PyIBL Documentation

Release 3.0.dev1

CMU DDMLab

August 05, 2014

CHAPTER

ONE

WHAT IS PYIBL?

PyIBL is a Python implementation of a subset of Instance Based Learning Theory (IBLT) [1]. It is made and distributed by the Dynamic Decision Making Laboratory of Carnegie Mellon University for making computational cognitive models supporting research in how people make decisions in dynamic environments.

Typically PyIBL is used by creating an experimental framework in the Python programming language, which uses one or more PyIBL `Agent` objects. The framework then asks these agents to make decisions, and informs the agents of the results of those decisions. The framework, for example, may be strictly algorithmic, may interact with human subjects, or may be embedded in a web site.

PyIBL is a library, or module, of Python code, useful for creating Python programs; it is not a stand alone application. Some knowledge of Python programming is essential for using it.

PyIBL is an ongoing project, and has been started with a small subset of IBLT. As it evolves it is expected that more and more of the IBLT will be encorporated into it. For example, PyIBL does not currently support similarity, spreading activation or deferred feedback, but those are planned for the near future. PyIBL is still sufficiently early in its development that future versions are likely to radically change some of the APIs exposed today, and some effort may be required to upgrade projects using the current version of PyIBL to a later version.

[1] Cleotilde Gonzalez, Javier F. Lerch and Christian Lebiere (2003), Instance-based learning in dynamic decision making, *Cognitive Science*, *27*, 591-635. DOI: 10.1016/S0364-0213(03)00031-4.

CHAPTER

TWO

INSTALLING PYIBL

PyIBL requires Python version 3.2 or later. If you do not have other constraints on what version(s) of Python you use, version 3.4 or later is recommended. Recent versions of Mac OS X and recent Linux distributions likely have a suitable version of Python installed, but it will probably need to be invoked as `python3` instead of just `python`, which typically runs a 2.x version. Python, for Windows, Mac OS X, Linux, or other Unices, can be downloaded from python.org, for free. On Windows python is often invoked as `py` rather than `python`.

Note that PyIBL is simply a Python module, a library, that is run as part of a larger Python program. To build and run models using PyIBL you do need to do some Python programming. If you're new to Python, a good place to start learning it is The Python Tutorial. To write and run a Python program you need to create and edit Python source files, and then run them. If you are comfortable using the command line, you can simply create and edit the files in your favorite text editor, and run them from the command line. Many folks, though, are happier using a graphical Integrated Development Environment (IDE). Many Python IDEs are available. One is IDLE, which comes packaged with Python itself, so once you've installed Python you should have it available.

PyIBL can be downloaded, for research use, from http://downloads.ddmlab.com/pyibl. You will be asked to register with the DDMLab before downloading PyIBL. Then download the PyIBL source distribution, pyibl-3.0.dev1.zip. Also available for download from the same page is a PDF version of this documentation, pyibl-3.0.dev1-documentation.pdf. After you have downloaded the source distribution, installing PyIBL should be as simple as running, in the directory into which you have downloaded it,

```
pip install pyibl-3.0.dev1.zip
```

Unfortunately there is great variability in what version, or versions, of Python are installed on any given machine; what version, or versions, of distribution software they use and/or have been installed; and a plethora of other complications. Depending upon your situation you may have to

- precede `pip` by `sudo`,
- use the `-t` option to `pip install`,
- manually install the required packages `ordered_set`, `verlib` and `prettytable` from PyPI,
- unpack by hand, and then run by hand the installation script, as described below,
- use whatever scheme your Python IDE supports,
- or some combination of these.

Whatever recipe you normally use to install other Python packages on your machine should work for PyIBL.

If you do not have `pip` available or do not want to use it you can simply unpack and install the source distribution by hand. After unpacking it with an appropriate tool (e.g. `unzip`) run, in the directory that has been created by unpacking the file:

```
python3 setup.py install
```

CHAPTER

THREE

TUTORIAL

Likely the easiest way to get started with PyIBL is by looking at some examples of its use. While much of what is in this chapter should be understandable even without much knowledge of Python, to write your own models you'll need to know how to write Python code. If you are new to Python, a good place to start may be The Python Tutorial.

3.1 A first example of using PyIBL

In the code blocks that follow, lines the user has typed begin with any of the three prompts,

```
$
>>>
...
```

Other lines are printed by Python or some other command.

First we launch Python, and make PyIBL available to it. While the output here was captured in Linux distribution in which you launch Python version 3 by typing `python3`, you installation my differ and you may launch it with `python`, `py`, or something else entirely; or start an interactive session in a completely different way using a graphical IDE.

```
$ python3
Python 3.4.1 (default, May 26 2014, 01:12:52)
[GCC 4.8.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyibl
>>>
```

Next we create an `Agent`, named `'My Agent'`.

```
>>> a = pyibl.Agent('My Agent')
>>> a
<Agent My Agent (0, 0, 0)>
>>>
```

The IBLT depends upon several parameters, which are typically adjusted by the modeler for an agent. We'll set the `Agent.noise` to 1.5 and the `Agent.decay` to 5.

```
>>> a.noise
0.0
>>> a.noise = 1.5
>>> a.noise
1.5
>>> a.decay = 5
>>>
```

We also have to tell the agent what do if we ask it to choose between options it has never previously experienced. One way to do this is to set a default by setting the agent's `Agent.defaultUtility` property.

```
>>> a.defaultUtility = 10.0
>>>
```

Now we can ask the agent to choose between two options, that we'll just describe using two strings. When you try this yourself you may get the opposite answer as the IBLT theory deliberately includes some randomness, which is particularly obvious in cases like this where there is no reason yet to prefer one answer to the other.

```
>>> a.choose('The Green Button', 'The Red Button')
'The Green Button'
>>>
```

Now return a response to the model. We'll supply 1.0.

```
>>> a.respond(1.0)
>>>
```

Because that value is significantly less than the default utility when we ask the agent to make the same choice again, we expect it with high probability to pick the other button.

```
>>> a.choose('The Green Button', 'The Red Button')
'The Red Button'
```

We'll give it an even lower utility than we did the first one.

```
>>> a.respond(-2.0)
>>>
```

If we stick with these responses the model will tend to favor the first button selected. Again, your results may differ in detail because of randomness.

```
>>> a.choose('The Green Button', 'The Red Button')
'The Green Button'
>>> a.respond(1.0)
>>> a.choose('The Green Button', 'The Red Button')
'The Red Button'
>>> a.respond(-2.0)
>>> a.choose('The Green Button', 'The Red Button')
'The Green Button'
>>> a.respond(1.0)
>>> a.choose('The Green Button', 'The Red Button')
'The Green Button'
>>> a.respond(1.0)
>>> a.choose('The Green Button', 'The Red Button')
'The Green Button'
>>> a.respond(1.0)
>>> a.choose('The Green Button', 'The Red Button')
'The Green Button'
>>> a.respond(1.0)
```

But doing this by hand isn't very useful for modeling. Instead, let's write a function that asks the model to make this choice, and automates the reply.

```
>>> def chooseAndRespond():
...     result = a.choose('The Green Button', 'The Red Button')
...     if result == 'The Green Button':
...         a.respond(1.0)
...     else:
...         a.respond(-2.0)
```

```
...     return result
...
>>> chooseAndRespond()
'The Green Button'
>>>
```

Let's ask the model to make this choice a thousand times, and see how many times it picks each button. But let's do this from a clean slate. So, before we run it, we'll cause `Agent.reset()` to clear the agent's memory.

```
>>> a.reset()
>>> results = { 'The Green Button' : 0, 'The Red Button' : 0 }
>>> for i in range(1000):
...     results[chooseAndRespond()] += 1
...
>>> results
{'The Red Button': 22, 'The Green Button': 978}
>>>
```

As we expected the model prefers the green button, but because of randomness, does try the red one occasionally.

Now lets add some other choices. We'll make a more complicated function that takes a dictionary of choices and the responses they generate, and see how they do. This will make use of a bit more Python, and, if it doesn't make sense to you, investigate Python dictionaries, that is objects of the `dict` class, and their `dict.keys()` method. The default utility is still 10, and so long as the responses are well below that we can reasonably expect the first few trials to sample them all before favoring those that give the best results; but after the model gains more experience, it will favor whatever color or colors give the highest rewards.

```
>>> def chooseAndRespond(choices):
...     result = a.choose(*choices.keys())
...     a.respond(choices[result])
...     return result
...
>>> a.reset()
>>> choices = { 'green': -5, 'blue': 0, 'yellow': -4,
...             'red': -6, 'violet': 0 }
>>> results = {}
>>> for color in choices.keys():
...     results[color] = 0
...
>>> for i in range(5):
...     results[chooseAndRespond(choices)] += 1
...
>>> results
{'green': 1, 'blue': 1, 'yellow': 1, 'red': 1, 'violet': 1}
>>> for i in range(955):
...     results[chooseAndRespond(choices)] += 1
...
>>> results
{'green': 18, 'blue': 441, 'yellow': 19, 'red': 19, 'violet': 463}
```

The results are as we expected.

3.2 Logging

PyIBL can write out detailed logs of what it's doing, including both details of what it's being asked to do and how it is responding, and details of its internal computations.

For demonstrating this we'll define a model that chooses between two options, `'risky'` and `'safe'`. When it chooses `'safe'` the reward is always 1.0, but when it chooses `'risky'` 30% of the time it will receive 4.0, but the rest of the time nothing.

```
>>> from pyibl import Agent
>>> from random import random
>>> a = Agent('risky/safe')
>>> a.noise=1.5
>>> a.decay=5.0
>>> a.defaultUtility=10.0
>>> def makeChoice():
...     result = a.choose('risky', 'safe')
...     if result == 'safe':
...         a.respond(1.0)
...     elif random() <= 0.3:
...         a.respond(4.0)
...     else:
...         a.respond(0.0)
...     return result
...
>>>
```

By default a log file is written in comma separated values (CSV) format, though that can easily be altered. While normally we would write the log to a file by calling `Population.logToFile()`, for this example we'll just let it write to standard output, which is the default if we don't specify a file.

To enable logging we set the agent's `Agent.logging` property to a set of strings describing which fields we'd like to see in the log. The possible such fields, and the strings to request them, are *described in the Reference section* of this manual. For starters in this example we'll just ask to see the trial number, the choice made, and the response received.

```
>>> a.logging = {'tTrial', 'tChoice', 'tResponse'}
>>> for i in range(10):
...     result = makeChoice()
...
tTrial,tChoice,tResponse
1,risky,0.0000
2,safe,1.0000
3,safe,1.0000
4,risky,0.0000
5,safe,1.0000
6,risky,0.0000
7,safe,1.0000
8,risky,4.0000
9,risky,0.0000
10,safe,1.0000
>>>
```

It first printed three column headings that look like the strings used to request them, and then ten lines, once for each choose/respond cycle.

Now we'll add a couple further fields to the log, the possible choices the agent is choosing between at each trial, and the blended values produced; at each trial it will be picking the choice with the larger blended value. We could just write out the whole set of attributes we want, but instead we'll use Python's set union operator to just add them to the existing set.

```
>>> a.logging |= {'oDecision', 'oBlendedValue'}
>>> a.reset()
>>> for i in range(10):
...     result = makeChoice()
```

```
...
tTrial,tChoice,tResponse,oDecision,oBlendedValue
1,safe,1.0000,risky,10.0000
1,safe,1.0000,safe,10.0000
2,risky,0.0000,risky,10.0000
2,risky,0.0000,safe,2.8242
3,safe,1.0000,risky,0.5979
3,safe,1.0000,safe,1.0943
4,safe,1.0000,risky,0.0492
4,safe,1.0000,safe,1.1291
5,safe,1.0000,risky,0.1274
5,safe,1.0000,safe,2.8958
6,risky,4.0000,risky,2.9752
6,risky,4.0000,safe,1.0126
7,risky,0.0000,risky,4.1094
7,risky,0.0000,safe,1.1840
8,risky,4.0000,risky,3.1863
8,risky,4.0000,safe,1.4147
9,safe,1.0000,risky,2.8826
9,safe,1.0000,safe,3.0415
10,risky,4.0000,risky,1.9414
10,risky,4.0000,safe,1.0655
>>>
```

Now we have two lines printed for every trial, one for each of the alternatives (oDecision) presented to the model to pick between. On both lines the choice (tChoice) that was actually made is the same, as is the actual payoff received (tResponse). The blended value is that, at this trial, for each of the two alternatives. Note that until a real result has been experienced the blended values are the default utility, but they then start to reflect the models actual experiences.

We'll add a few further fields to the log, these describing the various instances present in the model at each trial. The fields we'll add are the utility of each instance (iUtility), when it has been experienced (iOccurrences), details of its activation (iActivationBase, iActivationNoise, iActivation) at this trial, and its retrieval probability (iRetrievalProbability) at this trail. Note that the total activation (iActivation) is just the sum of the computed base activation and the noise.

```
>>> a.logging |= {'iUtility', 'iOccurrences', 'iActivation', 'iRetrievalProbability'}
>>> a.reset()
>>> for i in range(10):
...     result = makeChoice()
...
tTrial,tChoice,tResponse,oDecision,oBlendedValue,iUtility,iOccurrences,iActivation,iRetrievalProbabil
1,safe,1.0000,risky,10.0000,10.0000,0,0.5182,1.0000
1,safe,1.0000,safe,10.0000,10.0000,0,-3.2292,1.0000
2,risky,4.0000,risky,10.0000,10.0000,0,-4.0032,1.0000
2,risky,4.0000,safe,3.3630,10.0000,0,-1.1146,0.2626
2,risky,4.0000,safe,3.3630,1.0000,1,1.0762,0.7374
3,risky,0.0000,risky,4.6605,10.0000,0,-4.5835,0.1101
3,risky,0.0000,risky,4.6605,4.0000,2,-0.1500,0.8899
3,risky,0.0000,safe,4.0279,10.0000,0,-5.6409,0.3364
3,risky,0.0000,safe,4.0279,1.0000,1,-4.2000,0.6636
4,safe,1.0000,risky,1.5001,10.0000,0,-7.2386,0.0319
4,safe,1.0000,risky,1.5001,4.0000,2,-2.5140,0.2954
4,safe,1.0000,risky,1.5001,0.0000,3,-0.7680,0.6728
4,safe,1.0000,safe,7.7438,10.0000,0,-3.0752,0.7493
4,safe,1.0000,safe,7.7438,1.0000,1,-5.3980,0.2507
5,risky,0.0000,risky,2.4222,10.0000,0,-9.1597,0.1167
5,risky,0.0000,risky,2.4222,4.0000,2,-7.0615,0.3138
5,risky,0.0000,risky,2.4222,0.0000,3,-5.7971,0.5695
5,risky,0.0000,safe,1.0420,10.0000,0,-9.1142,0.0047
```

```
5,risky,0.0000,safe,1.0420,1.0000,"1,4",2.2639,0.9953
6,safe,1.0000,risky,1.1608,10.0000,0,-11.1392,0.0082
6,safe,1.0000,risky,1.1608,4.0000,2,-3.7375,0.2696
6,safe,1.0000,risky,1.1608,0.0000,"3,5",-1.6476,0.7221
6,safe,1.0000,safe,9.0471,10.0000,0,-6.9011,0.8941
6,safe,1.0000,safe,9.0471,1.0000,"1,4",-11.4270,0.1059
7,risky,4.0000,risky,3.1665,10.0000,0,-10.1344,0.0823
7,risky,4.0000,risky,3.1665,4.0000,2,-5.9717,0.5858
7,risky,4.0000,risky,3.1665,0.0000,"3,5",-7.1772,0.3319
7,risky,4.0000,safe,1.6158,10.0000,0,-6.3350,0.0684
7,risky,4.0000,safe,1.6158,1.0000,"1,4,6",-0.7958,0.9316
8,risky,0.0000,risky,3.8982,10.0000,0,-10.0171,0.0052
8,risky,0.0000,risky,3.8982,4.0000,"2,7",1.0505,0.9615
8,risky,0.0000,risky,3.8982,0.0000,"3,5",-6.0847,0.0333
8,risky,0.0000,safe,1.0187,10.0000,0,-14.6961,0.0021
8,risky,0.0000,safe,1.0187,1.0000,"1,4,6",-1.5994,0.9979
9,safe,1.0000,risky,0.5787,10.0000,0,-9.7556,0.0060
9,safe,1.0000,risky,0.5787,4.0000,"2,7",-3.2219,0.1298
9,safe,1.0000,risky,0.5787,0.0000,"3,5,8",0.8006,0.8643
9,safe,1.0000,safe,1.2674,10.0000,0,-10.6717,0.0297
9,safe,1.0000,safe,1.2674,1.0000,"1,4,6",-3.2763,0.9703
10,safe,1.0000,risky,0.8400,10.0000,0,-11.4662,0.0153
10,safe,1.0000,risky,0.8400,4.0000,"2,7",-6.3433,0.1716
10,safe,1.0000,risky,0.8400,0.0000,"3,5,8",-3.0439,0.8130
10,safe,1.0000,safe,1.1395,10.0000,0,-10.9373,0.0155
10,safe,1.0000,safe,1.1395,1.0000,"1,4,6,9",-2.1313,0.9845
>>>
```

Note that the number of lines at each trial increases as further instances are added to memory. Each instance corresponds to a unique combination of the possible decision (oDecision) and a utility (iUtility).

Often a modeler will want to run experiments together that are similar, but differ from one another in some way. It is often useful to be able to combine the results off all into a single log, with some field distinguishing them. PyIBL provides the `Population.block` property to address this need. The modeler can set this property to any desired value, and, by asking for an 'tBlock' column in the log, see what value it had for each of the log entries.

For example, imagine we wanted to run the above experiment for four different participants, each with its own set of experiences and memories, for five trials apiece. To do this we simply run the overall five trial experiment in an outer loop, once through for each participant, calling `Agent.reset()` at the beginning of each of the out loops to clear the agent's memory. To see which participant is which in the log we also set the block property at the beginning of each outer loop, and include a tBlock column in the log.

```
>>> a.logging = { 'tBlock', 'tTrial', 'tChoice', 'tResponse' }
>>> for participant in range(1,5):
...     a.reset()
...     a.block = "participant-" + str(participant)
...     for trial in range(5):
...         result = makeChoice()
...
tBlock,tTrial,tChoice,tResponse
participant-1,1,risky,4.0000
participant-1,2,safe,1.0000
participant-1,3,risky,0.0000
participant-1,4,safe,1.0000
participant-1,5,safe,1.0000
participant-2,1,safe,1.0000
participant-2,2,risky,4.0000
participant-2,3,risky,0.0000
participant-2,4,safe,1.0000
```

```
participant-2,5,risky,0.0000
participant-3,1,safe,1.0000
participant-3,2,risky,0.0000
participant-3,3,safe,1.0000
participant-3,4,risky,4.0000
participant-3,5,risky,0.0000
participant-4,1,risky,0.0000
participant-4,2,safe,1.0000
participant-4,3,safe,1.0000
participant-4,4,risky,0.0000
participant-4,5,safe,1.0000
>>>
```

3.3 Multiple agents and populations

A PyIBL model is not limited to using just one agent. It can use as many as the modeler wishes. For this example we'll have ten players competing for rewards. Each player, at each turn, will pick either `'safe'` or `'risky'`. Any player picking `'safe'` will always receive 1 point. All those players picking `'risky'` will share 7 points evenly between them; if fewer than seven players pick `'risky'` those that did will receive more than if they had picked `'safe'`, but if more than seven players pick `'risky'` they will do worse.

```
>>> from pyibl import Agent
>>> agents = [Agent('Agent-' + str(i+1)) for i in range(10)]
>>> for a in agents:
...     a.noise = 2.0
...     a.decay = 4.5
...     a.defaultUtility = 20
>>>
>>> def playRound():
...     choices = [a.choose('safe', 'risky') for a in agents]
...     risky = [a for a, c in zip(agents, choices) if c == 'risky']
...     reward = 7 / len(risky)
...     for a in agents:
...         if a in risky:
...             a.respond(reward)
...         else:
...             a.respond(1)
...
>>>
```

We could enable logging for each of these ten agents, and write ten separate log files. But with multiple agents it is usually more convenient to have all the agents write to the same log file, and include an agent's name in each line that it writes. To make this easy we collect all our agents into a PyIBL `Population`, by setting each agent's `Agent.population` property, and configure the population's log. Because this model will emit a lot of log information we write the log to a file.

```
>>> from pyibl import Population
>>> p = Population()
>>> for a in agents:
...     a.population = p
...
>>> for i in range(100):
...     playRound()
...
>>> p.logging = {'tAgent', 'tTrial', 'tChoice', 'tResponse', 'oDecision', 'oBlendedValue'}
>>> p.logToFile('logfile.csv')
```

```
'logfile.csv'
>>> for i in range(100):
...     playRound()
...
>>>
```

The log file is about two thousand lines long, so let's just look at the first twenty and last twenty lines of it.

```
$ head -n 20 logfile.csv
tAgent,tTrial,tChoice,tResponse,oDecision,oBlendedValue
Agent-1,1,safe,1,safe,20.0000
Agent-1,1,safe,1,risky,20.0000
Agent-2,1,safe,1,safe,20.0000
Agent-2,1,safe,1,risky,20.0000
Agent-3,1,safe,1,safe,20.0000
Agent-3,1,safe,1,risky,20.0000
Agent-4,1,safe,1,safe,20.0000
Agent-4,1,safe,1,risky,20.0000
Agent-5,1,risky,3.5000,safe,20.0000
Agent-5,1,risky,3.5000,risky,20.0000
Agent-6,1,safe,1,safe,20.0000
Agent-6,1,safe,1,risky,20.0000
Agent-7,1,safe,1,safe,20.0000
Agent-7,1,safe,1,risky,20.0000
Agent-8,1,safe,1,safe,20.0000
Agent-8,1,safe,1,risky,20.0000
Agent-9,1,risky,3.5000,safe,20.0000
Agent-9,1,risky,3.5000,risky,20.0000
Agent-10,1,safe,1,safe,20.0000
$ tail -n 20 logfile.csv
Agent-1,100,risky,0.8750,safe,1.1613
Agent-1,100,risky,0.8750,risky,1.3327
Agent-2,100,safe,1,safe,2.9859
Agent-2,100,safe,1,risky,1.4528
Agent-3,100,risky,0.8750,safe,1.1551
Agent-3,100,risky,0.8750,risky,1.5729
Agent-4,100,risky,0.8750,safe,1.0251
Agent-4,100,risky,0.8750,risky,1.0777
Agent-5,100,risky,0.8750,safe,1.0058
Agent-5,100,risky,0.8750,risky,1.0116
Agent-6,100,risky,0.8750,safe,1.0234
Agent-6,100,risky,0.8750,risky,1.3029
Agent-7,100,risky,0.8750,safe,1.0082
Agent-7,100,risky,0.8750,risky,1.0312
Agent-8,100,risky,0.8750,safe,1.0963
Agent-8,100,risky,0.8750,risky,1.4120
Agent-9,100,risky,0.8750,safe,1.0147
Agent-9,100,risky,0.8750,risky,1.1045
Agent-10,100,safe,1,safe,1.0527
Agent-10,100,safe,1,risky,1.0045
$
```

We see that there are two lines, one corresponding to each of the potential decisions, for each agent at each trial. At the beginning the blended values are dominated by the default utility, but at the end dominated by experience, which for most of the agents has shown that the risky choice is usually a little bit better, though at this particular trial those making the risky bet lost. That the blended value for the safe bet is so high for Agent-2, and a little high than you might expect for Agent-10, is the result of noise in the activation computation, which on this trial has paid off for these two agents!

3.4 Situations

Often we want the choices an agent makes to be affected by a possibly varying situation that can affect which decision is better at this time. For a PyIBL agent a situation is simply an ordered collection of values, called "attributes". When we create an agent besides supplying its name, we can supply the names of one or more attributes that describe the situations relevant to its choices. Then when asking the agent to make a choice instead of passing just the decisions as arguments to `Agent.choose()`, we pass `SituationDecision` objects to the agent, each combining a potential decision with its current situation.

In fact, whenever we call an agent's choose method we are always passing it SituationDecisions. The choose method is simply clever and when it sees a choice that is not a SituationDecision it creates one with the provided value as the decision, and all the attributes of the situation set to `None`. While we've been using strings as decisions, any Python object that is hashable, except `None`, can be a decision. Attribute values in situations can be any Python object that is hashable.

As a concrete example, we'll have our agent decide which of two buttons, `'left'` or `'right'`, to push. But one of these buttons will be illuminated. Which is illuminated at any time is decided randomly, with even chances for either. Pushing the left button earns a base reward of 1, and the right button of 2; but when a button is illuminated its reward is doubled.

We'll define our agent to have situations with one attribute, `'illuminated'`, which says whether or not the button which a SituationDecision represents is illuminated.

```
>>> from pyibl import Agent
>>> from random import random
>>> a = Agent('My Agent', 'illuminated')
>>> a.noise = 0.5
>>> a.decay = 3.0
>>> a.defaultUtility = 5
>>>
```

We'll create two SituationDecisions, one for each button. Because a SituationDecision should have exactly the attributes an agent is expecting, all SituationDecisions are actually associated with an agent, and are created by calling that agent's `Agent.situationDecision()` method.

```
>>> sds = {'left': a.situationDecision('left', False),
...        'right': a.situationDecision('right', False)}
>>>
```

While we've created them both with the button un-illuminated, the code that actually runs the experiment will turn one of them on, randomly. Note that while we pass SituationDecisons to the choose method, choose only returns the decision it has selected, not the whole SituationDecison.

```
>>> def punchButton():
...     if random() <= 0.5:
...         sds['left'].set('illuminated', True)
...         sds['right'].set('illuminated', False)
...     else:
...         sds['left'].set('illuminated', False)
...         sds['right'].set('illuminated', True)
...     result = a.choose(*sds.values())
...     reward = 1 if result == 'left' else 2
...     if sds[result].get('illuminated'):
...         reward *= 2
...     a.respond(reward)
...     return result
...
>>>
```

Now we'll run it 2,000 times, counting how many times each button is picked, and how many times an illuminated button is picked.

```
>>> results = {'left': 0, 'right': 0, True: 0, False: 0}
>>> for i in range(2000):
...     result = punchButton()
...     results[result] += 1
...     results[sds[result].get('illuminated')] += 1
...
>>> print(results)
{False: 486, True: 1514, 'left': 505, 'right': 1495}
>>>
```

CHAPTER

FOUR

REFERENCE

Here are described details of the PyIBL API.

4.1 Agents

class pyibl.**Agent**(*name=None*, **attributes*)

A cognitive entity learning and making decisions based on its experience from prior decisions. The main entry point to PyIBL. An Agent has a *name*, a string, which can be retrieved with the `name` property. The name cannot be changed after an agent is created. If, when creating an agent, the *name* argument is not supplied or is `None`, a name will be created of the form `'Anonymous-Agent-n'`, where *n* is a unique integer. An agent also has zero or more *attributes*, named by strings. The attribute names can be retrieved with the `attributes` property, and also cannot be changed after an agent is created. Attempting to use a non-string (other than `None` for the *name*) for an agent or attribute name raises an `IllegalArgumentError`.

The most important methods on agents are `choose()` and `respond()`. The choose method is typically called with two or more arguments, each a `SituationDecision`. The agent chooses, based on its prior experiences, which of those SitutationDecisions will result in the greatest reward, and returns the corresponding `decision` from it. The respond method informs an agent of the result of its most recent choice; between any two requests that an agent make a choice the outcome of the earlier choice must be delivered to the agent by calling respond, or an `IllegalStateError` is raised. An outcome is a real (that is, not complex) number, where larger is implicitly "better" in some way. Before a PyIBL model learns from outcomes delivered to it by respond, the learning in the model must be bootstrapped in some way. There are two mechanisms for doing so: the `prepopulate()` method and the `defaultUtility` property. Parameters controlling the learning model can be adjusted by setting the values of the `noise`, `decay` and `temperature` properties. If it is desired to use an agent multiple times it may be `reset()`, erasing all memory of past interactions, though preserving its `noise`, `decay` and `temperature` parameters, as well as its logging settings.

Details of an agent's actions, and the computations underlying them, may be captured in a log. This is controlled by setting properties and calling methods on the agent's `Population`, which can be retrieved at the value of the agent's `population` property. Alternatively most such logging methods and properties are also provided by Agent, which simply delegate to the agent's population. Like populations, agents can be used with Pythons `with` method to ensure the agent's population's log is close when exiting a block of code.

attributes

A tuple of the names of the attributes included in all situations associated with decisions this agent will be asked to make. These names are assigned when the agent is created and cannot be changed, and are strings. The order of them in the returned tuple is the same as that in which they were given when the agent was created, in which values are assigned when creating SitutationDecisions, and reported by the `situation` property of any SitutationDecisions created by this agent.

block

The `Population.block` property of this agent's `population`. May be assigned to, in which case it

sets the agent's population's block property. Note that so assigning to this affects the logging behavior of any other agents that belong to the same population.

choose(**situationDecisions*)

Selects which of the *selectionDecisions* is expected to result in the largest payoff, and returns its decision. Each of the *situationDecisions* must have a different `SituationDecision.decision`. The situations of the *situuationDecisions* must have the same attributes as this agent, and will typically have been created by its `siutationDecision()` method. If any deciions are duplicated, or if any of the *situationDecisions*'s attributes do not match those of this agent, an `IllegalArgumentError` is raised.

It is also possible to supply no *situationDecisions*, in which case those used in the most recent previous call to this method are reused. If there was no previous call to choose, an `IllegalArgumentError` is raised.

As a convenience decisions may be passed as arguments insteaad of one or more of the *situationDecisions*. As in `SituationDecision` such a decision must be a Python object that is hashable, but is not `None`. For each such decision passed a SituationDecision will be implictly constructed with the decision and with all its attributes having the value `None`. If any decisions cited are not hashable, are `None`, or duplicate other decisions cited, including those cited in a `SituationDecision`, an `IllegalArgumentError` is raised.

For each of the *situationDecisions* it finds all instances in memory for the same decision in the same situation, and computes their activations at the current time based upon when in the past they have been seen, modified by the value of the `decay` property, and with noise added as controlled by the `noise` property. Looking at the activations of the whole ensemble of matching instances a retrieval probability is computed for each, and these are combined to arrive at blended value expected for each decision. This blending operation depends upon the value `temperature` property; if none is supplied a default is computed based on the `noise`. The decision chosen and returned is that with the highest blended value. In case of a tie, if `noise` is non-zero one will be chosen at random, and otherwise an arbitrary, but deterministic, choice is made among tied decisions. Note that the return value is the decision from the SitutaionDecision, and not the whole SituationDecision.

After a call to choose a corresponding response must be delivered to the agent with `respond()` before calling choose again, or an `IllegalStateError` will be raised.

decay

Controls the rate at which activation for previously experienced instances in memory decay with the passage of time. Time in this sense is dimensionless, and simply the number choose/respond cycles that have occurred since the agent was created or last `reset()`, as reported by the *tTrial or tIteration fields in log files*. Typically a positive, possibly floating point, number between about 0.5 to about 10. If zero, the default, memory does not decay. If set to `None` it revert it reverts the value to its default, zero.

defaultUtility

The utility, or a function to compute the utility, if there is no matching instance for a `SituationDecision`. When `choose()` is called, for each SituationDecision passed to it it is first ascertained whether or not an instance exists that matches that decision in the given situation. If there is not, the value of this property is consulted. Note that an instance added with `prepopulate()` counts as matching, and will prevent the interrogation of this property.

The value of this property may be a number, in which case when needed it is simply used as the default utility. If it is not a number, it is assumed to be a function that takes one argument, a SituationDecion. When a default utility is needed that function will be called, passing the SituationDecision in question to it, and value returned, which should be a number, will be used. If at that time the value is not a function of one argument, or it does not return a number, an `IllegalStateError` is raised.

The `defaultUtilityPopulates` property, which is `True` by default, controls whether or not an instance is added for each interrogation of the attr:*defaultUtility* property. If an instance is added, it is added as by `prepopulate()`. Note that, except for the first interrogation of this property, such added

instances will have timestamps greater than zero.

Setting this property to `None` or `False` causes no default probability to be used. In this case, if `choose()` is called for a decision in a situation for which there is no instance available, an `IllegalStateError` will be raised.

`defaultUtilityPopulates`
Whether or not a default utility provided by the `defaultUtility` property is also entered as an instance in memory. This property has no effect is defaultUtility is `None` or `False`.

`logToDatabase`(*database*, *table*, *create=False*, *clear=False*)
Calls the `Population.logToDatabase()` method of this agent's `population`, with the same arguments. Note that calling this affects the logging behavior of any other agents that belong to the same population.

`logToFile`(*file*, *heading=True*, *precision=4*, *dialect='excel'*)
Calls the `Population.logToFile()` method of this agent's `population`, with the same arguments. Note that calling this affects the logging behavior of any other agents that belong to the same population.

`logToList`(*lst=None*)
Calls the `Population.logToList()` method of this agent's `population`, with the same *lst* argument. Note that calling this affects the logging behavior of any other agents that belong to the same population.

`logging`
The `Population.logging` property of this agent's `population`. May be assigned to, in which case it sets the agent's population's logging property. Note that so assigning to this affects the logging behavior of any other agents that belong to the same population.

`name`
The name of this Agent. It is a string, provided when the agent was created.

`noise`
The amount of noise to add during instance activation computation. This is typically a positive, possibly floating point, number between about 0.5 and 10. If zero, the default, no noise is added during activation computation. If set to `None` it reverts the value to its default, zero. If an explicit `temperature` is not set, the value of noise is also used to compute a default temperature for the value blending operation.

`occurrencesLimit`
The `Population.occurencesLimit` property of this agent's `population`. May be assigned to, in which case it sets the agent's population's occurrencesLimit property. Note that so assigning to this affects the logging behavior of any other agents that belong to the same population.

`population`
The `Population` to which this Agent currently belongs. Every agent belongs to some Population. When an agent is freshly created it belongs to a freshly created Population that will be discarded if this agent is subsequently reassigned to a different population, by assigning to this property. Setting this property to `None` causes a new Population to be created and set as this agent's population. Attempting to assign to this property anything than a Population or `None` raises an `IllegalArgumentError`.

`prepopulate`(*outcome*, **situationDecisions*)
Adds instances to memory, one for each of the *situationDecisions*, with the given outcome, at the current time, without advancing that time. Time is a dimensionless quantity, simply a count of the number of choose/respond cycles that have occurred since the agent was created or last `reset()`, and is report in logs using the *tTrial or tIteration columuns*.

This is typically used to enable startup of a model by adding instances before the first call to `choose()`. When used in this way the timestamp associated with this occurrence of the instance will be zero. Subsequent occurrences are possible if `respond()` is called with the same outcome after `choose()` has

returned the same decision in the same situation, in which case those reinforcing occurrences will have later timestamps. An alternative mechanism to facility sartup of a model is setting the `defaultUtility` property of the agent. While rarely done, a modeler can even combine the two mechanisms, if desired.

It is also possible to call prepopulate after choose/respond cycles have occurred. In this case the instances are added with the current time as the timestamp. This is one less than the timestamp that would be used were an instance to be added by being experienced as part of a choose/respond cycle instead. Each agent keeps internally a clock, the number of choose/respond cycles that have occurred since it was created or last `reset()`. When `choose()` is called it advances that clock by one *before* computing the activations of the existing instances, as it must as the activation computation depends upon all experiences having been in the past. That advanced clock is the timestamp used when an instance is added or reinforced by `respond()`. If an attempt is made to add a prepopulated instance for a decision in a situation at a time for which an instance has already occurred a warning will be issued, but the instance will still be added or reinformaced as appropriate.

respond(*outcome*, *flush=True*, *close=False*)

Provide the *outcome* resulting from the most recent decision returned by `choose()`. The *outcome* should be a non-complex number, where larger numbers are considered "better." This results in the creation or reinforcemnt of an instance in memory for the decision, in the situation it had when chosen, with the given outcome, and is the fundamental way in which the PyIBL model "learns from experience."

If there has not been a call to choose since the last time respond was called an `IllegalStateError` is raised. If *outcome* is not a non-complex number an `IllegalArgumentError` is raised.

If *flush* is not false and this agent's `population` has logging enabled, any logging information still buffered at the conclusion of respond's actions will be written to disk, the database, or the open stream, as appropriate; *flush* has no special effect if the log is being written to a list. If *close* is not false the log file or database will be closed; it has no effect on an open stream passed to `Population.logToFile()` nor on logging to a list. Note that if a log file or database is so closed, it will automatically be reopened next time there is a need to write to it.

temperature

The temperature parameter in the Boltzman Equation used for blending values. If `none`, the default, the square root of 2 times the value of `noise` will be used. If `None` and there is also no noise, a waring will be printed and a default of 1 will be used for the temperature.

4.2 SituationDecisions

class `pyibl.`**SituationDecision**(*agent*, *decision*, *situation*)

A possible decision paired with a collection current values of attributes, for an `Agent` to choose between. A SituationDecions should only be created by calling an agent's `Agent.situationDecision()` method, which will ensure it has the correct attributes for that agent. The attribute names are strings. The decision and all attribute values must be hashable Python objects, and the decision may not be `None`. The `decision` property of a SituationDecision is the decision; it may not be changed after the SituationDecision is created. The `attributes` property of a SituationDecision is a tuple of the attribute names of its situation. The values of those attributes may be retrieved as a tuple with the `situation` property. Individual attribute values may be retrieved with the `get()` method, and changed with the `set()` method. The agent that created a SituationDecision can be found using its `agent` property

agent

The `Agent` that created this SitutationDecision, and with which it remains associated.

decision

The decision this SituationDecision represents. The decision is set when the SitutationDecision is created by an agent and may not be changed. It is typically a string or number, but may be any hashable object.

get (*attribute*)
Returns value of the attribute named *attribute* of the situation of this SituatonDecision

set (*attribute*, *value*)
Sets the value of the the attribute named *attribute* in this SituationDecisiion. The provided *value* must be hashable; if it is not an `IllegalArgumentError` is raised.

situation
A tuple summarizing the situation this SituationDecision currently represents. Each attribute of the situation is an element of the returned tuple, in the same order as the :class"*Agent*'s attributes are represented in its `attributes` property. The value of an attribute must be hashable. The values of all the situtation's attributes can be set by setting this property to a tuple or list of the correct length. An `IllegalArgumentError` is raised if the tuple or list is longer than the number of attributes, or if any of the values supplied is not hashable. If the tuple or list supplied is shorter than the number of attributes in this SituationDecision the trailing ones are left unchanged. The values of individual attributes can be retrieved with the `get()` method and changed with the `set()` method of this SituationDecision.

4.3 Logging and Populations

PyIBL can write a potentially extensive log file of the operations it is asked to perform, and the details of the computations agents make in arriving at choices. Instead of writing it to a file, it is also possible to direct that it be written to a SQLite database, or made available in memory for further manipulation by Python code.

When multiple PyIBL `Agent`s are cooperating in a single surrounding framework it is usually most convenient to have all their logging information consolidated in a single log, albeit keyed by `Agent.name`. For this reason logging is controlled by a `Population` object, to which one or more Agents belong. All Agents belong to exactly one Population, each Population containing zero or more Agents. Logging is configured with methods and properties of Populations. For convenience, most such methods and properties also exist for Agents, in which case they delegate the corresponding action to their Population; not that this means any logging configuring set using methods and properties of an Agent will affect all other Agents belonging to the same Population.

The destination for logging information is set by the most recent call to one of the methods `logToFile()`, `logToDatabase()` or `logToList()`. If none of those methods is called, but logging information has been requested, it will be printed to the standard output. The user stipulates that certain pieces of information be included in the log using the `Population.logging` property. This is set to one or more strings, described below. Setting it to `None`, or to an empty set, turns off logging. When logging is performed, at least one line of the log (or row in the database table, or sublist of the list of data made available in Python) is produced for each `choose()`/`respond()` cycle. Depending upon the details to be included in the log, there may be multiple lines written for each such cycle.

Most of the various strings specifying information to be logged begin with one of the letters *t*, *o* or *i*, indicating whether the information so logged varies by trial, by option, or by instance. If all the fields being logged are of trial level, only one line will be written for each `choose()`/`respond()` cycle of an agent. If all the fields being logged are of either trial or option level, at each cycle a line will be written for every option available at the time of that cycle. The option level information may vary between lines, but all the trial level information will be duplicated between lines. If any of the instance level fields are requested there will be a line written for every instance in the agent at the time of the cycle. All the trial level information will be duplicated in all these lines, and for each possible option, all the option level will be duplicated in the corresponding lines.

The various fields are always written in a fixed order. Fields that are not requested are omitted, but those that are included are never reordered. The descriptions of the various constants below are in the order in which the fields are written to the file. Note that case is significant when writing these strings: for example, it is an error to supply 'tagent' or 'TAGENT' instead of 'tAgent'.

'sequence'
When included in a `Population`'s `logging` property a column headed `sequence` will be added to the log, containing a sequence number counting which line of the file it is, starting with 1. For logs stored in CSV files

or lists this is implicit in their structure, and the sequence is unlikely to be of use; however, for logs stored in database tables that are not guaranteed to have a definite order this can be helpful.

'tAgent'
When included in a Population's logging property a column headed tAgent will be added to the log, containing the name of the agent for which this row of the log is reporting.

'tDecayParam'
When included in a Population's logging property a column headed tDecayParam will be added to the log, containing this agent's decay parameter value at the time the log entry was written.

'tNoiseParam'
When included in a Population's logging property a column headed tNoiseParam will be added to the log, containing this agent's decay parameter value at the time the log entry was written.

'tTemperature'
When included in a Population's logging property a column headed tTemperature will be added to the log, containing this agent's temperature value, used to compute instances' retrieval probabilities, at the time this log entry was written. If the Population.temperature property has been set to a non-None value with setTemperature(), that value will be used. If it is None a default value will be used: this is normally the Agent.noise parameter times the square root of 2, but if noise is zero, or nearly zero, a warning is printed (once, only), and a default value of 1 is used for the temperature. Whatever value is actually used will be reflected in the tTemperature columns of the log.

'tBlock'
When included in a Population's logging property a column headed tBlock will be added to the log, containing the most recently set value of the Population's Population.block property.

'tTrial'
When included in a Population's logging property a column headed tTrial will be added to the log, containing the number of choose/respond cycles that have occurred since the agent was last reset(), or the Population.block was changed. This is one based: it will be 1 for the first trial, and so on. If the only instances in an agent are prepopulatted ones this will be zero. Trial and iteration only differ when an agent is not always reset when a block changes, meaning that learning is carried over between problems or participants.

'tIteration'
When included in a Population's logging property an column headed tIteration will be added to the log, containing the number of choose/respond cycles that have occurred since the agent was last reset(). This is one based: it will be 1 for the first iteration, and so on. If the only instances in an agent were added with Agent.prepopulate() before the first choose/respond cycle this will be zero. Trial and iteration only differ when an agent is not always reset when a block changes, meaning that learning is carried over between problems or participants.

The following table demonstrates the interaction between tBlock, tTrial and tIteration. Each row of the table corresponds to one choose/respond cycle of the PyIBL model. If the other actions described in the first column take place on the Population pop of that model, the values that are put into the log are as shown.

Action	tBlock	tIteration	tTrial
`pop.block = 'A'` `choose/respond`	A	1	1
`choose/respond`	A	2	2
`choose/respond`	A	3	3
`pop.block = 'B'` `choose/respond`	A	4	1
`choose/respond`	A	5	2
`pop.block = 'C'` `choose/respond`	C	6	1
`choose/respond`	C	7	2
`choose/respond`	C	8	3

`pop.resetAgents()`
`pop.block = 'D'`
`choose/respond`
	D	1	1

`choose/respond`	D	2	2
`pop.block = 'E'` `choose/respond`	E	3	1
`choose/respond`	E	4	2
`choose/respond`	E	5	3

`'tChoice'`
: When included in a `Population`'s `logging` property a column headed `tChoice` will be added to the log, containing the decision actually chosen at this stage, and returned as the value of `Agent.choose()`.

`'tChoiceSituation'`
: When included in a `Population`'s `logging` property a column headed `tChoiceSituation` will be added to the log, containing the situation active for the decision chosen and returned as the value of `Agent.choose()`.

'tResponse'
When included in a Population's logging property a column headed tResponse will be added to the log, containing the outcome, the value supplied back to the agent by Agent.respond().

'oDecision'
When included in a Population's logging property rows will be added to the log for every known option in the agent, and a column headed oDecision will be added, containing with the name of the possible decision being described and evaluated.

'oSituation'
When included in a Population's logging property rows will be added to the log for every known option in the agent, and a column headed oSituation will be added, containing with the value of the situation for the situation-decision being described and evaluated.

'oBlendedValue'
When included in a Population's logging property rows will be added to the log for every known option in the agent, and a column headed oBlendedValue will be added to the log, containing the blended value for the option corresponding to this row.

'iUtility'
When included in a Population's logging property rows will be added to the log for every instance in the agent, and a column headed iUtility will be added, containing the utility for this instance.

'iReasonAdded'
When included in a Population's logging property rows will be added to the log for every instance in the agent, and a column headed iReasonAdded will be added, containing a description for why an instance was first added to the Agent's memory. It will be one of the three values

experienced The instance was first encountered in the usual way, by being experienced. That is, the instance's situation-decision was returned by Agent.choose(), the corresponding utility was first encountered in the corresponding call to Agent.respond().

prepopulated The instance was added by a call to Agent.prepopulate().

defaulted The instance was added based on the value of Agent.defaultUtility when Agent.defaultUtilityPopulates was true.

Note that the reason applies only to the first occurrence of a given instance. Any subsequent occurrences are always experienced ones.

'iFrequency'
When included in a Population's logging property rows will be added to the log for every instance in the agent, and an column headed iFrequency will be added to the log, containing a count of the number of times this instance has been seen.

'iOccurrences'
When included in a Population's logging property a column headed iOccurrences will be added to the log, containing a string listing the times (that is, the values of iteration) for all occurrences of this instance. The string contains a series of integers, in increasing order, separated by commas. This list can grow arbitrarily long. If preferred it can be limited by the Population's occurrencesLimit property, which limits the list to at most the specified number of recent occurrences. If this property is None, which is the default, there is no limit.

'iActivationBase'
When included in a Population's logging property rows will be added to the log for every instance in the agent, and an column headed iActivationBase will be added to the log, containing the value of the instances's activation, at this step, before noise is added to it.

'iActivationNoise'
When included in a Population's logging property rows will be added to the log for every instance in the

agent, and an column headed `iActivationNoise` will be added to the log, containing the noise that will be added to this instance's activation, at this step.

'iActivation'
When included in a `Population`'s `logging` property rows will be added to the log for every instance in the agent, and an column headed `iActivation` will be added to the log, containing the value of the activation of this instance at this step. It is the sum of the iActivationBase and iActivationNoise columns.

'iRetrievalprobability'
When included in a `Population`'s `logging` property rows will be added to the log for every instance in the agent, and a column headed `iRetrievalProbability` will be added to the log, containing the retrieval probability computed for this instance at this step.

'unusedDecisions'
When `Agent.choose()` is not always called with the same set of decisions, there will be instances in memory that need not be consulted when making a choice. If `'unusedDecisions'` is included in a `Population`'s `logging` property *and* appropriate details are being emitting to the log, lines will also be included for this otherwise unused, at this time, options and instances, together with the instances' current activations and retrieval probabilities, if requested. If blended values are requested, they will be blank for the lines corresponding to unused decisions, as there is no meaningful blended value in this case.

'unusedSitutations'
When `Agent.choose()` is called with varying situations for a dicision there will be instances in memory that need not be consulted when making a choice. If `'unusedSituationss'` is included in a `Population`'s `logging` property *and* appropriate details are being emitting to the log, lines will also be included for this otherwise unused, at this time, options and instances, together with the instances' current activations and retrieval probabilities, if requested. If blended values are requested, they will be blank for the lines corresponding to unused situations, as there is no meaningful blended value in this case.

It is often convenient to have a set of all the possible strings used to control logging. This can be used to generate the most complete log possible, or by using a set difference expression to remove just a few of the options, a log that is nearly as complete as possible.

`pyibl.`**LOG_ALL**
A `frozenset` containing all the possible option strings controlling logging detail.

class `pyibl.`**Population**
A collection of `Agent` objects, sharing logging information. Every Agent belongs to exactly one Population, and all the agents within a population share the same log, if one is enabled. Methods for configuring logs apply to populations. Log related methods of agents actually apply to that agent's population and so affect the logging behavior of all other agents that belong to that population. There is a fixed order of the agents in a population, the order they were added to it by setting their `population` property. A number of methods on Population simply delegate to a population's agents, being called on each agent, in that fixed order.

Logging is turned on, and the details of what information is to be included configured, by setting the `logging` property. Where to write the log is determinded by calling on of the methods `LogToFile()`, `LogToDatabase()` or `LogToList()`. The `agents` property returns a tuple of the agents that are a part of a population. If a population opens a log file (as opposed to receiving an already open file) or database, it can be closed by calling the population's `close()` method. A population can be used with Pythons `with` method to ensure that close is called when leaving a block of code, and such use is recommended.

agents
Returns a tuple of all the Agents currently in this Population. The order is the fixed order that they will be traversed by operations operating on all of them.

block
A hashable Python object that will be written to a log in the `'tBlock'` column. This is typically useful for keeping track of participants or experimental conditions. Multiple values can easily be multiplexed into this column by setting this property to a tuple. If block is being used as an integer counter a particularly

easy way to manipulate it may be with the `NEXT` constant. If an attempt is made to set this property to a non-hashable object, other than `NEXT`, an `IllegalArgumentError` is raised.

choose(**situationDecisions*)
Calls the `Agent.choose()` method of each agent in this population, and returns a tuple of the resulting decisions. The *situationDecisions* are all passed, in the same order, too all of the agents. The calls to choose are made on the agents in the order in which the agents appear in the population's `agents` property, which is the order they were added to the population. Note that for it to be possible to use this method all the agents in the population must expect exactly the same attributes. If this population has no agents an `IllegalStateError` is raised.

logToDatabase(*database*, *table*, *create=False*, *clear=False*)
Arranges for this population's log to be written to a table in a SQLite database. The *database* can be a string, in which case a database of that name is opened; or it can be a connection object which is used as is. The *table* is converted to a string and used as the name of the table into which to write log rows. If the boolean *create* is true (it is false by default) a table of the given name will be created with appropriate columns; otherwise it should exist already. if the boolean *clear* is true (it is false by default) when starting a new log all existing rows in the table will be deleted; otherwise the new rows are added to those already present.

logToFile(*file*, *heading=True*, *precision=4*, *dialect='excel'*)
Arranges for this population's log to be written to a file. If *file* is `None`, or another false value, the standard output is used. If *file* is a file object, or any other object implmenting the **write()** method, it is used as is. Otherwise is is converted to a string and used as a filename, which is opened for writing; the contents of any existing file of that name are lost. The `close()` method will only close a file that has been opened by this population. If an already open file, or `None`, was pass as the *file* argument this population will never attempt to close it.

The file is typically written in Commad Separated Values (CSV) format. The format can be change by providing a different *dialect*, a string. See the documentation of Python's csv module for details of what dialects are available. If *heading* is not false when first writing to this file a header row is written. Floating point numbers are rounded to the given *precision*, which defaults to four digits after the decimal point.

logToList(*destination=None*)
Arranges for this population's log to be "written" to a list in memory, for possible future manipulation. Returns the list to which logging information will be added. For each "row" of the log a sublist will be added to this list, and each "column" will be added as an element of that sublist. Unlike when writing to a file the objects added to the sublists are never converted to strings or otherwise reformatted. If *destination* is supplied it should be the list to which logging information is added; if it is `None`, the default, a new, empty list will be created and used. The *destination* need not actually be a list, it can be any object that implements the **extend()** method. If the value of *destination* does not implement that method, and is not *None*, an `IllegalArgumentException` is raised.

logging
A set of *strings describing columns to be added to the log*. As a convenience assigning a string to this property is equivalent to assigning a single element set containing just that string to it. Any other iterable of strings may also be assigned to this property, it being equivalent to assigning a set of those strings to it. If an empty set, or `None`, is assigned to this property it turns logging off. The default value of this property in a newly created population is `None`. If any of the strings assigned to this property are not among *those that name logging columns* an `IllegalArgumentError` is raised.

occurrencesLimit
The maximum number of values to include in the iOccurrences field of a log, or `None` if there is no limit.

prepopulate(*outcome*, **situationDecisions*)
Calls the `Agent.prepopulate()` method of all the Agents in this Population, with the given *outcome* and *situationDecisions*. Raises an `IllegalStateError` if this Population has no agents.

resetAgents()
Calls the `Agent.reset()` method of all the Agents in this Population. Raises an `IllegalStateError` if this Population has no agents.

respond(*outcomes*, *flush=True*, *close=False*)
Calls the `Agent.respond()` method of each agent in this population, passing them the values in *outcomes*. The agents are responded to in the order in which they were added to this population, the same order they appear in the `agents` property, and the values chosen from *outcomes* are selected in parallel in this same order. The *outcomes* should be an iterable of non-complex numbers.

If *flush* is not false and this population has logging enabled, any logging information still buffered at the conclusion of the last respond's actions will be written to disk, the database, or the open stream, as appropriate; *flush* has no special effect if the log is being written to a list. If *close* is not false the log file or database will be closed; it has no effect on an open stream passed to `logToFile()` nor on logging to a list. Note that if a log file or database is so closed, it will automatically be reopened next time there is a need to write to it.

An `IllegalArgumentError` is raised if *outcomes* does not contain the same number of values as this population contains agents.

setDecay(*value*, **agentNames*)
Sets the `Agent.decay` property of all the Agents in this Population to *value*. Raises an `IllegalStateError` if this Population has no agents.

setDefaultUtility(*value*)
Sets the `Agent.defaultUtility` property of all the Agents in this Population to *value*. Raises an `IllegalStateError` if this Population has no agents.

setDefaultUtilityPopulates(*value*)
Sets the `Agent.defaultUtilityPopulates` property of all the Agents in this Population to *value*. Raises an `IllegalStateError` if this Population has no agents.

setNoise(*value*)
Sets the `Agent.noise` property of all the Agents in this Population to *value*. Raises an `IllegalStateError` if this Population has no agents.

setTemperature(*value*)
Sets the `Agent.temperature` property of all the Agents in this Population to *value*. Raises an `IllegalStateError` if this Population has no agents.

pyibl.**NEXT**
A constant used for incrementing the value of `Population.block` property of a `Population`. When that property is set to `NEXT`, if it is already a number, it will be incremented by 1; otherwise it will be set to 1.

4.3.1 Errors

exception pyibl.**IllegalArgumentError**
Raised by many PyIBL methods when passed a somehow defective argument; inherits from `ValueError`.

exception pyibl.**IllegalStateError**
Raised by many PyIBL methods when called in a way that exposes some incorrect internal state; inherits from `RuntimeError`.