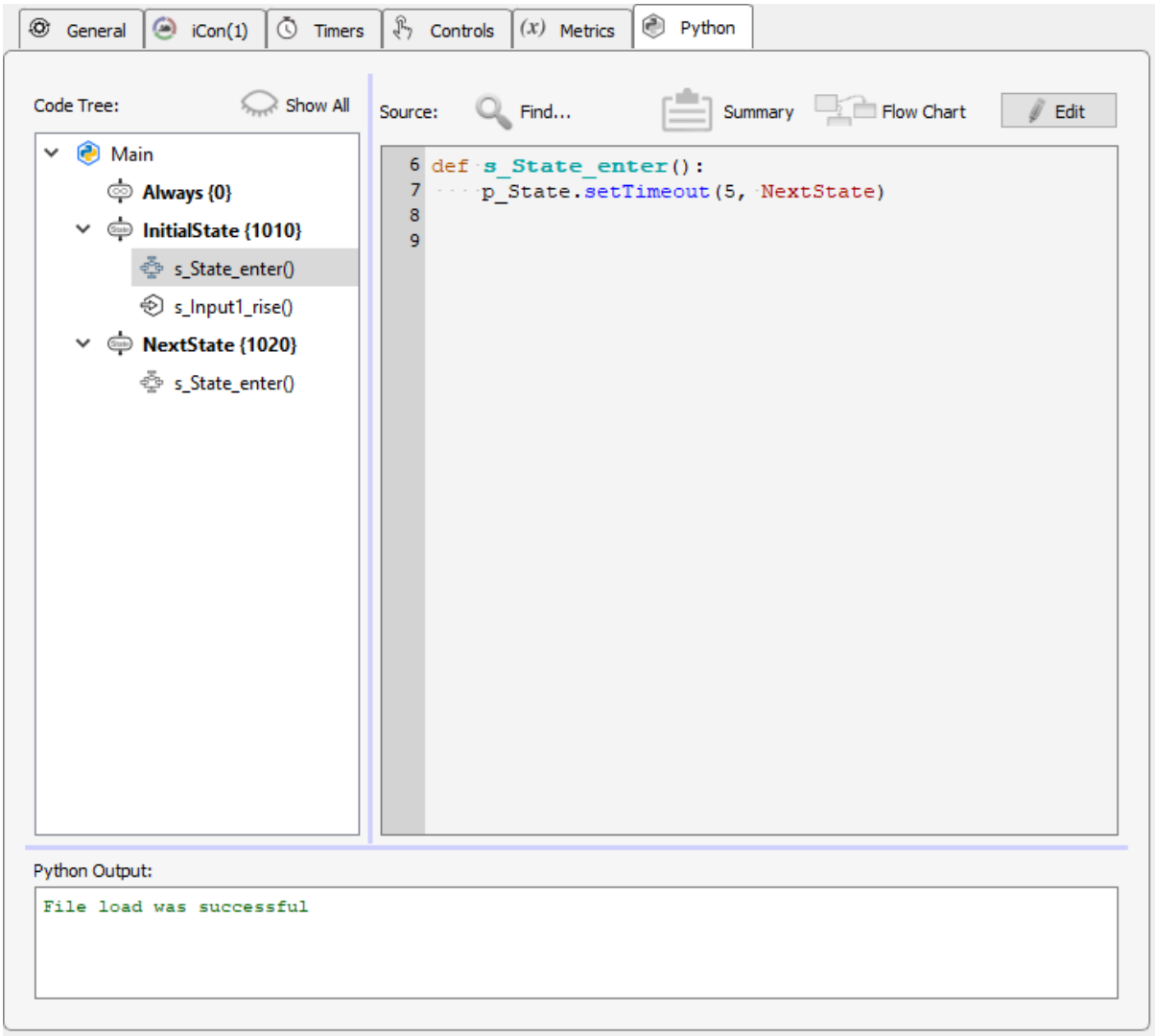


Asset names are linked to their method calls in the Python code. Assets that appear in the Python code will have a lock icon next to their names in their asset tab. If you decide to change the name of an asset after writing Python code that interacts with it, click the lock icon to change the name of the asset and update all of the instances of this name in the Python code.

Code Tree

The parser identifies all of the states and all of the methods that are written within the states that respond to events, and builds the Code Tree.

You can click on any item in the Code Tree and the Editor shows you just the selected state or methods so you are just editing that part of the code. If you want to look at the entire file, click 'Main' in the Code Tree.

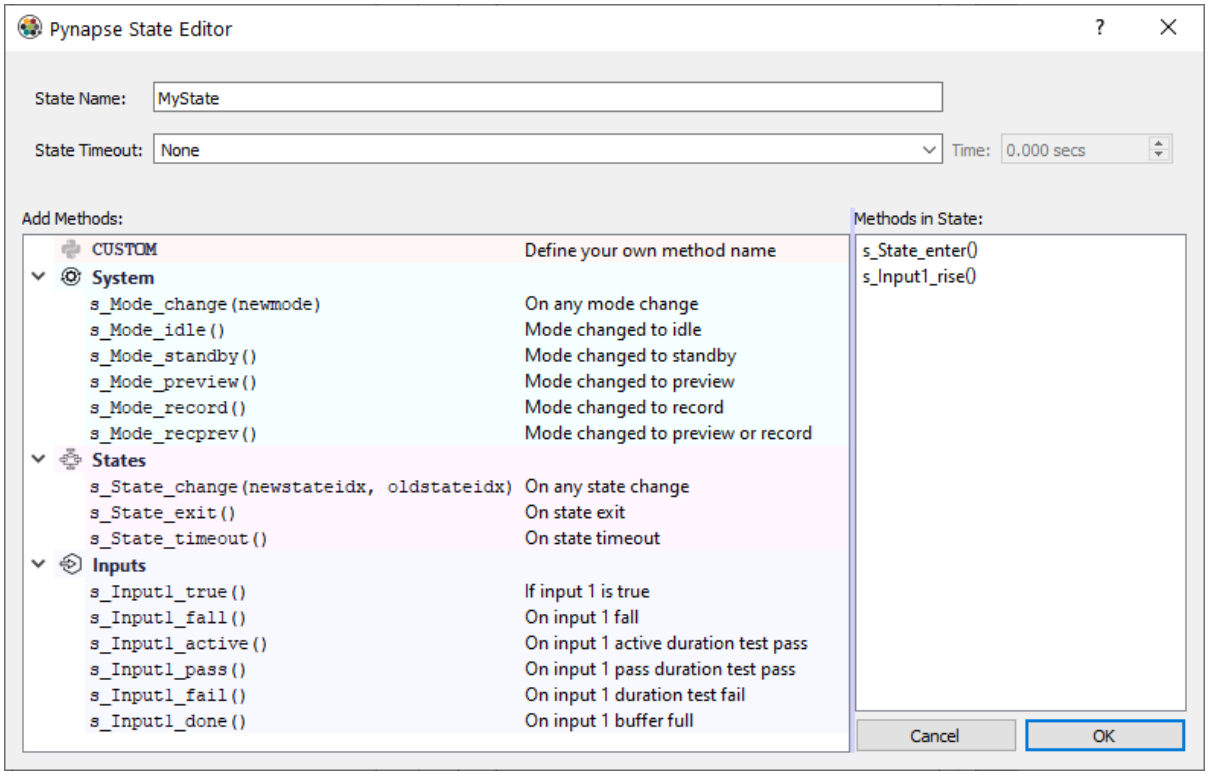


Note

By default the Code Tree shows only Pynapse states and slot methods. Sometimes you'll write methods inside states that aren't Pynapse slots. The 'Show All' button will include these methods in the Code Tree.

To Add a State

The easiest way to add a state is by right-clicking on an existing state in the Code Tree (or 'Main') and adding a state from the menu. This brings up a state creation wizard that shows you all the available hardware events you can capture with the state. You choose which methods to include in the code and a state timeout if desired. This saves you from having to remember the exact syntax for creating a state every time.



You can also edit the Python code directly with the required state structure:

```
class MyNewState:  #StateID = ?
    pass
```

Note

After you create a state, you can get back to the creation wizard by right-clicking the state name in the Code Tree → Reconfigure.

Working with StateIDs

Classes defined with the `#StateID = ?` comment are parsed as Pynapse states. If the StateID is `?` then Pynapse will automatically assign a number to the state for you. The StateIDs are shown next to the state name in the Code Tree.

> **Note**
>
> You can include regular classes in the code that aren't Pynapse states by excluding the `#StateID` comment from the class definition.

The StateID number is saved into the data tank when state changes occur during runtime. It is important that the StateIDs are consistent across recordings. If you make changes to your Pynapse source code and all of the StateIDs are `?` then these numbers will change if you add or remove states from your source code. This will make it harder to organize your data during post-processing if you are trying to compare data made with your newer experiment to recordings made with earlier versions of your experiment.

The solution is to lock the StateIDs to a value right in the comment, like `#StateID = 555`. If you already have code written with automatically generated StateIDs, you can lock the current StateIDs in place by right-clicking on 'Main' and select 'Commit State IDs'. This will overwrite all of the `#StateID = ?` comments with their assigned StateID, like `#StateID = 1010`.

> **Important**
>
> StateIDs have to be defined in order (top to bottom) in the Python source code. For example, this code is invalid because the state class defined with `#StateID = 2` is before `#StateID = 1`.

```
class Always: #StateID = 0
    def s_Mode_standby():
        p_State.switch(MyState1)

class MyState1: #StateID = 2
    def s_State_enter():
        print('MyState1')
        p_State.setTimeout(1, MyState2)

class MyState2: #StateID = 1
    def s_State_enter():
        print('MyState2')
        p_State.setTimeout(1, MyState1)
```

Flow Chart

As experiments get more complicated, it is helpful to see an overview of how the states, inputs, and outputs are connected. Click the Flow Chart button to see a graphical representation of all these links. Double-click on a state in the Flow Chart to show it in the code editor. Right-click the Flow Chart button to center the dialog on screen.

Summary

Click the Summary button to see a table view of all the Pynapse assets and their parameters - Inputs, Outputs, Metrics, Controls, etc. This is helpful to troubleshoot at a glance, and to help while coding your experiment. You don't have to keep flipping back and forth between tabs to get asset names. Right-click the Summary button to center the dialog on screen.

Organizing Your Code

Python code is stored either directly in the experiment or locally on your hard drive.

Python Code Blocks

'Main' is the default block of Pynapse code in the Code Tree saved with the experiment. You can have up to three other 'Local' Pynapse code blocks that are also saved in the experiment. These are not files on disk but rather saved with the experiment in the Synapse database. To add a code block, right-click on empty space in the Code Tree and select 'Add Local Pynapse Block'. You can import/export blocks if desired.

Python Local Files

You can import locally saved Python files from your hard drive into Pynapse. This is a convenient way to share common code between experiments. This also gives you a way to develop your own own classes/modules outside of Synapse and link to them from the experiment

You can link to as many existing files on disk as you want and they get imported automatically. To import a Python file, right-click on empty space in the Code Tree select 'Add Python Import File', and choose the Python file.

TDT Modules

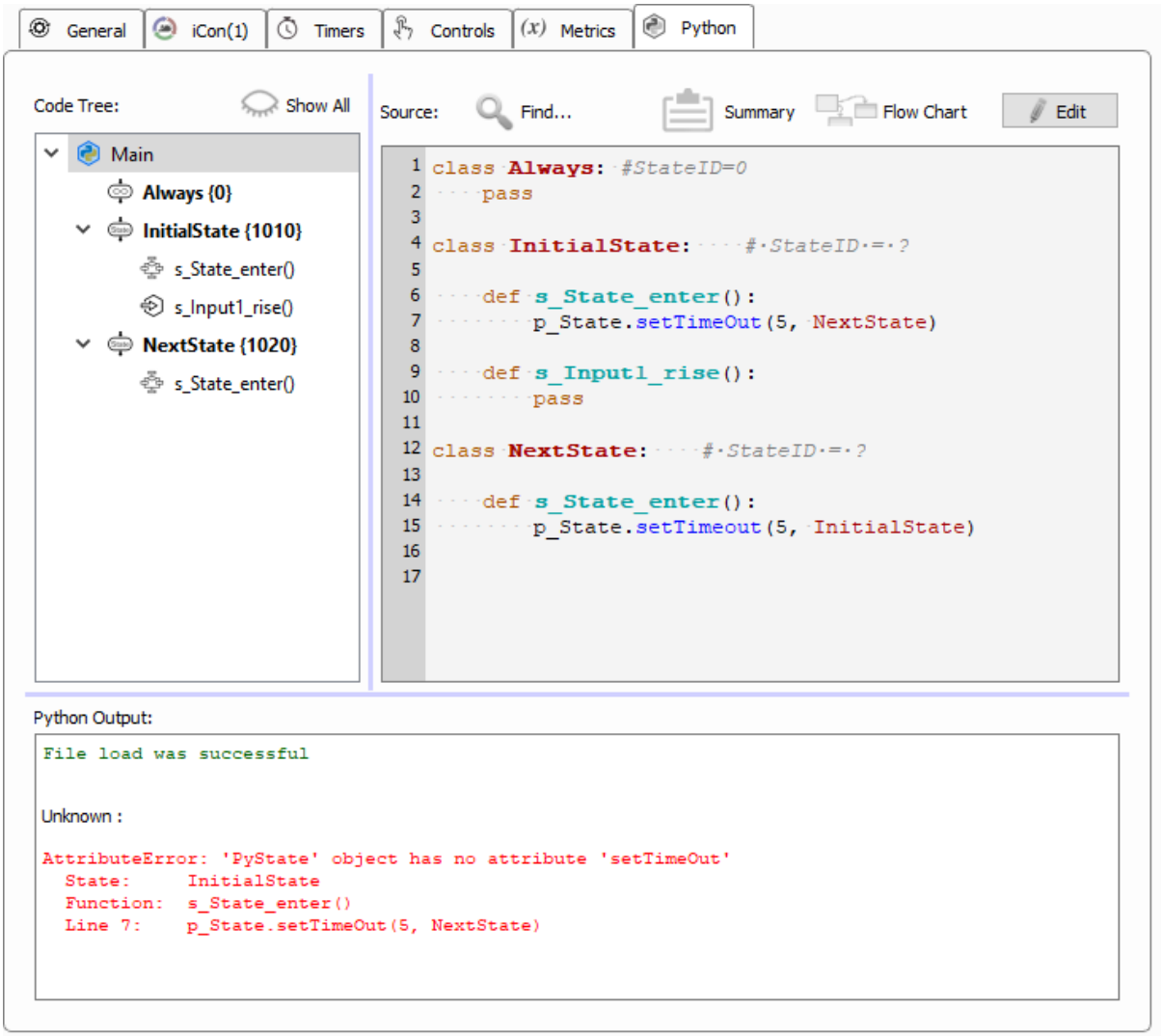
TDT provides several external modules for stimulus design and experiment templates for common operant conditioning protocols.

To make a TDT module available, right-click on empty space in the Code Tree and select 'Add Python Module'. This loads the list of modules from `C:\TDT\Synapse\PynapseLibs` and makes them selectable. The imported classes can then be instantiated in your Python code.

Testing

You can right-click on any state, method, or Python file in the Code Tree and select 'Test'. This will load the source file and run every method inside of it. Any obvious errors that will come up at run-time (like naming problems) are shown in the Python Output window at the bottom, with a reference to the method and line number causing the error.

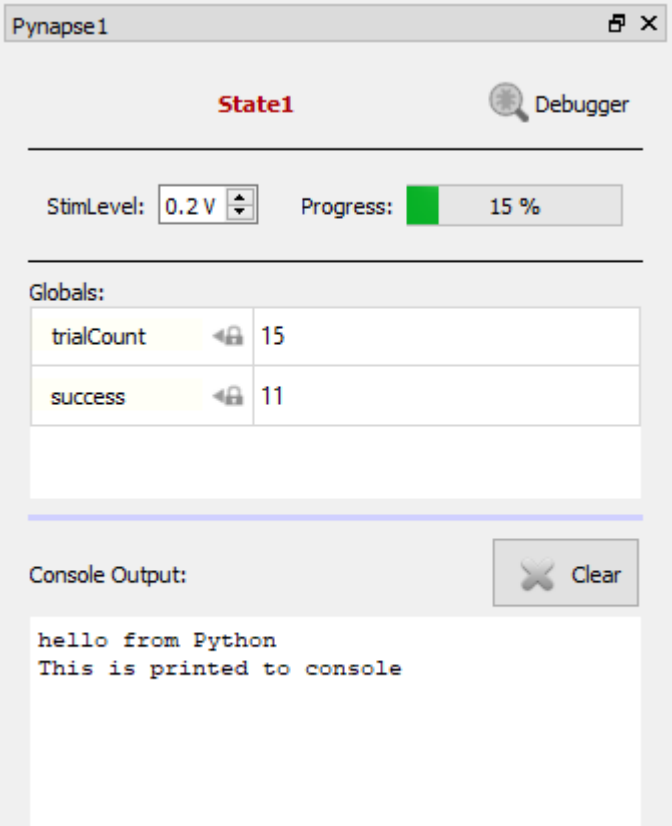
In the example below, `setTimeout` was incorrectly capitalized (should have used code completion!).



See Run-Time and Debugging for more debugging tips.

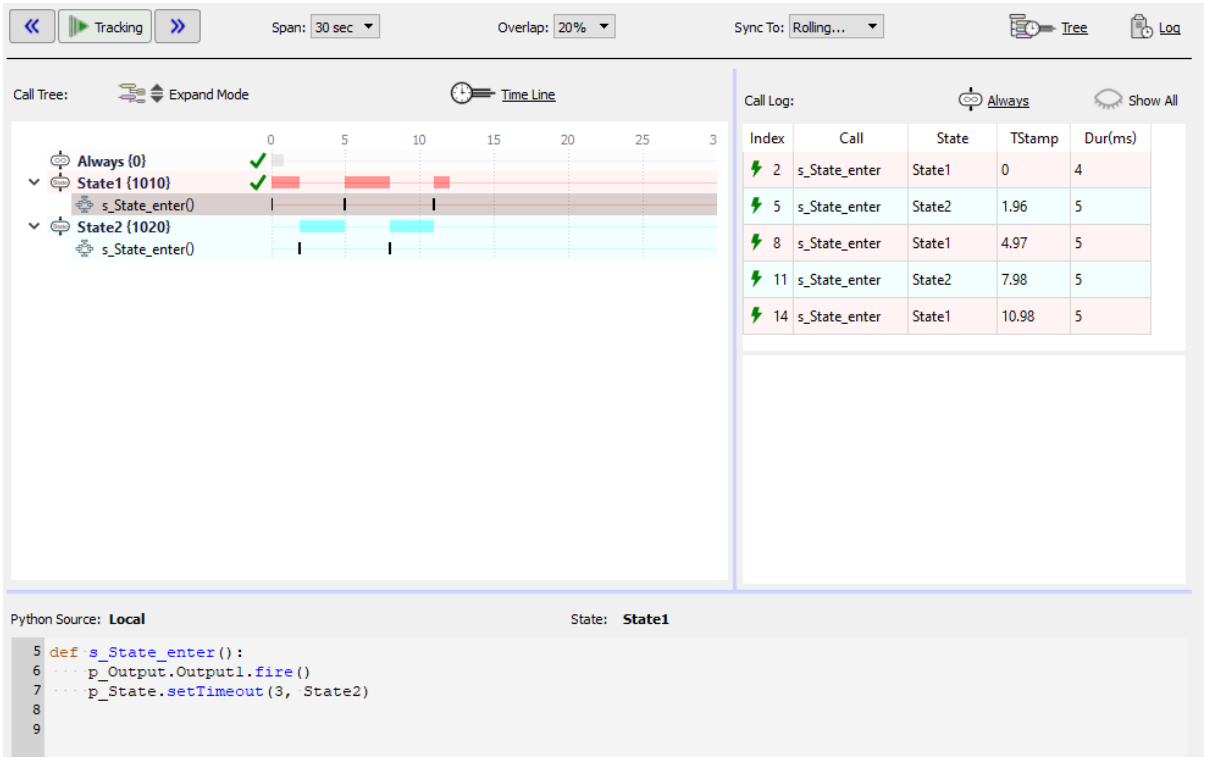
Run-Time and Debugging

The Pynapse run-time interface has two modes. The default view shows the current State in red text, any Controls and Metrics assets defined at design-time, and the Console Output shows any `print` outputs or error messages from the Python code.



Debug View

Click the **Debugger** icon to open the expanded run-time view. This gives you a more in-depth look at the events in Pynapse and allows you to manipulate the Pynapse state and manually trigger events during the experiment.



Call Tree

The Call Tree shows all the states and slot methods that Pynapse knows about. An icon next to the state/slot name gives you more status information:

Icon	Description
✓	Currently active state*
⚡	A green lightning bolt is briefly shown next to successful calls
⚠	A red warning sign is briefly shown next to calls that had errors

> **Note**
>
> * The Always state is always also active and slot methods within it can always be triggered by Pynapse events

Expand Mode determines what methods to show within the states.

Icon	Description
	Show only the states and no slot methods
	Show all slot methods within current state
	Show all slot methods within all states

Time Line

The Time Line shows when a state is active and when slot methods fired. If there was an error during the slot method, it will show a red bar. You can click on events in the time line to see more information about it in the Call Log.

The controls at the top can stop/resume the time line, or move backward/ forward in time for review.

Span is how much time to show in the time line. When it gets to the end it clears the time line and starts drawing from the left. **Overlap** is what percentage of the end of the previous time line to include when the time line refreshes.

Sync To can be used to reset the time line when a particular state change happens, so you can easily follow the events in a particular state. This is helpful for debugging rapid events using a short time **Span**.

Call Log

All Pynapse events are captured in the Call Log with a timestamp and the amount of time spent in that method.

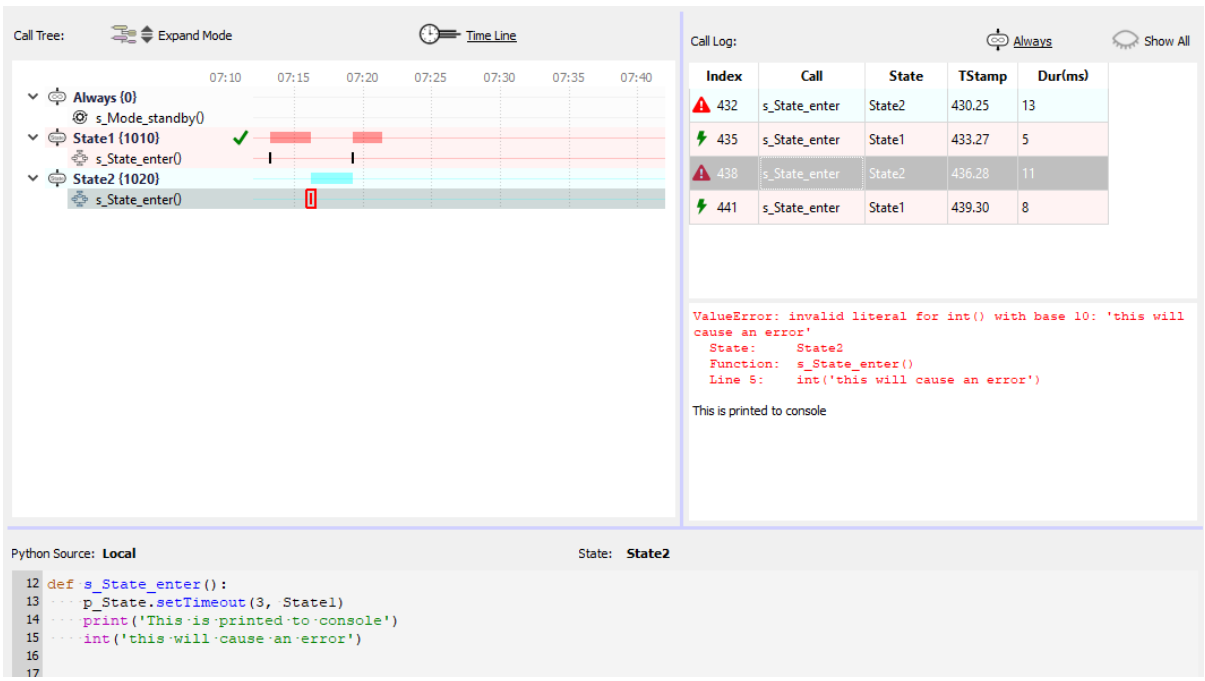
> **Tip**
>
> The information shown here is also optionally saved to a text file with the rest of your data. Enable the **Call Logging** setting in the General Tab at design-time.

Icon	Description
	You can optionally exclude the 'Always' state calls by turning off the **Always** button
	The Show All button will show every Pynapse event, including state timeouts, internal state changes, and global variable changes
	A green lightning bolt is shown next to successful calls
	A red warning sign is shown next to calls that had errors

If you click on an event in the call log, that event will be highlighted in the Call Tree, and vice versa. If it was an error, it will be outlined in red.

The window underneath the call log shows the console output for the selected call, including any error messages generated by that call, so you can track it down in the source code.

> **Note**
>
> Pause the time line so the Call Log doesn't refresh while you are trying to look at the error messages.



Debugging

Manual Control

You can manually control the state flow in the Call Tree. Double-click on any state to switch Pynapse into that state. The event will be capture in the Call Log.

You can also double-click on a slot method to trigger it manually. This will not be captured in the call log.

> **Important**
>
> The **Allow Run-Time Call Control** check box on the General Tab at design-time must be enabled for manual control to work.

Code Viewing

The bottom of the Debugging window is a view of the source code. Click on a state or a slot method in the Call Tree to see its source.

Tips and Tricks

Timeout Errors

If any method takes longer than ~3 seconds to execute, you will see this message in the Console Output window:

`Timeout error while waiting for response from Python kernel`

Avoid doing anything computationally intensive that takes longer than 3 seconds. Also, avoid using `time.sleep` statements as a way to control experiment flow. Use additional states and `setTimeout` to manage experiment flow.

Here's an example of poor design that will cause a timeout error:

```
import time
class State1:   #StateID = ?

    def s_Input1_rise():
        print('button pressed')
        time.sleep(5)  # this will cause Timeout error
        print('do something')
```

Instead, use the Pynapse state machine to keep track of the time delay for you:

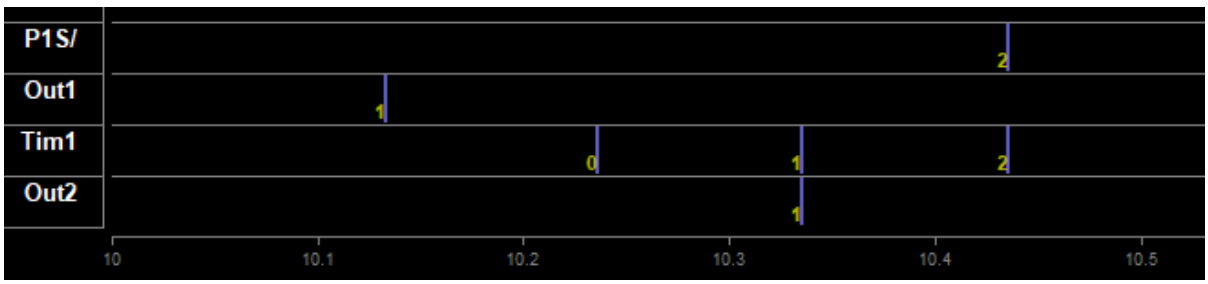
```
class State1:   #StateID = ?

    def s_Input1_rise():
        print('button pressed')
        p_State.setTimeout(5, State2)

class State2:
    def s_State_enter():
        print('do something')
```

Synchronizing Events

By default, all outputs in the Python code are executed sequentially as they are written. In the example below, outputs and timers are turned on in a slot method. The `time.sleep` statements are used for demonstration purposes to exaggerate the effect by adding additional latency between each call.

```
import time
class MyState1: #StateID = 1
    def s_MyInput_rise():

        p_Output.MyOutput1.fire()
        time.sleep(0.1)

        p_Timer.MyTimer1.setPeriod(.1)
        p_Timer.MyTimer1.setRepeats(3)
        p_Timer.MyTimer1.start()
        time.sleep(0.1)

        p_Output.MyOutput2.fire()
        time.sleep(0.1)

        p_State.switch(MyState2)
```

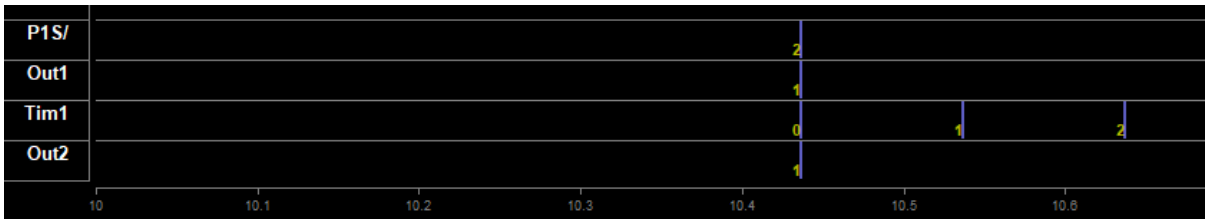
The two outputs and the timer have 'Epoc Save' turned on. The runtime output looks like this:



Event	Description
P1S/	State change event timestamp
Out1	Output1 fired
Tim1	Timer1 ticked
Out2	Output2 fired

You can see the noticeable 100 ms gaps in between the output events, and all of these events occur before the state change ('P1S/' = 2 in this example).

For coordinating stimulus events or anything else that has to happen on the hardware simultaneously, the Outputs, Parameters, and Timers have a **Sync to State Change** option. If all of the outputs and timers in the last example had this option enabled, then the result looks like this:



The `sleep` delays are still there but now all outputs fire precisely when the state changed to 2.

> **Important**
>
> The `p_State.switch` statement must come *after* any calls to set the timers or outputs for this to work properly.

Delays

The polling loop delay depends on the 'Polling Rate' setting in the Pynapse General Tab. The typical round-trip delays (read Pynapse input → set Pynapse output) are shown below.

Polling Rate Maximum	Delay
enabled	4-5 ms
disabled	~40 ms

For tighter behavioral state control, always enable Maximum Polling Rate.

> **Note**
>
> Maximum polling rate is not available when using Corpus hardware emulation

> **Note**
>
> If using the SynapseAPI class in Pynapse, there is a variable delay that ranges from 5 to 30 ms. If the computer is under heavy processing, there can be delay spikes up to ~100 ms.

> **Important**
>
> Metric and Control asset 'writes' go through the SynapseAPI and have a longer delay.
>
> Any calls that 'read' an asset value (except for Metric which are python variables) also go through the SynapseAPI.

Run-time Plots

If you would like to do online plotting or make your own custom GUIs then matplotlib and ipykernel==4.10.1 must also be installed in your Python environment.

If you want to plot something on screen using matplotlib you must include this line of code at the top of your Python code:

```
%matplotlib
```

> **Note about using a different matplotlib backend**
>
> If for some reason you need to set the matplotlib backend is set, once it is set it cannot be changed for the entire interpreter session. For example, if in between recordings you change `%matplotlib qt` to `%matplotlib tk`, the second statement is ignored and qt backend will be used. If Pynapse code gets modified such that a different backend is used, a complete restart of Synapse is required.

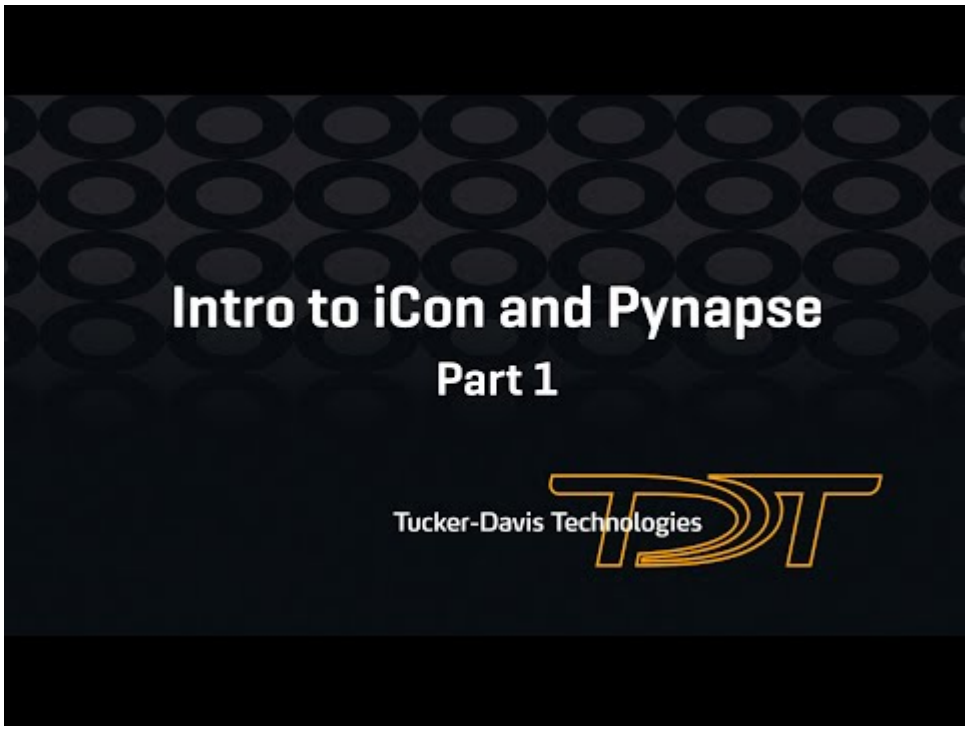
Pynapse Training Videos

Introduction

Intro to iCon & Pynapse: Part 1

Intro to iCon & Pynapse: Part 1

Connect your iCon hardware and configure your Synapse rig. See an overview of the Pynapse gizmo and create your first experiment using Pynapse.



Intro to iCon & Pynapse: Part 2

Intro to iCon & Pynapse: Part 2

Use the Pynapse state machine, Metrics, and Session Manager to create a more complicated experiment.

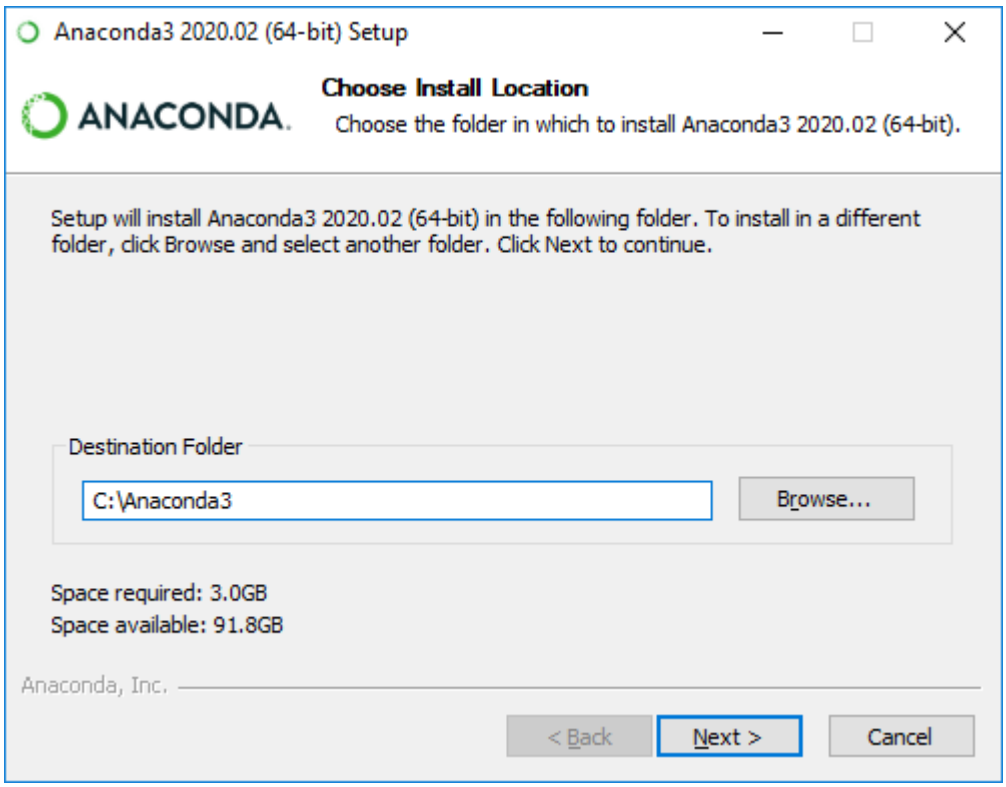
Installing Anaconda Python

Anaconda installs from https://docs.anaconda.com/anaconda/install/windows/

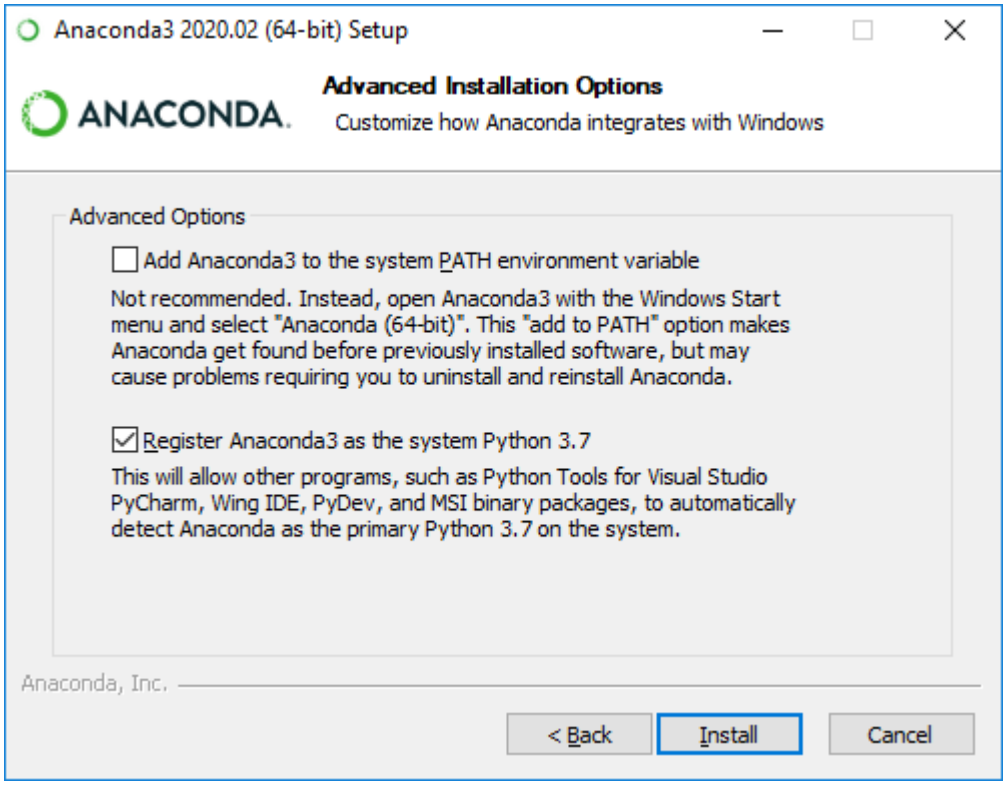
> **Note**
>
> We recommend installing 64-bit Python on 64-bit Windows 10. This is the Python 3.x "64-Bit Graphical Installer" option.
>
> On 32-bit machines, install the Python 3.x "32-Bit Graphical Installer".

1. During installation:
    a. Select 'Install for All Users'
    b. Change the installation directory to C:\Anaconda3

Example Installer 1



Example Installer 2



Environments

A Python virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages. The default environment is called `base`.

Environments are simply directories on disk, so it is easy to delete/recreate environments if they get in a bad state. It is more difficult to do this with the `base` environment. It is therefore recommended that you create a specific environment for Pynapse to use.

Environments are created in Anaconda using an Anaconda Prompt. Note that you can specify the Python version for each environment as well.

Here's how to create a Python 3.7 environment to use in Pynapse:

1. In Windows, go to Start → "Anaconda Prompt (Anaconda3)". This starts you in the `base` environment.
2. If you installed Anaconda 3.8 or higher, then the `base` environment needs to be modified to support Pynapse environment integration.

    a. Type this in the command prompt:

```
pip install ipykernel==4.10.0 pyzmq==19.0.1 jedi==0.17.0
```

3. Type this to create the Python 3.7 environment called 'pynapse':

```
conda create --name pynapse python=3.7 ipykernel=4.10.0 pyzmq=19.0.1
jedi=0.17.0
```

4. The fresh environment is mostly empty, so you'll want to add common packages. From the Anaconda Prompt, first type this to activate the new environment:

```
conda activate pynapse
```

5. Then install some of the libraries we'll want to use. For example, install the tdt package for data analysis:

```
pip install tdt
```

> **Important**
>
> Anytime you want to add packages to an environment, do it through the Anaconda Prompt and activate your environment first.

6. If you want to add your own custom runtime plotting to Pynapse, create the environment with this:

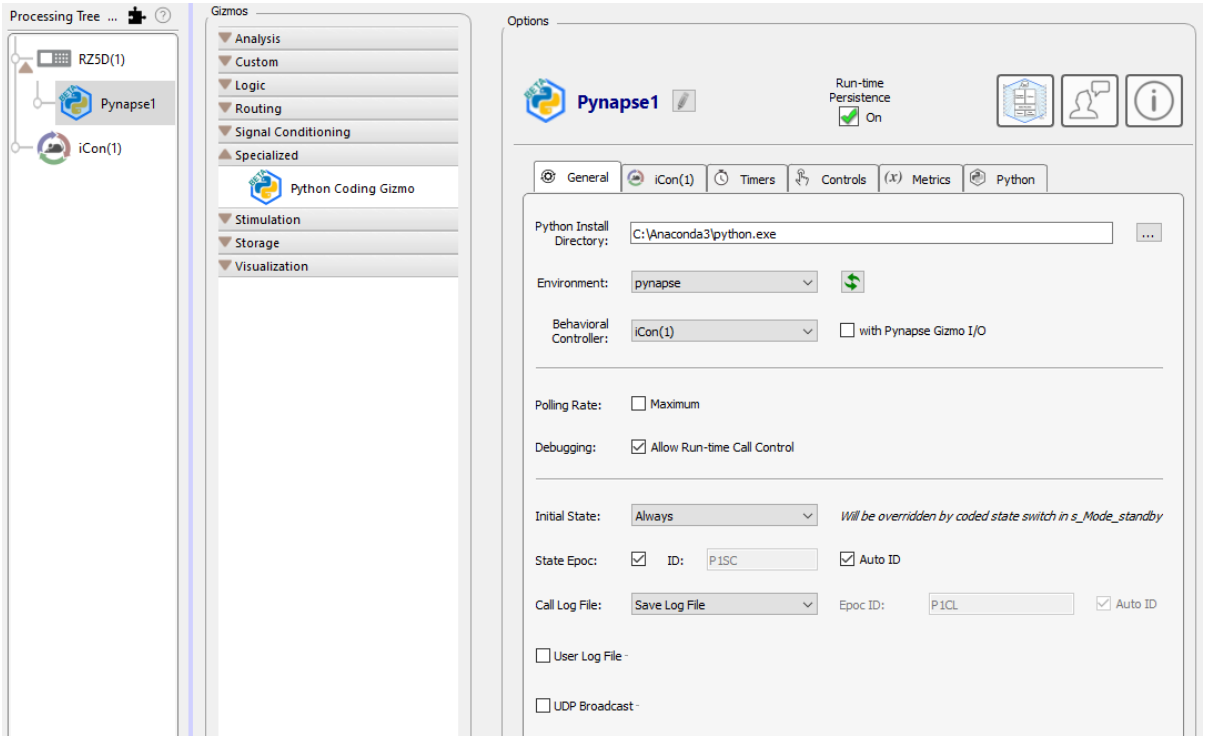
```
pip install matplotlib
```

Pynapse Setup

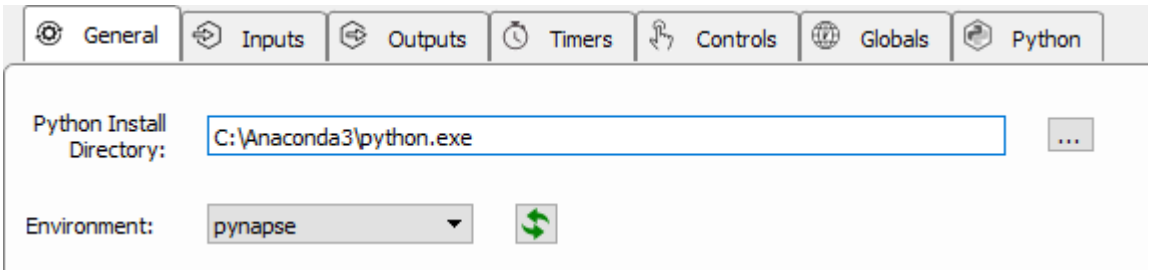
1. In Synapse, go to Menu → Preferences and set the Python Directory to

```
C:\Anaconda3
```

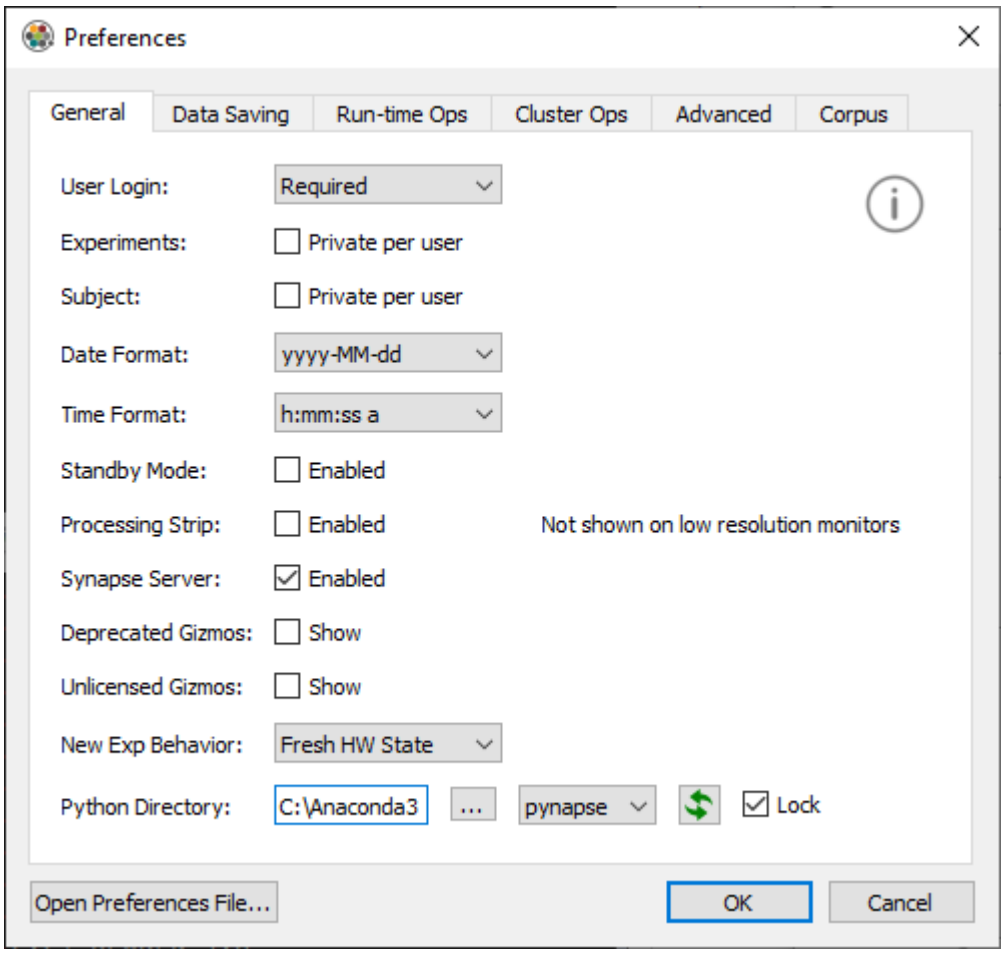
2. Add a Python Coding Gizmo from the Specialized gizmo list.



3. The Pynapse General Tab has a drop down that shows the environments created in Anaconda, so you can choose the environment directly in the GUI. Click the refresh button and choose the `pynapse` environment we just created. Commit the change.



4. In Synapse, go to Menu → Preferences. The Python Directory and Environment will have updated to the path you just used. Select 'Lock' so this path and environment are the defaults whenever you use a Pynapse gizmo.

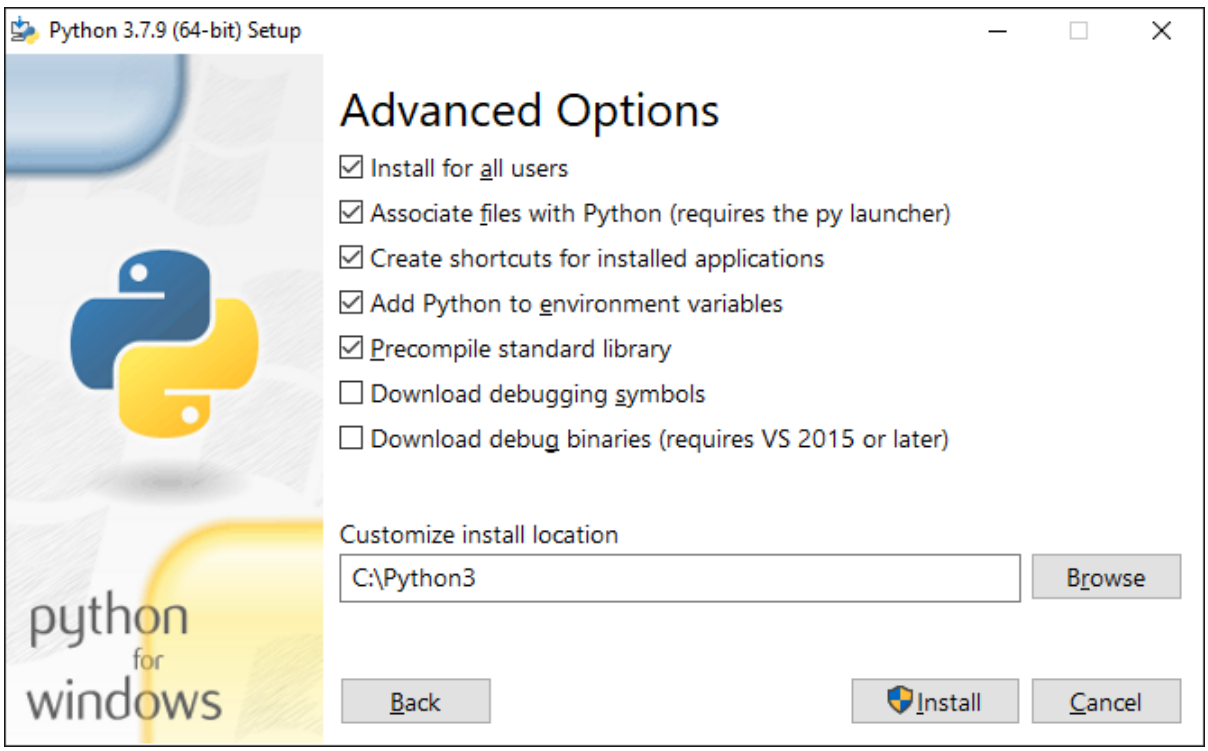

Preferences
General
Data Saving
Run-time Ops
Cluster Ops
Advanced
Corpus
User Login:
Required
Experiments:
Private per user
Subject:
Private per user
Date Format:
yyyy-MM-dd
Time Format:
h:mm:ss a
Standby Mode:
Enabled
Processing Strip:
Enabled
Not shown on low resolution monitors
Synapse Server:
Enabled
Deprecated Gizmos:
Show
Unlicensed Gizmos:
Show
New Exp Behavior:
Fresh HW State
Python Directory:
C:\Anaconda3
...
pynapse
Lock
Open Preferences File...
OK
Cancel

Installing Standard Python

Standard python.exe installs from https://www.python.org/downloads/windows/. Pynapse works best with Python 3.7.

> **Note**
>
> We recommend installing 64-bit Python on 64-bit Windows 10. This is the Python 3.7.x "Windows x86-64 executable installer" option. The latest available version of 3.7.x is 3.7.9.
>
> On 32-bit machines, install the Python 3.7.x "Windows x86 executable installer".

1. During installation, select 'Customize Installation'
    a. Select 'Install for All Users' and 'Add Python to environment variables'
    b. Change the installation directory to C:\Python3



Environments

A Python virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages. The default environment is called 'base'.

Environments are simply directories on disk, so it is easy to delete/recreate environments if they get in a bad state. It is more difficult to do this with the 'base' environment. It is therefore recommended that you create a specific environment for Pynapse to use.

Environments are created in standard python.exe installation with the venv module.

From the command line:

```
python -m venv C:\Python3\envs\pynapse
```

To install pip packages from the command line (Start → cmd), you first activate the environment. Pynapse requires a few packages with specific versions (see https://github.com/ipython/ipykernel/issues/358 and https://github.com/ipython/ipykernel/issues/518 for more information).

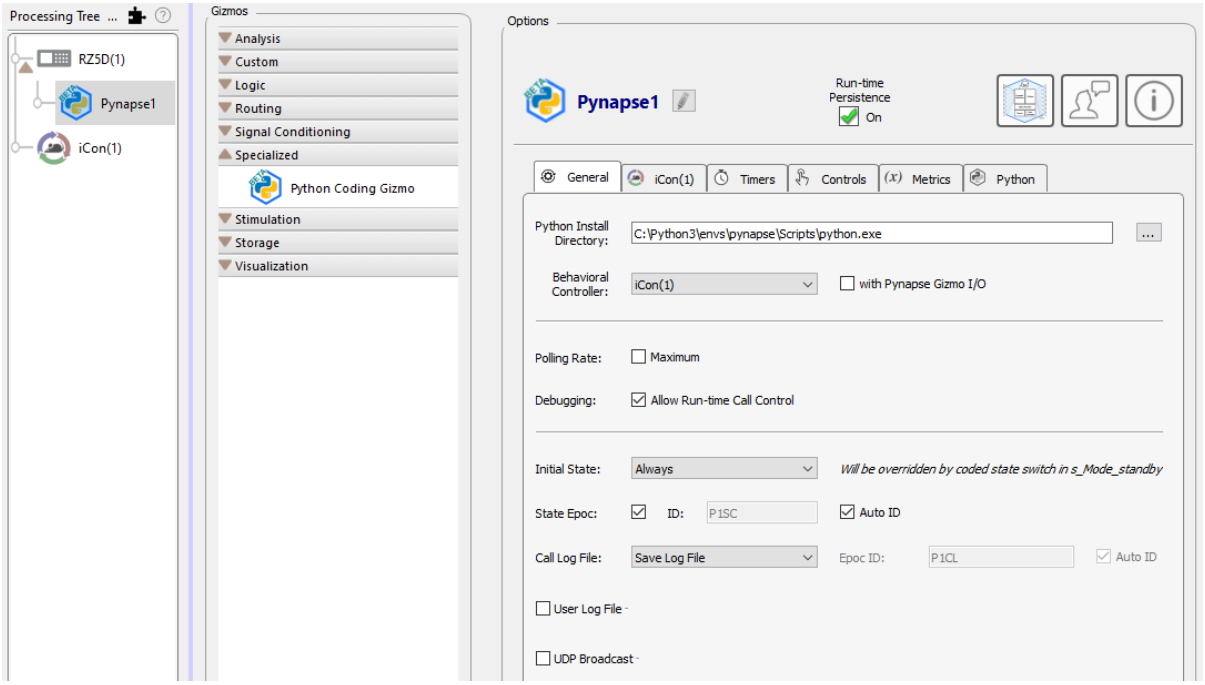
```
C:\Python3\envs\pynapse\Scripts\activate.bat
pip install ipykernel==4.10.0 pyzmq==19.0.1 jedi==0.17.0
ipython_genutils==0.2.0
```

The tdt package is useful for offline data analysis:

```
pip install tdt
```

Pynapse Setup

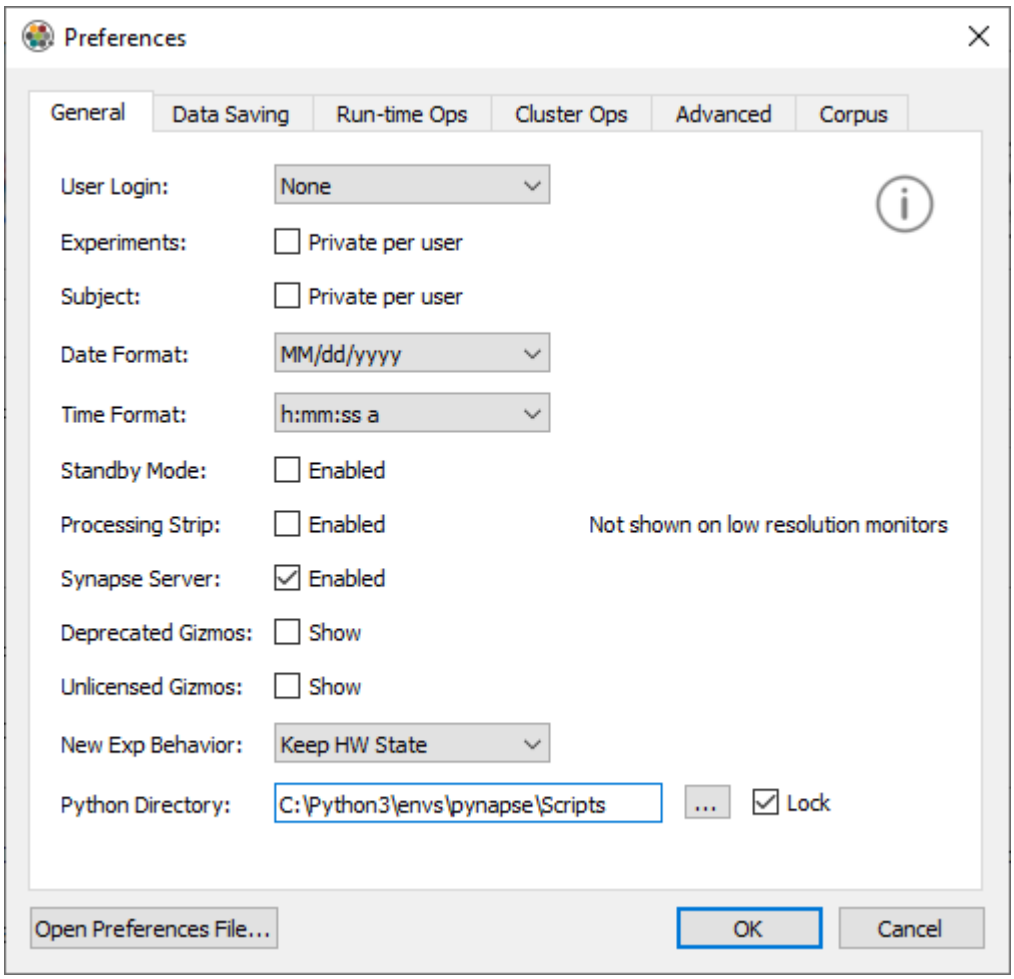
1. In Synapse, add the Python Coding Gizmo from the Specialized gizmo list.



2. In Pynapse General Tab, set the Python Install Directory to your installed Python environment, which in this case is:

   `C:\Python3\envs\pynapse\Scripts\python.exe`

3. In Synapse, go to Menu → Preferences. The Python Directory will have updated to the path you just used. Select 'Lock' so this path is the default path whenever you use a Pynapse gizmo.


Preferences
General
Data Saving
Run-time Ops
Cluster Ops
Advanced
Corpus
User Login:
None
Experiments:
Private per user
Subject:
Private per user
Date Format:
MM/dd/yyyy
Time Format:
h:mm:ss a
Standby Mode:
Enabled
Processing Strip:
Enabled
Not shown on low resolution monitors
Synapse Server:
Enabled
Deprecated Gizmos:
Show
Unlicensed Gizmos:
Show
New Exp Behavior:
Keep HW State
Python Directory:
C:\Python3\envs\pynapse\Scripts
...
Lock
Open Preferences File...
OK
Cancel