# Puppy Python Controller Documentation

*Release 1.0*

**Matthias Baumgartner**

October 07, 2013

# CONTENTS

There's this robot, Puppy, a four-legged artificial dog. Although it is actually built and can be controlled in the real-world, some applications require its simulated version. This may for example be to prevent the motors from damage or to do hundreds of experiments in short time. A popular choice for simulating such robots, is [Webots]. Webots already provides a Python interface to implement arbitrary simulation controllers. The PuPy module builds on top of that but also abstracts from the concrete Puppy robot model. The focus lies on implementing various controllers, while the robot model is assumed to be already available. This way, it provides a simple facility to further abstract from the simulation setup and offers an easy way to implement a new controller for Puppy. The idea is to offer an abstraction such that most of the controllers are easy to implement. More involved or in some sense special controllers may be off the grid and therefore still require more complex implementation. Yet, especially when starting work on Puppy, this module should make your life much easier.

The module is written in pure Python (version 2.7). Since [Webots] requires two controllers - one for the supervisor and one for the robot - the module's core consists of two classes: an overlay for the `Robot` and `Supervisor`. These are the main abstractions from the Webots interface. Especially the `Robot` provides a natural way of building a controller in the well-known sense-think-act cycle. It also deals with the communication to the simulator, exonerate the programmer with simulation-related issues. It offers a way to build a controller within a pure Python/NumPy environment. For the `Supervisor`, typical use cases have been prepared, also abstracting from the Webots interface. With these core functions, it should be possible to implement a simple controller within minutes and a minimal amount of code.

To run the module, some more libraries are required. See the *Download* page for more information.

CHAPTER

# ONE

# CONTENTS

## 1.1 Webots ecosystem

**Contents**

### 1.1.1 Introduction

The [Webots] simulator already offers Python bindings. The reference manual can be found here.

Besides the models of the environment and of course the robot, Webots requires two controllers which implement the robot and simulation logic. The robot model is not part of this library but assumed to be available. The focus of this module lies on the controllers. The robot controller implements the robot's logic. If there's no such controller, the robot will just remain on its place. It's the controller's responsibility to define motor targets (possibly with respect to sensory readouts) on behalf of which the simulator will act. The Supervisor controller offers a global view (thus it may access some information, the robot may not) and allows to start, restart, revert or quit the simulation. As hinted, it is used to supervise the experiment. Both controllers have to be implemented in a seperate file and correctly linked to a webots world (see Supporting material) in order to be executed.

**Warning:** When running, the Supervisor and Robot controllers are executed as seperate processes. When the simulation is reverted (or quit, naturally), both processes are terminated. This introduces all the common issues with concurrency and inter process communication. Keep this in mind and consider restarting instead of reverting a simulation. Also make use of the intrinsic communication channels.

Webots offers Python bindings, but the respective module is only available within a Webots execution. Since we don't want to force this module to be within Pythons default module search path, the PuPy module must not directly depend on it. Due to this, a bit more complex approach must be taken. The library offers two classes (`WebotsPuppyMixin`, `WebotsSupervisorMixin`) which have to be mixed into Webot's Robot and Supervisor class at runtime. To do so more easily, a builder is offered for both (`robotBuilder()`, `supervisorBuilder()`).

For example, in the controller script, setting up a working instance is as easy as this:

```
>>> from controller import Robot
>>> import PuPy
>>> robot = PuPy.robotBuilder(Robot, <args>)
```

with `<args>` the arguments to `WebotsPuppyMixin`. The example works analogous for the Supervisor. If you subclass `WebotsPuppyMixin` or `WebotsSupervisorMixin`, the working instance can be set up through the `builder()` function.

## 1.1.2 Reference

`PuPy.`**`mixin`**`(`*cls_mixin*, *cls_base*`)`

Create a class which derives from two base classes.

The intention of this function is to set up a working *webots* controller, using webots's Robot class and the Puppy mixin from this library.

**`mixin`** The mixin class, e.g. *WebotsPuppyMixin* or *WebotsSupervisorMixin*

**`base`** The base class, e.g. *Robot* or *Supervisor*

`PuPy.`**`builder`**`(`*cls_mixin*, *cls_base*, **args*, ***kwargs*`)`

Set up a mixin class and return an instance of it.

`PuPy.`**`robotBuilder`**`(`*cls_base*, **args*, ***kwargs*`)`

Return an instance of a webots puppy robot.

`PuPy.`**`supervisorBuilder`**`(`*cls_base*, **args*, ***kwargs*`)`

Return an instance of a webots puppy supervisor.

## 1.1.3 Supporting material

When experimenting with several controllers and environments, webots requires a world file (and some others) for each configuration. Usually, these differ only in some aspects while most of the content is identical. Also, the controller is part of the configuration, making it very annoying to keep track of subtle differences and working versions. To support development, two small scripts are provided which take care of this problem.

### Splitter

Starting with a world (including the robot and environment), it is first split into its parts and each one stored with a certain name. This way, information about the robot or world can be extracted and later identified. For example, you can store several versions of the robot without redundant data. This script extracts the desired information from the world file and stores it in an own data structure. Specify what parts should be saved and under what name.

**Note:** The script requires the robot to be named *puppy*.

```
#!/bin/bash

# TODO: Copy stuff like textures, prototypes, ...

usage()
{
cat << EOF
usage: $0 options <Input world file>
```

```
   <DESC>

OPTIONS:
   -a      Save terrain, puppy, supervisor (give name)
   -t      Save terrain (give name)
   -p      Save puppy (give name)
   -s      Save supervisor (give name)
   -c      Save controllers (give path)
   -b      Target base directory (give path)
   -h      Show this message
EOF
}

SAVECONTROLLER=0
BASE="/home/matthias/studium/master/work/webots/data"
INPUT=
TERRAIN=
PUPPY=
SUPERVISOR=
CTRL=

while getopts “a:t:p:s:c:b:h” OPTION
do
     case $OPTION in
         a)
             TERRAIN=$OPTARG
             PUPPY=$OPTARG
             SUPERVISOR=$OPTARG
             ;;
         t)
             TERRAIN=$OPTARG
             ;;
         p)
             PUPPY=$OPTARG
             ;;
         s)
             SUPERVISOR=$OPTARG
             ;;
         c)
             CTRL=$OPTARG
             ;;
         b)
             BASE=$OPTARG
             ;;
         h)
             usage
             exit 1
             ;;
         ?)
             echo "?"
             usage
             exit
             ;;
     esac
done

shift $((OPTIND-1))
INPUT=$@
```

```
# check input
if [[ ! -e $INPUT ]] || [[ -z $BASE ]]
then
     usage
     exit 1
fi

# analyze file
START_PUPPY=`cat $INPUT | grep -n '^DEF puppy Robot' | awk '{split($0,a,":"); print a[1]}'`
START_SUPER=`cat $INPUT | grep -n '^Supervisor' | awk '{split($0,a,":"); print a[1]}'`
LINES=`wc -l $INPUT | awk '{split($0,a); print a[1]}'`

if [ $START_PUPPY -gt $START_SUPER ]; then
    echo "Cannot split the world file; Please reformat"
    exit 1
fi

HEAD=`head -n $[$START_PUPPY-1] $INPUT`
ROBOT=`head -n $[$START_SUPER-1] $INPUT | tail -n $[$START_SUPER - $START_PUPPY]`
SUPER=`tail -n $[$LINES-$START_SUPER+1] $INPUT`

mkdir -p $BASE/terrain $BASE/puppy $BASE/supervisor

if [[ ! -z $TERRAIN ]]; then echo "$HEAD" > $BASE/terrain/$TERRAIN; fi
if [[ ! -z $PUPPY ]]; then echo "$ROBOT" > $BASE/puppy/$PUPPY; fi
if [[ ! -z $SUPERVISOR ]]; then echo "$SUPER" > $BASE/supervisor/$SUPERVISOR; fi

if [[ $CTRL ]]
then
    PCTRL=`echo "$ROBOT" | grep 'controller' | sed 's/^.*controller\s\+"\(.*\)"/\1/'`
    SCTRL=`echo "$SUPER" | grep 'controller' | sed 's/^.*controller\s\+"\(.*\)"/\1/'`

    PTH=`readlink -f $INPUT`
    PTH=`dirname $PTH`
    mkdir -p $CTRL
    cp $PTH/../controllers/$PCTRL/$PCTRL.* $CTRL/puppy
    cp $PTH/../controllers/$SCTRL/$SCTRL.* $CTRL/supervisor

    # get the filename extension of the scripts
    #filename=$(basename "$fullfile")
    #extension="${filename##*.}"
    #filename="${filename%.*}"
fi
```

### Builder

You have to define a robot and supervisor controller and choose an environment. The script will set up a simulator world from these arguments and start webots. This way, controllers (and environments) may be mixed quickly without having several files with almost the same content. The controller scripts are symlinked, so they can be edited while the world is active in webots. Naturally, the changes take effect after reverting the simulation.

**Note:** This script is written for common Linux systems (it requires bash and symlinks). Other plattforms may not be supported.

**Note:** The supplementary files (terrain, plugins, prototypes, etc.) must be installed in the same location as the script itself. If this behaviour is not desired, set the `$BASE` variable to the correct directory. You can generate some of these files with the Splitter script above. Also check out the Downloads section.

**Note:** If you wish to create a snapshot of an active controller, the Splitter script may be useful. It copies the controllers to some location, usually outside of your common working directory.

```
#!/bin/bash

usage()
{
cat << EOF
usage: $0 -c <puppy controller> -s <supervisor controller> [optional options] <output>

   Note: It may be convenient to set the default value of -b in
         this script manually (default is script location)

OPTIONS:
   -c      Puppy controller
   -s      Supervisor controller
   -t      Terrain
   -p      Puppy
   -f      Overwrite target
   -b      Directory of your webots data
   -m      starting mode of webots: stop, realtime (default), run or fast
   -h      Show this message
EOF
}

WEBOTS="/usr/local/bin/webots"
BASE="$(dirname "$(readlink "${BASH_SOURCE[0]}")")/data"

TRG=

PCTRL=
SCTRL=
WORLD="default"
PUPPY="default"
SUPERVISOR="default"
MODE="stop"
while getopts "hp:s:t:c:b:m:" OPTION
do
     case $OPTION in
         h)
             usage
             exit 1
             ;;
         p)
             PUPPY=$OPTARG
             ;;
         t)
             WORLD=$OPTARG
             ;;
         c)
             PCTRL=$OPTARG
             ;;
         s)
             SCTRL=$OPTARG
             ;;
         b)
             BASE=$OPTARG
             ;;
         m)
             MODE=$OPTARG
             ;;
         ?)
             usage
             exit
             ;;
     esac
done

shift $((OPTIND-1))

# target
if [[ -z $@ ]]; then
    usage
    exit 1
fi

TRG=$@

if [[ -e $TRG ]]; then
    echo "WARNING: Target exists. Overwriting"
    rm -R $TRG
fi

# check input files
if [[ -z $PCTRL ]] || [[ -z $SCTRL ]] || [[ -z $TRG ]]
then
     usage
     exit 1
fi

# check working directory
if [[ ! -e $BASE/terrain ]] || [[ ! -e $BASE/puppy ]] || [[ ! -e $BASE/supervisor ]] || [[ ! -e $BAS
then
    echo "Base directory seems not to fit. Please change your working directory"
    echo "The current base is $BASE"
    usage
    exit 1
fi

# check other input files
if [[ ! -e $BASE/terrain/$WORLD ]]; then echo "WORLD not found"; exit 1; fi
if [[ ! -e $BASE/puppy/$PUPPY ]]; then echo "PUPPY not found"; exit 1; fi
if [[ ! -e $BASE/supervisor/$SUPERVISOR ]]; then echo "SUPERVISOR not found"; exit 1; fi

# check mode argument
if [[ ! $MODE == "stop" ]] && [[ ! $MODE == "realtime" ]] && [[ ! $MODE == "run" ]] && [[ ! $MODE ==
then
    echo "The webots run mode is $MODE, but must be one of (stop, realtime, run, fast)."
    exit 1
fi
```

```
# create data structure
mkdir -p $TRG
cp -R $BASE/supplementary/* $TRG
mkdir -p $TRG/controllers/

# set up world file
cat $BASE/terrain/$WORLD > $TRG/worlds/puppy_world.wbt
cat $BASE/puppy/$PUPPY | sed 's/controller ".*"/controller "puppyController"/' >> $TRG/worlds/puppy_w
cat $BASE/supervisor/$SUPERVISOR | sed 's/controller ".*"/controller "supervisorController"/' >> $TR

# copy controller
mkdir -p $TRG/controllers/puppyController
mkdir -p $TRG/controllers/supervisorController
PPTH=( $PCTRL ) # make array
SPTH=( $SCTRL )
PPTH[0]=`readlink -f "${PPTH[0]}"` # replace controller file by absolute path
SPTH[0]=`readlink -f "${SPTH[0]}"`
PPTH="${PPTH[@]}" # rejoin array elements to single string
SPTH="${SPTH[@]}"
cat $BASE/controllers/generic/generic.py | sed "s|^CMD=''|CMD='$PPTH'|" > $TRG/controllers/puppyContr
cat $BASE/controllers/generic/generic.py | sed "s|^CMD=''|CMD='$SPTH'|" > $TRG/controllers/supervisor
#rm $TRG/controllers/puppyController/puppyController.py 2>/dev/null
#rm $TRG/controllers/supervisorController/supervisorController.py 2>/dev/null
#ln -s $PPTH $TRG/controllers/puppyController/puppyController.py
#ln -s $SPTH $TRG/controllers/supervisorController/supervisorController.py


# start webots
cd $TRG
$WEBOTS --mode=$MODE worlds/puppy_world.wbt
```

### 1.1.4 Downloads

The above scripts are deployed together with the Puppy model, originally developed and contributed to this project by Matej Hoffmann and Stefan Hutter. All files can be retrieved from the the *Download* page (see *Available downloads*).

## 1.2 Supervisor

**Contents**

### 1.2.1 Introduction

Each simulation run is observed and possibly influenced by a Supervisor. It has complete access of all the Robot's and environment's properties and can exclusively execute certain actions (such as start, stop or record the simulation). To implement such a supervisor, one needs to set up a script and link it to the webots world (see *Webots* *ecosystem*). Within the script, the module `controller` can be imported (at runtime only), specifically its class `controller.Supervisor`. It provides the basic Webots interface.

The mixin `WebotsSupervisorMixin` extends the base functionality of `controller.Supervisor` by a `WebotsSupervisorMixin.run()` method. Within this method, the infinite control loop is executed. It only returns shortly before the simulation is terminated. In the loop, a number of user-defined checks is executed periodically. These checks are the main tool to customize the supervisor, since they are also allowed to invoke supervisor actions. The checks are executed one after another with no guaranteed order. They can be interpreted as a list of Strategies. A check should evaluate its condition (generally speaking, execute its code) and return if no action has to be taken. When reverting or restarting, it is advised to go through the predefined structures to ensure consistency.

For the typical use cases, checks have been predefined. They all are successors of `SupervisorCheck` and further divide into `RevertCheck` and `RespawnCheck`, based on their effect. However, note that checks are defined in a way that also allows them to be functions instead of classes, thus this hierarchy is optional.

### 1.2.2 Example

The supervisor facility is intended to be used from within the webots simulator [Webots]. The simulator automatically appends the controller module to Python's search path, such that the `Supervisor` class can be imported. Of course, the `PuPy` module is also made available:

```
>>> from controller import Supervisor
>>> import PuPy
```

The supervisor is capable of executing some checks from time to time and react correspondingly. Two checks are added, one that limits the time of the experiment and another one to catch a tumbled robot. Both checks revert the simulation if the respective condition is met:

```
>>> checks = []
>>> checks.append(PuPy.RevertTumbled(grace_time_ms=2000))
>>> checks.append(PuPy.RevertMaxIter(max_duration_ms=20*50*2*20))
```

With these checks, the supervisor can be created. For this, one has to go through the respective builder. This is because the webots module is not available within PuPy. In the case of the supervisor, `supervisorBuilder()` needs to be called with `Supervisor` as its first argument and the `WebotsSupervisorMixin`'s arguments following.

```
>>> s = PuPy.supervisorBuilder(Supervisor, 20, checks)
```

With this, there's a supervisor instance `s` which now can be started. The call to `WebotsSupervisorMixin.run()` will only exit when the simulation is somehow aborted.

```
>>> s.run()
```

### 1.2.3 Reference

**class** `PuPy.`**`WebotsSupervisorMixin`**(*sampling_period_ms*, *checks=None*)

Webots supervisor 'controller'. It actively probes the simulation, performs checks and reverts the simulation if necessary.

**`sampling_period_ms`** The period in milliseconds which the supervisor uses to observe the robot and possibly executes actions.

**`checks`** A list of callables, which are executed in order in every sampling step. A check's interface must be compliant with the one of `SupervisorCheck`.

**run**()
Supervisor's main loop. Call this function upon script initialization, such that the supervisor becomes active.

The routine runs its checks and reverts the simulation if indicated. The iterations counter is made available to the checks through *WebotsSupervisorMixin.num_iter*.

Note, that reverting the simulation restarts the whole simulation which also reloads the supervisor.

**class** `PuPy.`**SupervisorCheck**
A template for supervisor's checks.

**__call__**(*supervisor*)
Evalute the check and implement the consequences.

**supervisor** The supervisor instance. For communication back to the robot, an *emitter* is available through *supervisor.emitter*.

**__weakref__**
list of weak references to the object (if defined)

**class** `PuPy.`**RevertCheck**
Bases: `PuPy.webots.SupervisorCheck`

A template for supervisor's revert checks.

**revert**(*supervisor*)
Revert the simulation.

**class** `PuPy.`**RevertMaxIter**(*max_duration_ms*)
Bases: `PuPy.webots.RevertCheck`

Revert the simulation if a maximum duration is exceeded.

**max_duration_ms** Maximum time a simulation may run, in milliseconds. After this limit, the simulation is reverted.

**class** `PuPy.`**RevertTumbled**(*grace_time_ms=2000*)
Bases: `PuPy.webots.RevertCheck`

Revert the simulation if the robot has tumbled.

**grace_time_ms** Let the robot run after tumbling for some time before the simulation is reverted. In milliseconds.

**class** `PuPy.`**RespawnCheck**(*reset_policy=0*, *arena_size=(0, 0, 0, 0)*)
Bases: `PuPy.webots.SupervisorCheck`

A template for supervisor's respawn checks.

**reset_policy** Where to respawn the robot (0=center [0,0], 1=current position, 2=random position). Instead of literals, use *RespawnCheck._reset_center*, *RespawnCheck._reset_current* and *RespawnCheck._reset_random*. In case of random respawn the *arena_size* needs to be provided.

**arena_size** Size of the arena as list [min_x, max_x, min_z, max_z].

**respawn**(*supervisor*)
Reset the robots position.

**class** `PuPy.`**RespawnTumbled**(*grace_time_ms=2000*, **args*, ***kwargs*)
Bases: `PuPy.webots.RespawnCheck`

Respawn the robot if it has tumbled.

**grace_time_ms** Let the robot run for some time after tumbling before it is respawned. In milliseconds

**class** `PuPy.`**`RespawnOutOfArena`**(*distance=2000*, *arena_size=(0, 0, 0, 0)*, **args*, ***kwargs*)
Bases: `PuPy.webots.RespawnCheck`

Respawn the robot if it comoes too close to the arena boundary.

**`distance`** The robot's distance to the arena boundary.

**`arena_size`** Size of the arena as list [min_x, max_x, min_z, max_z].

**class** `PuPy.`**`QuitMaxIter`**(*max_duration_ms*)
Bases: `PuPy.webots.SupervisorCheck`

Quit webots if a time limit is exceeded. (in milliseconds!)

**`max_duration_ms`** Maximum running time, in milliseconds.

## 1.3 Robot

**Contents**

### 1.3.1 Introduction

The `WebotsPuppyMixin` class connects the Puppy robot model to the Webots' programming interface. It is designed such that it simplifies and unifies the development of a custom Puppy controller. Just like the *Supervisor*, the `WebotsPuppyMixin` has to be instantiated through a builder. At creation, the class requires an `actor` callback (which implements the controller) and some timings. The timings define the length of different epochs, namely the intervalls at which the `actor` is executed and how many sensor readings will be available. The callback must be compliant with `RobotActor`. Note that although the interface is specified in a class structure, it is designed in a way that also fits a plain function.

Similar to `WebotsSupervisorMixin`, the `WebotsPuppyMixin.run()` method implements the main, infinite control loop. Within the loop, a sense-think-act cycle is running. Sensor data are automatically read, so the sense-part is taken care of. Since the think and act parts are problem-dependent, they have to be implemented by the `actor`. Each call to the `actor` must return an iterator for the next of motor targets. These are incrementally enforced on the motors, such that the `actor` may indeed define the robot's behaviour. To do so, the sensor readings between two calls are supplied. To see how this works exactly, consult the documentation of `WebotsPuppyMixin` and `RobotActor`.

The `WebotsPuppyMixin` abstracts from the Webots API in the sense that it takes care of initialization and readout of Puppy's sensors. Thus, the `actor` may focus on the pure controller implementation. Some simple controllers have been prepared, for them see the *Control* page.

### 1.3.2 Example

In the controller script, Webots' `controller` module will be present, so it is imported together with the `PuPy` module.

```
>>> from controller import Robot
>>> import PuPy
```

As a simple illustrative walking pattern, a `Gait` is initialized. Details are not relevant right here (see the *Control* page for that), this just specifies a way of moving around.

```
>>> gait = PuPy.Gait({
>>>     'frequency' : (1.0, 1.0, 1.0, 1.0),
>>>     'offset'    : ( -0.23, -0.23, -0.37, -0.37),
>>>     'amplitude' : ( 0.56, 0.56, 0.65, 0.65),
>>>     'phase'     : (0.0, 0.0, 0.5, 0.5)
>>> })
```

With this specification, an `actor` can be set up. Again, it's not important what the actor concretely does but its mere existence: It implements the act-step in the sense-think-act cycle. Again, for details see *Control*.

```
>>> actor = PuPy.ConstantGaitControl(gait)
```

When everything is ready, the robot instance is created through the respective builder, as already mentioned. The first argument is Webots' `Robot` class, followed by arguments of `WebotsPuppyMixin` (`actor` in this case).

```
>>> r = PuPy.robotBuilder(Robot, actor)
```

The instance is ready, its main loop awaits execution. Let's do this as the last script line:

```
>>> r.run()
```

When the simulation terminates (i.e. reverts or quits), the main loop will actually be broken and the above call returns. Make sure that if there's code below this point, it doesn't prevent termination. There's a one second timeframe before the script gets killed by Webots.

### 1.3.3 Reference

**class** `PuPy.`**`WebotsPuppyMixin`**`(`**args*, ***kwargs*`)`

Bases: `PuPy.webots.WebotsRobotMixin`

The actual Puppy Robot implementation.

**`event_handlers`** List of receiver callbacks. Only allowed as keyword argument. The callbacks must implement the `event_handler_template()` interface.

**class** `PuPy.`**`WebotsRobotMixin`**`(`*actor*, *sampling_period_ms=20*, *ctrl_period_ms=2000*, *motor_period_ms=None*, *event_period_ms=None*, *noise_ctrl=None*, *noise_obs=None*`)`

Webots Robot controller. It samples all sensors and periodically consults an `actor` for control decisions.

**`actor`** A function which determines the motor targets for the next control period. See *RobotActor* for specifics.

The function must return an interator which is valid for at least *ctrl_period_ms* / *motor_period_ms* steps. In each step, it must return a list of four motor targets.

The actor's interface is defined by `RobotActor`. Note however, that the interface is organized such that the class structure may be obsolete.

**`sampling_period_ms`** The period according to which sensors are sampled. In milliseconds.

**`ctrl_period_ms`** The period of control actions. In milliseconds. Must be a larger than or equal to the motor period and, if larger, a multiple thereof.

**`motor_period_ms`** The period according to which motor targets are set. In milliseconds. Usually, the same as *sampling_period_ms* (the default). If not, it's advised that it's a multiple of the sampling period, otherwise the observations per control decision may become funny.

**event_period_ms** The period in milliseconds that is used for polling the receiver. Should optimally be a multiple of the control or sampling period or the Supervisor's sampling period.

**noise_ctrl** Additive zero-mean gaussian noise on the motor targets. Additional to whatever *webots* does. Either a scalar of 4-tuple is expected, which represents the noise variances for all of the motors or each individual one, respectively. Use None to discard the noise (default).

**noise_obs** Additive zero-mean gaussian noise on the motor targets. Additional to whatever *webots* does. Either a scalar of dict is expected, which represents the noise variances for all of the sensors or each individual one, respectively. In the latter approach, the dict keys have to correspond to the sensor name. Use None to discard the noise (default).

**add_emitter**(*name*)
Add an emittor called `name`.

**add_motor**(*name*, *device*)
Add a motor `name` to the robot with the corresponding webots node `device`.

**add_receiver**(*receiver_name*, *callback=None*)
Add a `receiver_name` for polling. If a new message is available `callback` is to be called. If `callback` is a list, all its items will be called on the same message.

**add_sensor**(*name*, *clbk*, *dim=1*)
Add a sensor `name` to the robot. The function `clbk` reads out sensor value(s). Each readout must either produce one (`dim` = 1) or three (`dim` = 3) readout values.

**get_readout**()
Return labels and a generator for reading out sensor values. The labels and values returned by the generator have the same order.

**motor_names**()
Return the motor's names.

**run**()
Main controller loop. Runs infinitely unless aborted by webots.

The controller operates on a sense, think, act cycle. In every step - according to the sampling period - the sensors are read out (sense). This information is passed to the control decision machine (think) and its result used for updating the motors (act).

The `actor` is provided with all sensor readings, starting from the previous call up to the current one. The very last readout is equal to the current state of the robot. The motor targets (denoted by trg) are the ones that have been applied in the previous step. This can be interpreted as the target that caused the sensor readings. Also note, that the motor targets returned by the `actor` will be effective immediately, i.e. the first target will be set right after the `actor` was called. Changed in version 1365: After version 1365 (10eb3eed-6697-4d8c-9aac-32ebf1d36239), the behaviour of the main loop was changed: There's an initialization of the motor target and the target executed before the measurement (so long, it was executed after). We need to discuss and test this to be more specific about the behaviour.

**send_msg**(*msg*, *emittor_name=None*)
Send a message `msg` through a device `emittor_name`. If the `emittor_name` is `None`, the message will be sent through all available devices.

`PuPy.`**event_handler_template**(*robot*, *epoch*, *current_time*, *msg*)
Template function for event_handler function of `WebotsRobotMixin:`.

**robot** Instance of `WebotsRobotMixin` with the current robot.

**epoch** `dict` containing the current sensor readings.

**current_time** `int()` with the current time step in ms.

**msg** `str()` containing the event message.

## 1.4 Control

**Contents**

### 1.4.1 Introduction

Given the `supervisor` and `robot`, a working controller can be set up. As described on the *Robot* page, this is achieved by implementing an `actor`, compliant with the `RobotActor` interface. Again, note that the interface is desined such that it may be matched by either a class or a function.

The actor gets the sensor measurements of the last epoch (i.e. since the previous execution of the actor) and information about the timeframe of the next action. Its purpose is to come up with a a sequence of motor targets that will be applied next. Read the reference of `WebotsPuppyMixin` carefully, so that you understand the setup and effect of the timings.

Although the task of implementing a controller is generally problem dependent, some typical actors have been prepared. The most common control setup is based on `Gaits`. A gait is a sinewave, expressed through its parameters (amplitude, frequency, phase and offset). Some parametersets that induce some specific walking patterns have already been found in various works. Yet, the gait itself does not implement a controller but only a series of motor targets. Besides that, the controller may also decide which gait to apply out of several choices. Or it might be desired to switch the gait during the experiment. Some actors have been prepared for this purpose, their reference and explanation can be found below.

One actor that deserves special attention is `RobotCollector`. It is a transparent actor, meaning that it delegates the action decision to another one but only stores the sensory data in a file (also see the Example).

### 1.4.2 Example

This example is very similar to the one at *Robot*; make sure, you understand the robot aspects.

Again, it's first required to import some modules:

```
>>> from controller import Robot
>>> import PuPy
```

In this example, the robot movement is based on some predefined walking patterns. A pattern is defined through a `Gait`, which generates a specific sinewave. The parameters of the wave have to be passed to the `Gait` constructor. Since there are four motors, there are four values for each parameter.

```
>>> boundLeft = PuPy.Gait({
>>>     'frequency' : (1.0, 1.0, 1.0, 1.0),
>>>     'offset'    : ( -0.23, -0.23, -0.37,-0.37),
>>>     'amplitude' : ( 0.38, 0.65, 0.47, 0.65),
>>>     'phase'     : (0.1141, 0.0, 0.611155, 0.5)
>>> })
```

One pattern would be quite boring, so another one is defined:

```
>>> boundRight = PuPy.Gait({
>>>     'frequency' : (1.0, 1.0, 1.0, 1.0),
>>>     'offset'    : ( -0.23, -0.23, -0.37, -0.37),
>>>     'amplitude' : ( 0.65, 0.38, 0.65, 0.47),
>>>     'phase'     : (0.0, 0.1141, 0.5, 0.611155)
>>> })
```

The gaits only implements a sequence of motor targets, not an actor. One might just apply one constant pattern (there's `ConstantGaitControl` for that) but may also come up with other gait-based controllers. Here, the goal is to randomly switch between the two gaits. For this, the respective controller is initialized with the two gaits:

```
>>> actor = PuPy.RandomGaitControl([boundLeft, boundRight])
```

This now is an actor, as it implements the `RobotActor` interface - of course, `RandomGaitControl` is designed to do so. Yet, it might also be interesting to store sensor readouts of the simulation in a file for later inspection. For this purpose, there exists the `RobotCollector`. It is a transparent actor, meaning that it simulates an actor towards the `WebotsPuppyMixin` but lets a 'true' actor do the work. It just stores the provided measurements in a file.

This transparency can also be observed in the initialization, when the previously defined actor is passed as argument, together with a target file.

```
>>> observer = PuPy.RobotCollector(actor, expfile='/tmp/puppy_sim.hdf5')
```

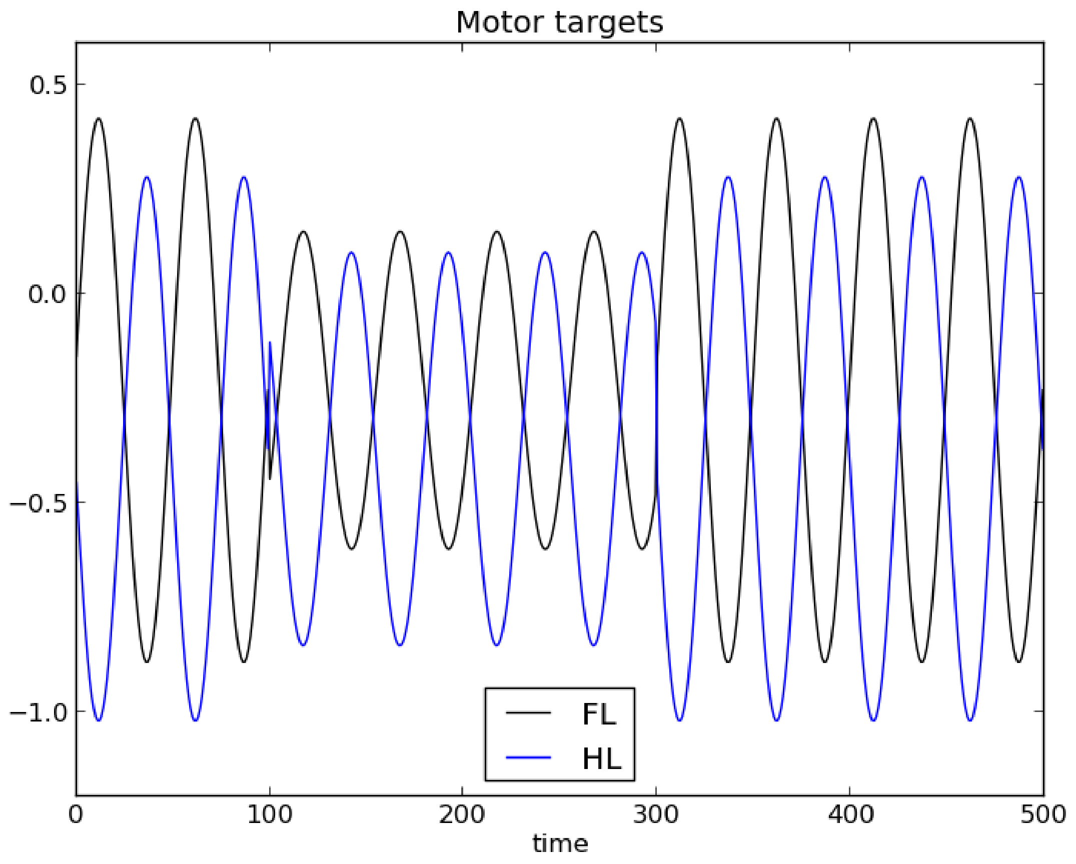
When the actor is initialized, it is passed to the constructor of `WebotsPuppyMixin` through the `robot builder`. Then, the robot controller can be executed and its effect observed in webots or from the data log.

```
>>> r = PuPy.robotBuilder(Robot, observer, sampling_period_ms=20)
>>> r.run()
```

Once some data has been gathered and the simulation is stopped, the data file may be inspected. Note that the code below is not part of the control script anymore but a seperate python instance (e.g. in a script or interactively). Note that this piece of code employs [matplotlib] for plotting.

```
>>> import h5py, pylab
>>> f = h5py.File('/tmp/puppy_sim.hdf5','r')
>>> pylab.plot(f['0']['trg0'][:500], 'k', label='FL')
>>> pylab.plot(f['0']['trg2'][:500], 'b', label='HL')
>>> pylab.title('Motor targets')
>>> pylab.xlabel('time')
>>> pylab.legend(loc=0)
>>> pylab.show()
```

In the appearing window, the target of the front and hind left motor is displayed. It can be observed that the shape of the target curves change between the two gaits defined. The figure below gives an example of this.



### 1.4.3 Reference

**class** `PuPy.`**`RobotActor`**

Template class for an actor, used in `WebotsPuppyMixin`.

The actor is called after every control period, when a new sequence of motor targets is required. It is expected to return an iterator which in every step produces a 4-tuple, representing the targets of the motors. The order is front-left, front-right, rear-left, rear-right.

**`epoch`** The sensor measurements in the previous control period. They are returned as dict, with the sensor name as key and a numpy array of observations as value.

Note that the motor targets are the ones that have been applied, i.e. those that lead up to the sensor measurements. Imagine this cycle:

```
trg[i] -> move robot -> sensors[i] -> trg[i+1] -> ...
```

Further, note that the `dict()` may be empty (this is guaranteed at least once in the simulator initialization).

**`time_start_ms`** The (simulated) time from which on the motor target will be applied. *time_start_ms* is weakly positive and strictly monotonic increasing (meaning that it is zero only in the very first call).

**`time_end_ms`** The (simulated) time up to which the motor target must at least be defined.

**`step_size_ms`** The motor period, i.e. the number of milliseconds pass until the next motor target is applied.

If the targets are represented by a list, it must at least have

(*time_end_ms* - *time_start_ms*) / *step_size_ms*

items and it has to be returned as iterator, as in

```
>>> iter(myList)
```

**class** `PuPy.`**`PuppyActor`**

Bases: `PuPy.control.RobotActor`

Deprecated alias for `RobotActor`.

**class** `PuPy.`**`Gait`**(*params*, *name=None*)

Motor target generator, using predefined gait_switcher.

The motor signal follows the parametrised sine

$$A \sin(2\pi f x + p) + B$$

with the parameters A, f, p, B

**`params`** `dict()` holding the parameters:

keys: amplitude, frequency, phase, offset

**values: 4-tuples holding the parameter values. The order is** front-left, front-right, rear-left, rear-right

**`iter`**(*time_start_ms*, *step*)

Return the motor target sequence in the interval [*time_start_ms*, *time_end_ms*].

**class** `PuPy.`**`RandomGaitControl`**(*gaits*)

Bases: `PuPy.control.PuppyActor`

From a list of available gaits, randomly select one.

**class** `PuPy.`**`ConstantGaitControl`**(*gait*)

Bases: `PuPy.control.PuppyActor`

Given a gait, always apply it.

**class** `PuPy.`**`SequentialGaitControl`**(*gait_iter*)

Bases: `PuPy.control.PuppyActor`

Execute a predefined sequence of gaits.

Note that it's assumed that *gait_iter* does not terminate permaturely.

**class** `PuPy.`**`RobotCollector`**(*actor*, *expfile*, *headers=None*)

Collect sensor readouts and store them in a file.

The data is stored in the [HDF5] format. For each simulation run, there's a group, identified by a running number. Within each group, the sensor data is stored in exclusive datasets, placed under the sensor's name.

---

**Note:** This class abstracts interface from implementation. Internally, either the HDF5 interface from [PyTables] or [h5py] may be used.

---

**`actor`** The `PuppyCollector` works as intermediate actor, it does not implement a policy itself. For this, another `actor` is required. It must match the `RobotActor` interface.

**`expfile`** Path to the file into which the experimental data should be stored.

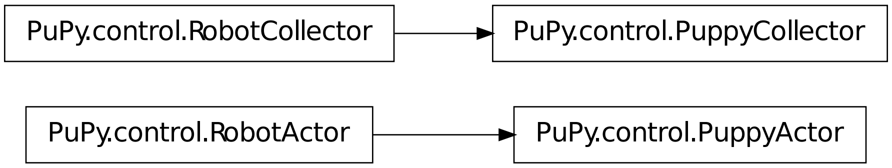
**headers** Additional headers, stored with the current experiment. A *dict* is expected. Default is None (no headers).

**class** `PuPy.`**`PuppyCollector`**(*actor*, *expfile*, *headers=None*)

Bases: `PuPy.control.RobotCollector`

Deprecated alias for `RobotCollector`.

### 1.4.4 Inheritance



## 1.5 Utils

### 1.5.1 Reference

`PuPy.`**`clear_empty_groups`**(*pth*)

Remove HDF5 groups with no datasets from `pth`. Return a list of removed groups.

**class** `PuPy.`**`Normalization`**(*pth=None*)

Bases: `object`

Supply normalization routines for sensor data. The normalization parameters (per sensor) are loaded from the `pth` file. See *py:meth:'load_normalization* for the file syntax.

**Todo**

Doesn't work with zero-mean and unit variance data. (There was a bug...).

The normalization takes two parameters, an *offset* ($o$) and *scale* ($s$). Data is normalized according to

$$x' = \frac{x - o}{s}$$

and denormalized

$$x = s * x' + o$$

The normalization parameters *offset* and *scale* must be set such that the desired mapping is achieved.

**denormalize_epoch** (*epoch*)
Denormalize all values in the `dict` `epoch`, where the key is regarded as sensor name.

**denormalize_value** (*sensor*, *value*)
Return the denormalized `value`, with respect to `sensor`.

**get** (*sensor*)
Return the normalization parameters of `sensor`. The order is (offset, scale). A valid result is returned in any case. If the sensor was not configured, the identity mapping is returned, i.e.

```
>>> nrm = Normalization()
>>> nrm.set('unknown_sensor', *nrm.get('unknown_sensor'))
>>> nrm.normalize_value('unknown_sensor', value) == value
True
```

However, a warning will be issued.

**load** (*pth*)
Load normalization parameters from a *JSON* file at `pth`.

The file structure must be like so:

```
{
    "<sensor name>" : [<offset>, <scale>],
}
```

For example:

```
{
    "hip0"   : [0.9678, 3.141],
    "touch0" : [-0.543, 1e3],
    "trg0"   : [1e-2, 8]
}
```

**Note:** This routine is only available, if the *json* module is installed.

**normalize_epoch** (*epoch*)
Normalize all values in the `dict` `epoch`, where the key is regarded as sensor name.

**normalize_value** (*sensor*, *value*)
Return the normalized `value`, with respect to `sensor`.

**params_stat** (*data*, *sensor=None*)
Find parameters *offset* and *scale* of `data`, such that normalized `data` has zero mean and unit variance.

If not `None`, the parameters will be attached to `sensor`.

The normalization parameters can be computed straight-forward: The *offset* is the mean, the *scale* the standard deviance.

**params_unit** (*data*, *sensor=None*)
Find parameters *offset* and *scale* of `data`, such that `data` is mapped to [-1.0, 1.0].

If not `None`, the parameters will be attached to `sensor`.

With $o = \min \text{data}$ and $s = \max \text{data} - \min \text{data}$, the normalization to [-1, 1] becomes

$$x' = 2\frac{x - o}{s} - 1$$

Since the normalization is is done differently (see `Normalization`), this has to be reformulated:

$$\begin{aligned} x' &= 2\frac{x-o}{s} - 1 \\ &= \frac{2}{s}(x-o) - \frac{2/s}{2/s} \\ &= \frac{2}{s}\left[x - o - \frac{1}{2/s}\right] \\ &= \frac{2}{s}\left[x - \left(o + \frac{s}{2}\right)\right] \end{aligned}$$

So, the effective normalization parameters are

$$\begin{aligned} \text{scale} &= \frac{s}{2} \\ \text{offset} &= o + \frac{s}{2} \end{aligned}$$

**save** (*pth*)

Store the normalization parameters in a *JSON* file at `pth`.

**set** (*sensor*, *offset*, *scale*)

Set the normalization parameters `offset` and `scale` for a specific `sensor`.

The normalization computes the following:

x' = ( x - `offset` ) / `scale`

## 1.6 Examples

### 1.6.1 A simple example

```
from controller import Supervisor
import PuPy

# checks
checks = []
checks.append(PuPy.RevertTumbled(grace_time_ms=2000))
checks.append(PuPy.RevertMaxIter(max_duration_ms=20*50*2*20))

# set up supervisor
s = PuPy.supervisorBuilder(Supervisor, 20, checks)

# run
s.run()
```

```
from controller import Robot
import PuPy
import os.path

# gait params
gait_params = {
    'bound_1Hz': {
            'frequency'  : (1.0, 1.0, 1.0, 1.0),
            'offset'     : ( -0.23, -0.23, -0.37, -0.37),
            'amplitude' : ( 0.56, 0.56, 0.65, 0.65),
            'phase'     : (0.0, 0.0, 0.5, 0.5)
        },
    'bound_left_1Hz': {
            'frequency' : (1.0, 1.0, 1.0, 1.0),
            'offset'    : ( -0.23, -0.23, -0.37,-0.37),
            'amplitude' : ( 0.38, 0.65, 0.47, 0.65),
            'phase'     : (0.1141, 0.0, 0.611155, 0.5)
        },
    'bound_right_1Hz': {
            'frequency' : (1.0, 1.0, 1.0, 1.0),
            'offset'    : ( -0.23, -0.23, -0.37, -0.37),
            'amplitude' : ( 0.65, 0.38, 0.65, 0.47),
            'phase'     : (0.0, 0.1141, 0.5, 0.611155)
        }
}

# gaits
gaits = [PuPy.Gait(gait_params[g], name=g) for g in gait_params]

# actor
actor = PuPy.RandomGaitControl(gaits)
#actor = PuPy.ConstantGaitControl(gaits[2])
observer = PuPy.RobotCollector(actor, expfile='/tmp/puppy_sim.hdf5')

# robot
r = PuPy.robotBuilder(Robot, observer, sampling_period_ms=20, noise_ctrl=None, noise_obs=None)

# run
r.run()
```

Try out the code by storing the excerpts on your disk and execute:

```
world_builder -c <pth/to/robot> -s <pth/to/supervisor> -t styrofoam /tmp/webots_test
```

### 1.6.2 A custom collector

It's basically the same example as above. The main difference is that the observation file contains a new dataset 'random' (with just random values). It has the same length as the other datasets but ten columns (instead of the usual single one).

```
from controller import Robot
import PuPy
import os.path
import numpy as np

# gait params
gait_params = {
    'frequency' : (1.0, 1.0, 1.0, 1.0),
    'offset'    : ( -0.23, -0.23, -0.37, -0.37),
    'amplitude' : ( 0.56, 0.56, 0.65, 0.65),
    'phase'     : (0.0, 0.0, 0.5, 0.5)
}

# gaits
gait = PuPy.Gait(gait_params)
```

```
# Multidimensional collector
class MyCollector(PuPy.RobotCollector):
    def __call__(self, epoch, time_start_ms, time_end_ms, step_size):
        if len(epoch) > 0:
            epoch['random'] = np.random.normal(size=(100,10))
        return super(MyCollector, self).__call__(epoch, time_start_ms, time_end_ms, step_size)


# actor
actor = PuPy.ConstantGaitControl(gait)
observer = MyCollector(actor, expfile='/tmp/puppy_sim.hdf5')


# robot
r = PuPy.robotBuilder(Robot, observer, sampling_period_ms=20, noise_ctrl=None, noise_obs=None)

# run
r.run()
```

### 1.6.3 Supervisor to Robot communication

The robot will walk unspectacularly but a message will be printed to the console every 100ms. The message is ‘Emitting `<nr>`‘, with `<nr>` a multiple of 100.

```
from controller import Supervisor
import PuPy

# emitter checks
class EmitterCheck(PuPy.SupervisorCheck):
    def __call__(self, supervisor):
        supervisor.emitter.send('Emitting ' + str(supervisor.num_iter))

checks = []
checks.append(EmitterCheck())

# set up supervisor
PuPy.supervisorBuilder(Supervisor, 100, checks).run()
```

```
from controller import Robot
import PuPy
import os.path
import numpy as np

# gait params
gait_params = {
    'frequency' : (1.0, 1.0, 1.0, 1.0),
    'offset'    : ( -0.23, -0.23, -0.37, -0.37),
    'amplitude' : ( 0.56, 0.56, 0.65, 0.65),
    'phase'     : (0.0, 0.0, 0.5, 0.5)
}

# gaits
gait = PuPy.Gait(gait_params)

# receiver callback
def recv1(robot, epoch, current_time, msg):
    print "recv1", msg
```

```
def recv2(robot, epoch, current_time, msg):
    print "recv2", msg

# actor
actor = PuPy.ConstantGaitControl(gait)


# robot
r = PuPy.robotBuilder(Robot, actor, event_period_ms=50, event_handlers=[recv1, recv2])
r.run()
```

## 1.7 Download

**Contents**

### 1.7.1 Installation

Using Pip Installs Python (Pip), simply type:

```
pip install http://www.igsor.net/research/PuPy/_downloads/latest.tar.gz
```

if you want to use the package from the webpage. If you have downloaded it yourself, use:

```
pip install path/to/PuPy.tar.gz
```

If you're using distutils, type:

```
tar -xzf path/to/PuPy.tgz       # extract files.
cd PuPy*                        # change into PuPy directory.
sudo python setup.py install    # install using distutils (as root).
#rm -R .                        # remove source. If desired, uncomment this line.
#cd .. && rmdir PuPy*           # remove working directory. If desired, uncomment this line but be ca
```

The project is also available on git, with the package and all supplementary data:

> git clone https://github.com/igsor/PuPy

Make sure all dependencies are installed on your system. This obviously includes [Python], as well as [SciPy] and [NumPy]. Furthermore, either [h5py] or [PyTables] is required. Since this module is tightly bound to [Webots] you should have it available as well.

If you wish to store the library outside of the system's paths, a neat way to link the library to Python is to use a path configuration file (`.pth`). Place a file `<modname>.pth` in python's default module search path for site packages. The file must contain nothing else than the actual path to the library. Note that the module's source must be within a subfolder of the specified path and the subfolder must have the same name as the module (i.e. `<modname>`). Normally, `<modname>` would be *PuPy*.

To get the site package path, type:

```
>>> import site
>>> site.getsitepackages()
['/usr/local/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages']
```

If in use, it also makes sense to install the additional scripts from the *webots page* in a system's default executable search path. Get these by typing into a bash:

```
tux@localhost:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

`/usr/local/sbin` should be a fair choice. It is advised to keep the script together with the other data, thus only a symlink is created (as root):

```
root@localhost:~# ln -s </pth/to/world_builder.sh> /usr/local/sbin/webots_builder
```

The same procedure may be repeated for `world_splitter.sh`.

### 1.7.2 Available downloads

PuPy library and documentation by Nico Schmidt and Matthias Baumgartner:

- `PuPy-1.0` (latest)
- `This documentation (pdf)`

Supplementary scripts and the Puppy webots model by Matej Hoffmann and Stefan Hutter:

- `Webots data and supplementary scripts`

The logo by Karin Baumgartner:

- `PuPy Logo (svg)`
- `PuPy Logo (pdf)`

### 1.7.3 License

This project is released under the terms of the 3-clause BSD License and CC BY. See the section *License* for details.

## 1.8 License

This project is released under the terms of the 3-clause BSD License.

## 1.9 Resources

## 1.10 Hic sunt dracones

**Todo**

Doesn't work with zero-mean and unit variance data. (There was a bug...).

(The *original entry* is located in /home/matthias/studium/master/libs/PuPy/PuPy/inputoutput.py:docstring of PuPy.Normalization, line 5.)

CHAPTER

TWO

# INDICES AND TABLES

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

# BIBLIOGRAPHY

[Webots] http://www.cyberbotics.com/

[HDF5] http://www.hdfgroup.org/HDF5/

[PyTables] http://www.pytables.org/

[h5py] http://www.h5py.org/

[SciPy] http://www.scipy.org/

[NumPy] http://numpy.scipy.org/

[Python] http://www.python.org

[matplotlib] http://matplotlib.org/

# PYTHON MODULE INDEX

# INDEX