

# SLAPP (Swarm-Like Protocol in Python) Reference Handbook

Pietro Terna

<mailto:pietro.terna@unito.it>

July 25, 2017

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	SLAPP and Swarm	4
1.2	Libraries to use SLAPP	4
1.3	SLAPP online	5
1.4	The README and the related files: discovering the two contents of SLAPP	5
1.4.1	Using SLAPP as a tutorial on agent-based programming foundations	6
1.4.2	Using SLAPP as an agent-based shell	7
<b>2</b>	<b>The <i>basic</i> project as a guide to the making of a new project</b>	<b>19</b>
2.1	How to <i>run</i> SLAPP	19
2.2	Scheduling	22
2.2.1	The scheduling mechanism at the level of the Observer	23
2.2.2	The scheduling mechanism at the level of the Model	26
2.2.3	The detailed scheduling mechanism within the Model (AE-SOP level), with <i>WorldState</i> and <i>macros</i>	30
2.2.3.1	The <i>WorldState</i> feature as a variable repository	33
2.2.3.2	The <i>WorldState</i> feature as a computational tool	34
2.2.3.3	The <i>macros</i>	34
2.2.3.4	The <i>if</i> structure	35
2.2.3.5	Agents adding and eliminating tasks into the detailed schedule	35
2.3	The agents and their sets	39
2.3.1	Sets of agents	40
2.3.2	The use of files <i>.txtx</i> to define the agents	41
2.3.3	Future developments about agents	42
<b>3</b>	<b>SLAPP multi-class: the <i>basic2classes</i> example</b>	<b>43</b>
3.1	Introducing the example	43

<b>4</b>	<b>Debugging a new project: the <i>debug</i> project as a micro tutorial</b>	<b>46</b>
<b>5</b>	<b>Other existing and upcoming projects</b>	<b>48</b>
5.1	Adding <i>turtles</i> : the <i>school</i> project . . . . .	48
5.2	Adding networks: the <i>production</i> project . . . . .	48
5.3	New projects and extensions . . . . .	48
5.3.1	Connecting to R, via Rserve . . . . .	48
5.3.2	Connecting to other applications, via Redis . . . . .	48
5.3.3	Parallel computations in SLAPP . . . . .	49
<b>6</b>	<b>SLAPP in IPython/Jupyter</b>	<b>50</b>
6.1	Running SLAPP in an IPython/Jupyter notebook . . . . .	50
6.2	Size of the pictures in IPython/Jupyter notebook . . . . .	50
6.3	<i>Turtle</i> graphics and IPython . . . . .	51
	<b>Appendices</b>	<b>52</b>
<b>A</b>	<b>Libraries for SLAPP</b>	<b>53</b>
A.1	Using Linux (via the Ubuntu distribution) . . . . .	54
A.2	Using Mac OS X . . . . .	55
A.3	Using Windows (referring to Windows 10) . . . . .	57
<b>B</b>	<b>On SLAPP execution</b>	<b>59</b>
<b>C</b>	<b>Problems with libraries</b>	<b>60</b>
C.1	A warning about fonts coming from <i>matplotlib</i> 1.5.1 . . . . .	60
<b>D</b>	<b>On <i>turtles</i></b>	<b>61</b>
	<b>Bibliography</b>	<b>63</b>
	<b>Index</b>	<b>64</b>

## List of Figures

1.1	Starting the <i>basic</i> project . . . . .	8
1.2	The output of the <i>basic</i> project . . . . .	9
1.3	Starting the <i>school</i> project . . . . .	10
1.4	The plain text output of the <i>school</i> project . . . . .	11
1.5	The graphical output of the <i>school</i> project . . . . .	12
1.6	Starting the <i>production</i> project . . . . .	13
1.7	The plain text output of the <i>production</i> project . . . . .	14
1.8	The graphical output of the <i>production</i> project . . . . .	15
1.9	Launching <i>Jupyter</i> from the SLAPP folder . . . . .	16
1.10	<i>Jupyter</i> main window . . . . .	17
1.11	The <i>basic</i> project running in <i>Jupyter</i> . . . . .	18
2.1	The representation of the schedule . . . . .	23
D.1	The Logo Foundation, at <a href="http://el.media.mit.edu/logo-foundation/">http://el.media.mit.edu/logo-foundation/</a> . . . . .	62

# Chapter 1

## Introduction

### 1.1 SLAPP and Swarm

SLAPP, as Swarm-Like Protocol in Python,<sup>1</sup> contains both a tutorial on agent-based programming foundations and a shell to run large simulation projects, having in mind the original Swarm<sup>2</sup> scheme.

Studying the tutorial is not strictly necessary to use SLAPP as an agent-based modeling *shell*. The tutorial is kept within the SLAPP distribution mainly to maintain the original choice of the Swarm project, but the shell is close to become an independent element.

The repository of SLAPP is at <https://github.com/terna/SLAPP>.

To read about Swarm and SLAPP, to examine several SLAPP applications, and ... a lot more, you can have a look to Boero *et al.* (2015) book.

### 1.2 Libraries to use SLAPP

To use SLAPP you need to install several Python libraries<sup>3</sup>. To do that, please follow Appendix A.

---

<sup>1</sup>From its 3.0 version, SLAPP is running in Python 3; at <https://github.com/terna/SLAPP/> you can also find the 2.0 version, related to Python 2; for the maintenance versions of the Python 2 implementation, we will adopt 2.0.x numbering.

From version 3.0.1 SLAPP adopts the PEP8, the Style Guide for Python Code at <https://www.python.org/dev/peps/pep-0008/>.

Due to this adoption, the reader can notice some aesthetic differences between the code reported here in the Handbook and that listed into the files.

<sup>2</sup>About Swarm, have a look to Minar *et al.* (1996). You can access Swarm website via <http://www.swarm.org>. The project started at the Santa Fe Institute (first release: 1994). It represents a milestone in agent-based simulation.

<sup>3</sup>From v.3.0, related to Python 3.

## 1.3 SLAPP online

To familiarize with SLAPP, we have a running version of SLAPP at <https://beta.mybinder.org/v2/gh/terna/SLAPP3/master?filepath=iRunShellOnline.ipynb>, via the wonderful Binder project (<https://beta.mybinder.org/>): try it!

It is also possible to view the file system, at <https://beta.mybinder.org/v2/gh/terna/SLAPP3/master>, to explore SLAPP and its application (to launch SLAPP click on `iRunShellOnline.ipynb`).

If you experience troubles in loading the running application or its file system, try to delete the cache of your browser or—better—open a private window (anonymous, in incognito, ...).

For advanced users:

1. (July 2017, temporary not working) with New/Terminal from the entry online page (file system view), you can directly work in a virtual machine (with a bash shell);
2. opening a .txt or .py file via the entry online page (file system view) you can edit and save it (in your tmp environment);
3. the button Upload in the entry online page allows you to upload files;

With Cell/All Output/Toggle Scrolling you can see the output of the online simulation in a scrolling window or in a plain one.

The online structure is based on JupyterHub<sup>4</sup>.

## 1.4 The README and the related files: discovering the two contents of SLAPP

The GitHub repository of SLAPP contains two README files.

- The `_readmeFirst.txt` file clarifies the content of the whole project.

We have both a tutorial and an agent-based simulation shell, coming from the Swarm (<http://www.swarm.org>) project, and named SLAPP for Swarm-Like Agent Protocol in Python.

You can find SLAPP as an Agent-based Model (ABM) shell, in the folder number 6.<sup>5</sup>

---

<sup>4</sup><https://github.com/jupyterhub/jupyterhub>.

<sup>5</sup>6 *objectSwarmObserverAgents\_AESOP\_turtleLib\_NetworkX*

Both the basic scheme of the tutorial, and all the files having in their names the prefix `Swarm_original`, are coming from the tutorial that was distributed by the Swarm Development Group via the `swarmapps` file (the last version, that we use here, is `swarmapps-objc-2.2-3.tar.gz`).

Those files are unmodified in SLAPP, but the correction of a few typos.

We can find the original package at <http://download.savannah.gnu.org/releases/swarm/apps/objc/sdg/swarmapps-objc-2.2-3.tar.gz> or at <http://nongnu.askapache.com/swarm/apps/objc/sdg/swarmapps-objc-2.2-3.tar.gz> or at <http://terna.to.it/swarm/swarmapps-objc-2.2-3.tar.gz>.

- The `README.md` file, written using *Markdown*,<sup>6</sup> introduces the new user to all the information reported above, underlining again that we have two possible ways of using SLAPP: both as a tutorial on agent-based programming foundations (see Section 1.4.1) or as an agent-based shell (see Section 1.4.2).

#### 1.4.1 Using SLAPP as a tutorial on agent-based programming foundations

- To study the tutorial, read first of all the content of the file `SLAPP tutorial.txt`, which is in the main SLAPP folder..
- The file `SLAPP tutorial.txt` guides the user through the development of an agent-based model that makes use of a lot of the ideas of Swarm; ideas now rooted in the SLAPP structure.

The model refers to the movement of a bug, randomly walking in a 2D space.

We start introducing a very simple and plain program, with the bug taking a random walk. Through a progression of models, we introduce both object-oriented and Swarm style programming.

Although this is a quite simple exercise, it shows how to build complex software from simple building blocks.

In this folder, we have several subfolders, each with a complete application and a `README` file that helps you to walk through the code.

You should start with the 1 `plainProgrammingBug` folder, and then proceed in the following order (the start files have a number, corresponding to that of their folder):

1 `plainProgrammingBug`

---

<sup>6</sup><http://whatismarkdown.com>

```
2 basicObjectProgrammingBug
3 basicObjectProgrammingManyBugs
4 basicObjectProgrammingManyBugs_bugExternal+_shuffle
5 objectSwarmModelBugs
6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX
7 (toBeDeveloped_aFewHints)
```

- We used Python to write the tutorial: you can find a lot of wonderful resources introducing the Python language. I suggest [Downey \(2012\)](#), a book that you can also read online at the address reported in the references; this book also exists in a slight different version, as a learning interactive tool ([Elkner et al., 2013](#)). An alternative way to start learning Python is represented by the introductory part of the wonderful online book of [Sargent and Stachurski \(2013\)](#) on quantitative economics. (There, you can also find an introduction to Julia<sup>7</sup>, a quite recent and highly powerful language.)
- For the correct attribution of the authorship, we recall the file `Swarm_original README in tu` related to the original tutorial. The file is in the main folder of SLAPP.

Note that the names of the `txt` files, here and in the subfolders, start with the prefixes `SLAPP` or `Swarm_original`. This choice is just to underline if we are referring to my reformulation in Python or to the original Swarm elements. (Swarm was based on Objective C<sup>8</sup> and successively also on Java; the tutorial was in Objective C).

### 1.4.2 Using SLAPP as an agent-based shell

- To start running the agent-based shell, you can read the content of the file: `SLAPP shell.txt` and install the required libraries (to install them, you can follow the explanations of Appendix A). Then open a terminal, go into the SLAPP main folder (where you have unzipped the SLAPP distribution obtained from <https://github.com/terna/SLAPP> and:<sup>9</sup>

---

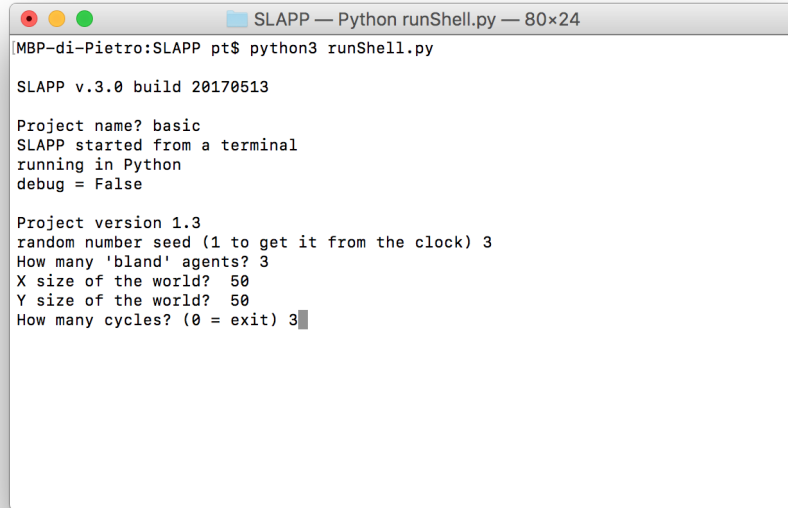
<sup>7</sup><http://julia-lang.org>; highly interesting <https://www.juliabox.org>

<sup>8</sup><https://en.wikipedia.org/wiki/Objective-C>

<sup>9</sup>About *running* SLAPP, read Section 2.1.



1 - launch the application **basic** as in Figure 1.1:<sup>10</sup>



```
MBP-di-Pietro:SLAPP pt$ python3 runShell.py

SLAPP v.3.0 build 20170513

Project name? basic
SLAPP started from a terminal
running in Python
debug = False

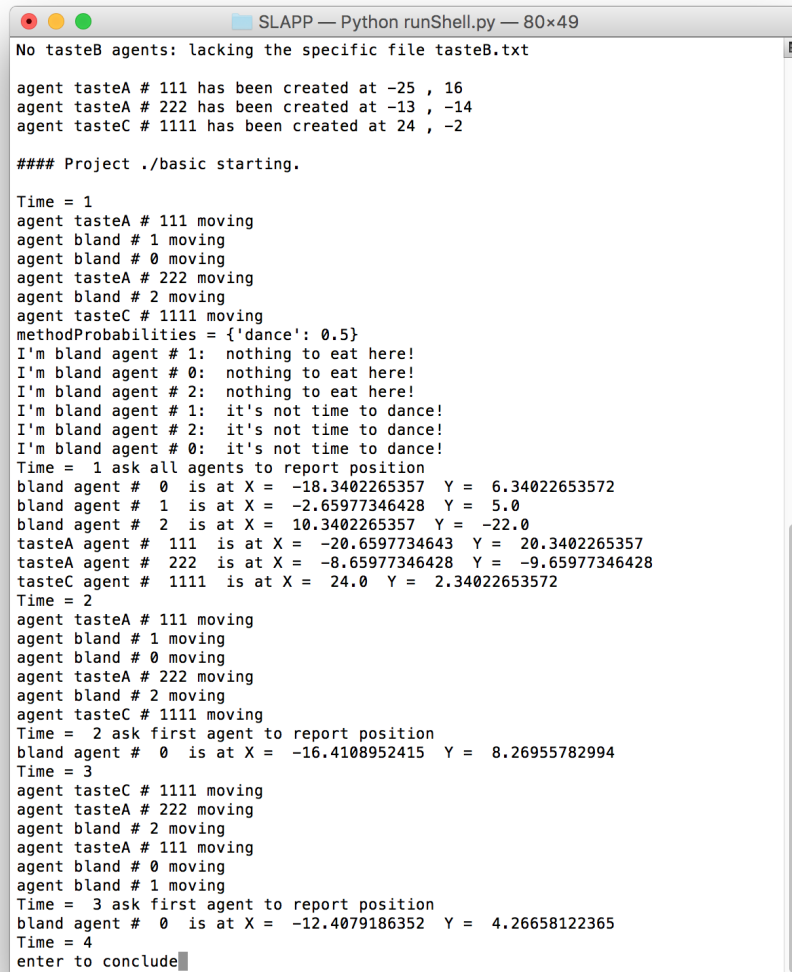
Project version 1.3
random number seed (1 to get it from the clock) 3
How many 'bland' agents? 3
X size of the world? 50
Y size of the world? 50
How many cycles? (0 = exit) 3
```

Figure 1.1: Starting the *basic* project

The effect is (plain text output only):

---

<sup>10</sup>Launching steps are also described in the initial *bullets* of Chapter 2.



```
No tasteB agents: lacking the specific file tasteB.txt

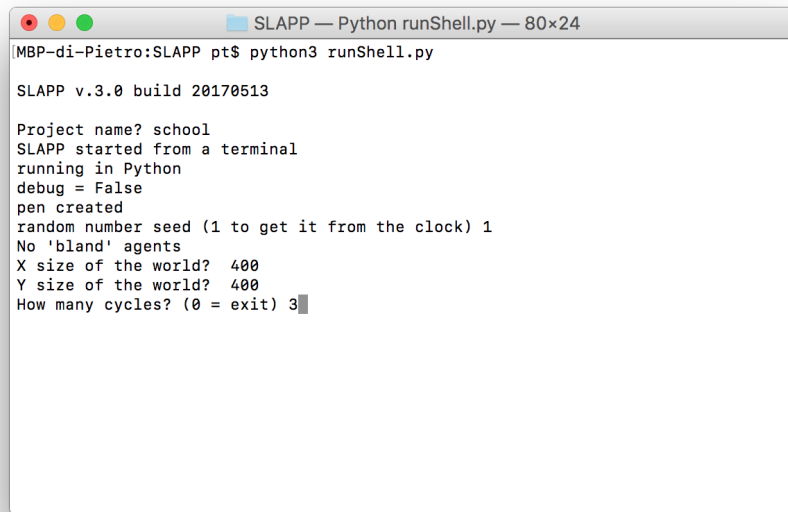
agent tasteA # 111 has been created at -25 , 16
agent tasteA # 222 has been created at -13 , -14
agent tasteC # 1111 has been created at 24 , -2

#### Project ./basic starting.

Time = 1
agent tasteA # 111 moving
agent bland # 1 moving
agent bland # 0 moving
agent tasteA # 222 moving
agent bland # 2 moving
agent tasteC # 1111 moving
methodProbabilities = {'dance': 0.5}
I'm bland agent # 1: nothing to eat here!
I'm bland agent # 0: nothing to eat here!
I'm bland agent # 2: nothing to eat here!
I'm bland agent # 1: it's not time to dance!
I'm bland agent # 2: it's not time to dance!
I'm bland agent # 0: it's not time to dance!
Time = 1 ask all agents to report position
bland agent # 0 is at X = -18.3402265357 Y = 6.34022653572
bland agent # 1 is at X = -2.65977346428 Y = 5.0
bland agent # 2 is at X = 10.3402265357 Y = -22.0
tasteA agent # 111 is at X = -20.6597734643 Y = 20.3402265357
tasteA agent # 222 is at X = -8.65977346428 Y = -9.65977346428
tasteC agent # 1111 is at X = 24.0 Y = 2.34022653572
Time = 2
agent tasteA # 111 moving
agent bland # 1 moving
agent bland # 0 moving
agent tasteA # 222 moving
agent bland # 2 moving
agent tasteC # 1111 moving
Time = 2 ask first agent to report position
bland agent # 0 is at X = -16.4108952415 Y = 8.26955782994
Time = 3
agent tasteC # 1111 moving
agent tasteA # 222 moving
agent bland # 2 moving
agent tasteA # 111 moving
agent bland # 0 moving
agent bland # 1 moving
Time = 3 ask first agent to report position
bland agent # 0 is at X = -12.4079186352 Y = 4.26658122365
Time = 4
enter to conclude
```

Figure 1.2: The output of the *basic* project

**2** - launch the application "school" as in the following window:

A terminal window titled "SLAPP — Python runShell.py — 80x24" is shown. The prompt is "MBP-di-Pietro:SLAPP pt\$ python3 runShell.py". The output shows the program version "SLAPP v.3.0 build 20170513", followed by a series of prompts and user inputs: "Project name? school", "SLAPP started from a terminal", "running in Python", "debug = False", "pen created", "random number seed (1 to get it from the clock) 1", "No 'bland' agents", "X size of the world? 400", "Y size of the world? 400", and "How many cycles? (0 = exit) 3".

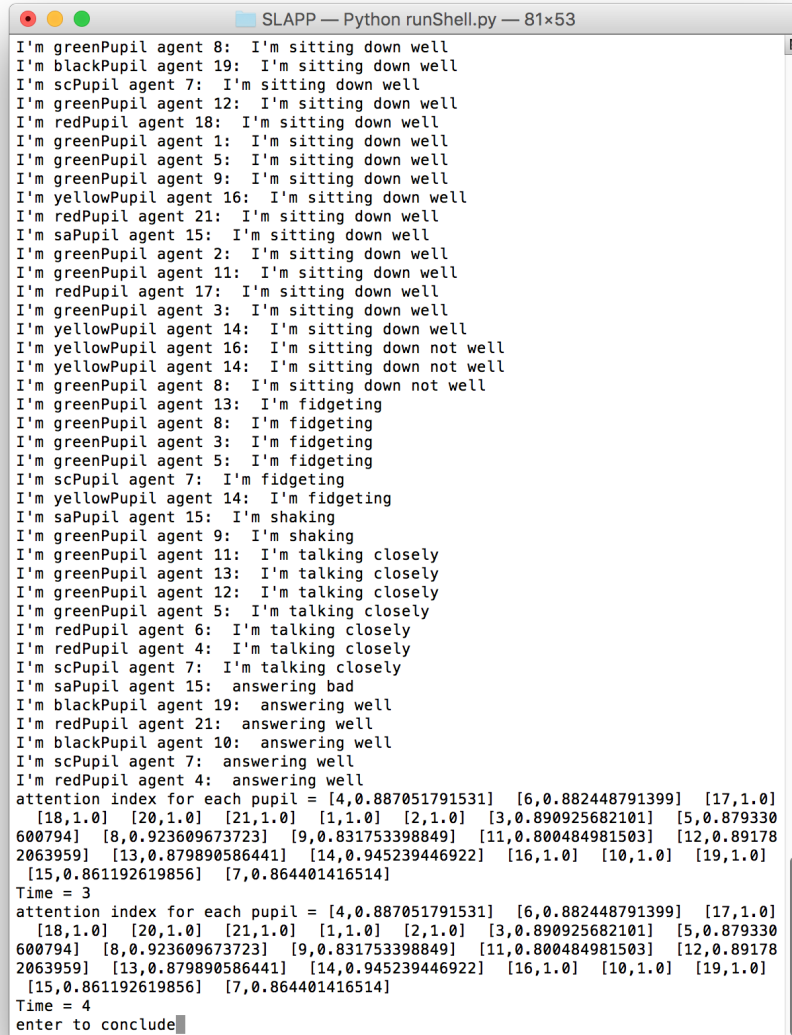
```
SLAPP — Python runShell.py — 80x24
MBP-di-Pietro:SLAPP pt$ python3 runShell.py

SLAPP v.3.0 build 20170513

Project name? school
SLAPP started from a terminal
running in Python
debug = False
pen created
random number seed (1 to get it from the clock) 1
No 'bland' agents
X size of the world? 400
Y size of the world? 400
How many cycles? (0 = exit) 3
```

Figure 1.3: Starting the *school* project

The effect is (plain text output):



```
I'm greenPupil agent 8: I'm sitting down well
I'm blackPupil agent 19: I'm sitting down well
I'm scPupil agent 7: I'm sitting down well
I'm greenPupil agent 12: I'm sitting down well
I'm redPupil agent 18: I'm sitting down well
I'm greenPupil agent 1: I'm sitting down well
I'm greenPupil agent 5: I'm sitting down well
I'm greenPupil agent 9: I'm sitting down well
I'm yellowPupil agent 16: I'm sitting down well
I'm redPupil agent 21: I'm sitting down well
I'm saPupil agent 15: I'm sitting down well
I'm greenPupil agent 2: I'm sitting down well
I'm greenPupil agent 11: I'm sitting down well
I'm redPupil agent 17: I'm sitting down well
I'm greenPupil agent 3: I'm sitting down well
I'm yellowPupil agent 14: I'm sitting down well
I'm yellowPupil agent 16: I'm sitting down not well
I'm yellowPupil agent 14: I'm sitting down not well
I'm greenPupil agent 8: I'm sitting down not well
I'm greenPupil agent 13: I'm fidgeting
I'm greenPupil agent 8: I'm fidgeting
I'm greenPupil agent 3: I'm fidgeting
I'm greenPupil agent 5: I'm fidgeting
I'm scPupil agent 7: I'm fidgeting
I'm yellowPupil agent 14: I'm fidgeting
I'm saPupil agent 15: I'm shaking
I'm greenPupil agent 9: I'm shaking
I'm greenPupil agent 11: I'm talking closely
I'm greenPupil agent 13: I'm talking closely
I'm greenPupil agent 12: I'm talking closely
I'm greenPupil agent 5: I'm talking closely
I'm redPupil agent 6: I'm talking closely
I'm redPupil agent 4: I'm talking closely
I'm scPupil agent 7: I'm talking closely
I'm saPupil agent 15: answering bad
I'm blackPupil agent 19: answering well
I'm redPupil agent 21: answering well
I'm blackPupil agent 10: answering well
I'm scPupil agent 7: answering well
I'm redPupil agent 4: answering well
attention index for each pupil = [4,0.887051791531] [6,0.882448791399] [17,1.0]
[18,1.0] [20,1.0] [21,1.0] [1,1.0] [2,1.0] [3,0.890925682101] [5,0.879330
600794] [8,0.923609673723] [9,0.831753398849] [11,0.800484981503] [12,0.89178
2063959] [13,0.879890586441] [14,0.945239446922] [16,1.0] [10,1.0] [19,1.0]
[15,0.861192619856] [7,0.864401416514]
Time = 3
attention index for each pupil = [4,0.887051791531] [6,0.882448791399] [17,1.0]
[18,1.0] [20,1.0] [21,1.0] [1,1.0] [2,1.0] [3,0.890925682101] [5,0.879330
600794] [8,0.923609673723] [9,0.831753398849] [11,0.800484981503] [12,0.89178
2063959] [13,0.879890586441] [14,0.945239446922] [16,1.0] [10,1.0] [19,1.0]
[15,0.861192619856] [7,0.864401416514]
Time = 4
enter to conclude
```

Figure 1.4: The plain text output of the *school* project

and as graphical output:

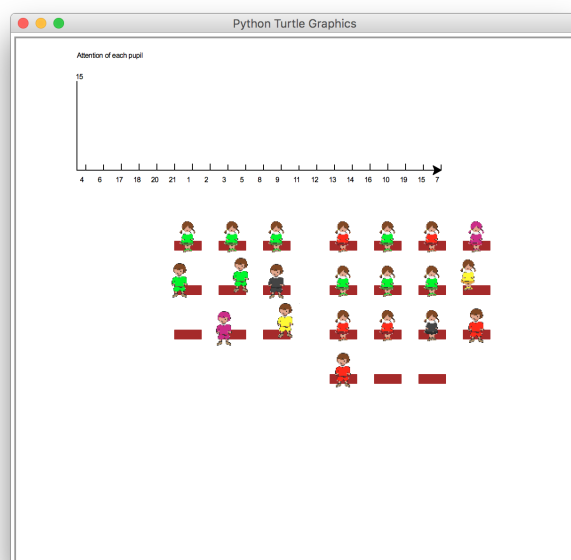
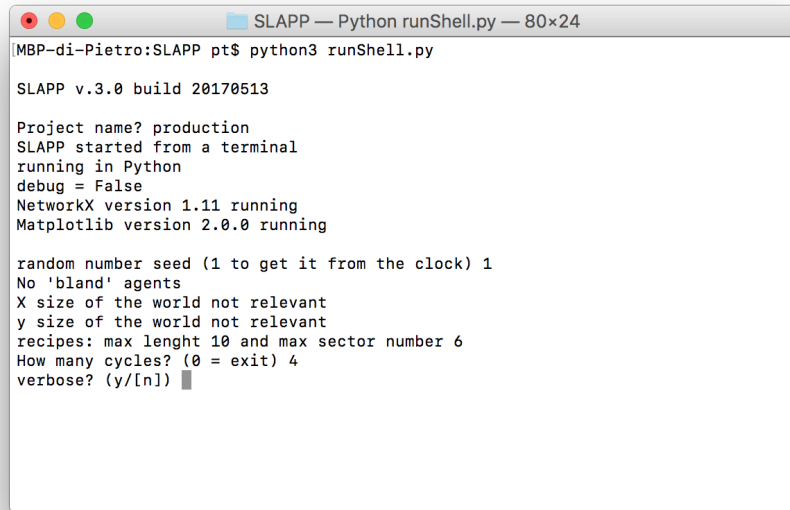


Figure 1.5: The graphical output of the *school* project

**3** - launch the application "production" as in the following window:

A terminal window titled "SLAPP — Python runShell.py — 80x24" showing the output of running "python3 runShell.py". The output includes version information, project name, and various configuration parameters.

```
MBP-di-Pietro:SLAPP pt$ python3 runShell.py

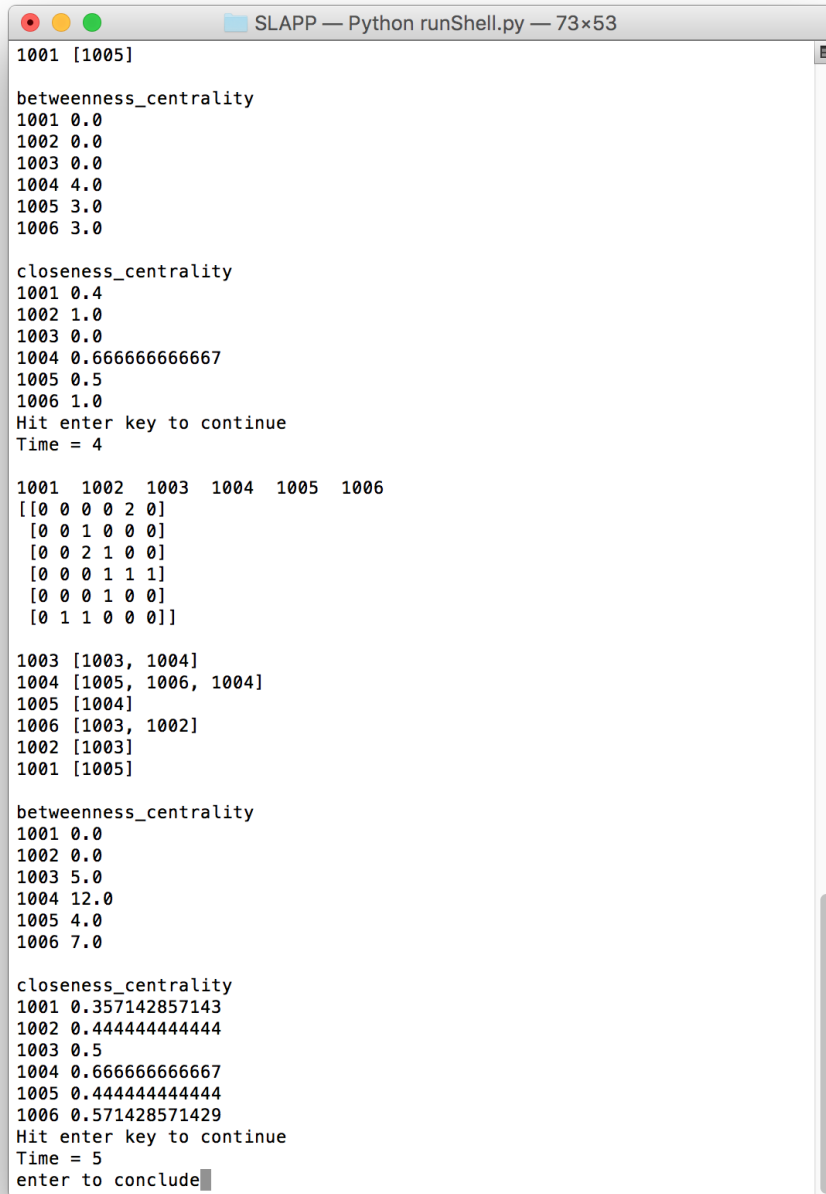
SLAPP v.3.0 build 20170513

Project name? production
SLAPP started from a terminal
running in Python
debug = False
NetworkX version 1.11 running
Matplotlib version 2.0.0 running

random number seed (1 to get it from the clock) 1
No 'bland' agents
X size of the world not relevant
y size of the world not relevant
recipes: max lenght 10 and max sector number 6
How many cycles? (0 = exit) 4
verbose? (y/[n])
```

Figure 1.6: Starting the *production* project

The effect is (plain text output):



```
1001 [1005]

betweenness centrality
1001 0.0
1002 0.0
1003 0.0
1004 4.0
1005 3.0
1006 3.0

closeness centrality
1001 0.4
1002 1.0
1003 0.0
1004 0.666666666667
1005 0.5
1006 1.0
Hit enter key to continue
Time = 4

1001 1002 1003 1004 1005 1006
[[0 0 0 0 2 0]
 [0 0 1 0 0 0]
 [0 0 2 1 0 0]
 [0 0 0 1 1 1]
 [0 0 0 1 0 0]
 [0 1 1 0 0 0]]

1003 [1003, 1004]
1004 [1005, 1006, 1004]
1005 [1004]
1006 [1003, 1002]
1002 [1003]
1001 [1005]

betweenness centrality
1001 0.0
1002 0.0
1003 5.0
1004 12.0
1005 4.0
1006 7.0

closeness centrality
1001 0.357142857143
1002 0.444444444444
1003 0.5
1004 0.666666666667
1005 0.444444444444
1006 0.571428571429
Hit enter key to continue
Time = 5
enter to conclude
```

Figure 1.7: The plain text output of the *production* project

and as graphical output:

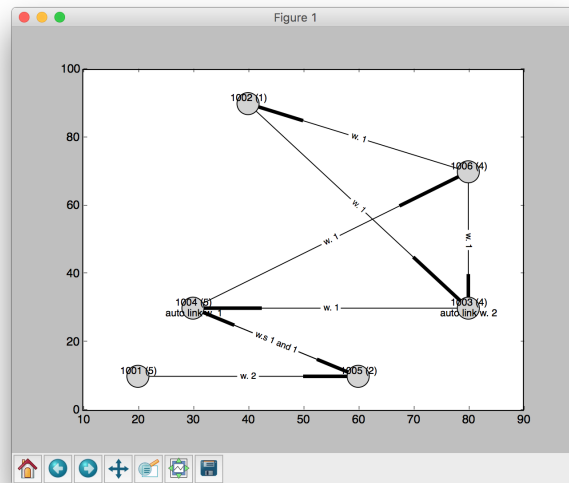


Figure 1.8: The graphical output of the *production* project

- 4 - If you prefer to work with Python in a notebook—using (i) the *IPython*<sup>11</sup> interactive version of Python or, better, (ii) the “agnostic language shell” named *Jupyter*<sup>12</sup>—via a terminal go into the main SLAPP folder (that where you have unzipped SLAPP) and launch Jupyter as in Figure 1.9:

---

<sup>11</sup><http://ipython.org>

<sup>12</sup><http://jupyter.org>





Figure 1.9: Launching *Jupyter* from the SLAPP folder

then chose `iRunShell.ipynb` as in Figure 1.10:

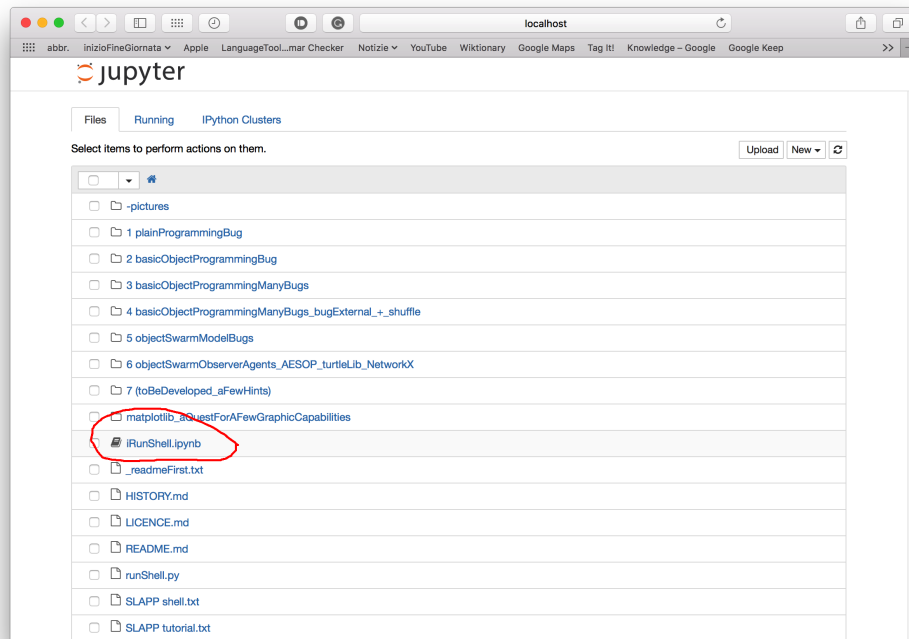
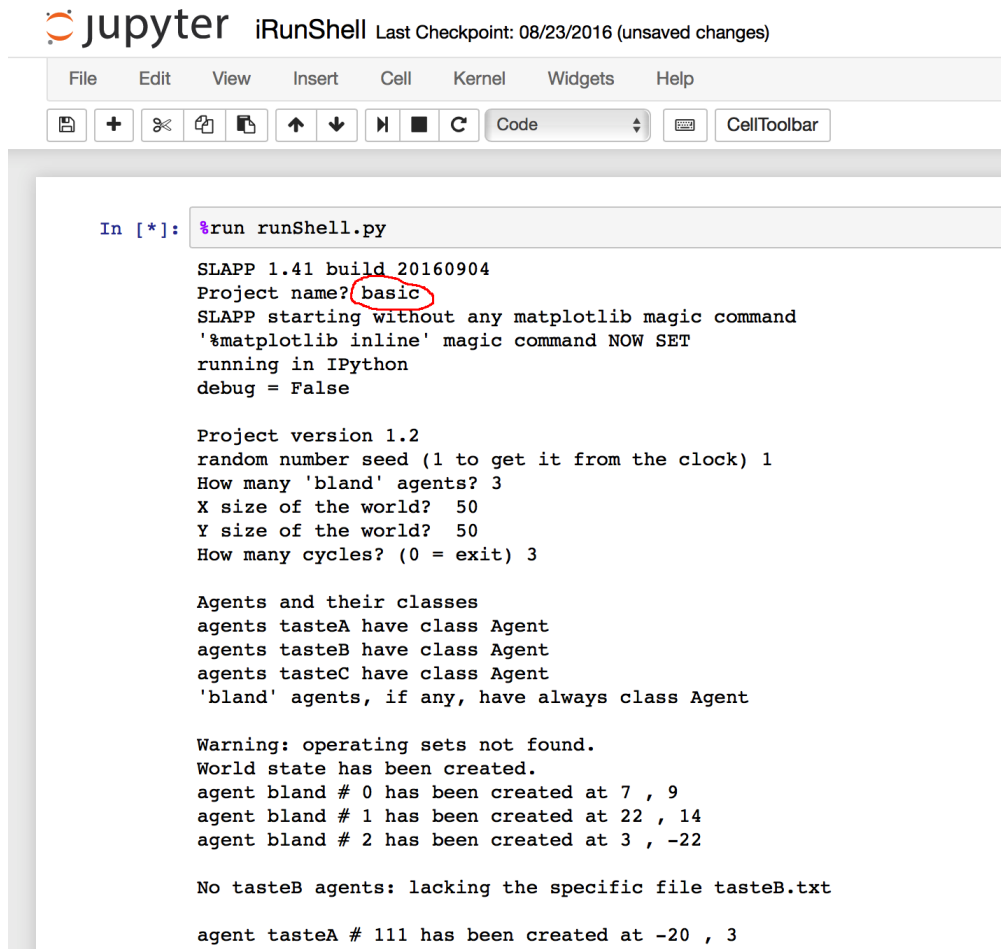


Figure 1.10: *Jupyter* main window

and finally chose the project and run it as in Figure 1.11:



The screenshot shows a JupyterLab interface. At the top, the Jupyter logo is followed by 'iRunShell' and 'Last Checkpoint: 08/23/2016 (unsaved changes)'. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Under the 'Cell' menu, there are icons for saving, adding, deleting, and running cells, along with a 'Code' dropdown and a 'CellToolbar' button. The main area contains a code cell with the prompt 'In [\*]:' followed by the command '%run runShell.py'. The output of this command is displayed below the cell. The output text includes version information for SLAPP, project name 'basic' (circled in red), startup messages, random seed, world size, number of agents, and agent creation details.

```
In [*]: %run runShell.py

SLAPP 1.41 build 20160904
Project name? basic
SLAPP starting without any matplotlib magic command
'%matplotlib inline' magic command NOW SET
running in IPython
debug = False

Project version 1.2
random number seed (1 to get it from the clock) 1
How many 'bland' agents? 3
X size of the world? 50
Y size of the world? 50
How many cycles? (0 = exit) 3

Agents and their classes
agents tasteA have class Agent
agents tasteB have class Agent
agents tasteC have class Agent
'bland' agents, if any, have always class Agent

Warning: operating sets not found.
World state has been created.
agent bland # 0 has been created at 7 , 9
agent bland # 1 has been created at 22 , 14
agent bland # 2 has been created at 3 , -22

No tasteB agents: lacking the specific file tasteB.txt

agent tasteA # 111 has been created at -20 , 3
```

Figure 1.11: The *basic* project running in *Jupyter*

## Chapter 2

# The *basic* project as a guide to the making of a new project

To let you familiarize with SLAPP we introduce the `basic` project (in Section 1.4.2 you had a view of it).

### 2.1 How to *run* SLAPP

The starting phase is introduced in the next *bullet*, in a detailed way, via the `basic` example.

- In the SLAPP distribution, we have the `basic` folder<sup>13</sup>, containing an introductory application.
  - We can launch the SLAPP simulation shell—via the `runShell.py` file that we find in the main folder of SLAPP—from a terminal, with<sup>14</sup>:  
`python3 runShell.py`
  - Alternatively, we can launch SLAPP as a simulation shell—via the `start.py` file that we find in the folder of SLAPP, i.e. `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`—from a terminal, with:  
`python3 start.py`  
The last is a possible way, but deprecated.
  - It is not possible to run `runShell.py` or `start.py` via the Python dedicated shells such as *IDLE* or *Spyder*, ... Instead, it is possible to use *Spyder* to run *start.py* or *runShell.py* in IPython.

---

<sup>13</sup>Within the `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX` folder.

<sup>14</sup>Usually, `python` is for Python 2 and `python3` for Python 3.

- To use SLAPP in IPython (in a `jupyter notebook`), go to the main folder of SLAPP via a terminal and then start  
`jupyter notebook`  
and finally click on `iRunShell.ipynb`.
- Alternatively again, we can also run SLAPP in IPython via the Spyder environment, executing `%run start.py` or `%run runShell.py`, going to their folder with `%cd` followed by the path to the folder.
- We can also launch SLAPP from a Jupiter QtConsole—e.g. Anaconda launcher—executing `%run start.py` or `%run runShell.py`, going to their folder with `%cd` followed by the path to the folder.
- A further possibility is that of launching IPython from a terminal with `ipython` command line, being in the SLAPP directory, and then executing `%run runShell.py`. In this case the graphic results will be the same of the execution from a terminal using Python.

In all cases, we immediately receive the request of choosing a project:  
`Project name?`

- In this example we reply `basic` (or `school` or `production`, for the other examples). If we want to create a new project, we simply add a new folder; the folder name will also be automatically that of the project, and we will choose it at the prompt above.

We also have a special folder, named `$$slapp$$`,<sup>15</sup> that the user is not supposed to modify. It is the folder where we store the kernel of SLAPP, i.e., its simulation engine. If you do not modify it, always building your applications in a separate folder, your work will not be affected by the modifications introduced by the new versions of SLAPP.

- We can set a default project: if we place *in the main SLAPP folder or in the folder* `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX` a file named `project.txt` containing the path to a folder (`basicTmp` as an example, so the content of the file could be something as `/Users/pt/Desktop/basicTmp`<sup>16</sup>), and the initial message of SLAPP will be:

```
path and project = /Users/pt/Desktop/basicTmp
do you confirm? ([y]/n):
```

---

<sup>15</sup>Always within `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.

<sup>16</sup>In Windows it would be better to use backslashes “\” instead of slashes “/”. Anyway (verified in Windows 10) also slashes work.

The feature is useful in two perspectives: (i) we can place our projects outside the SLAPP folder; (ii) we can avoid typing the name of the project when, in the debugging phase, we launch it a lot of times. The file `project.txt` can also contain a relative path, e.g., `./basic`; the starting directory is that of the `start.py` file.

For advanced users: we can access the position of the *project* via the variable `project` automatically appearing in `commonVar.py`. So, somewhere in the code:

```
import commonVar as common
... = common.project
...
```

`common.project` above contains (i) or the path (relative to the `start.py` position) of a project existing within the SLAPP hierarchy, or (ii) the absolute path to a project placed outside.

- Resuming the explanation: now we are looking at the message:

```
SLAPP started from a terminal
running in Python
debug = False
```

```
Project version 1.2
random number seed (1 to get it from the clock)
```

The `Project version` message is implemented as a suggestion only in the `basic` project, specifying the version of the project into the file `commonVar.py` and managing it via the file `parameters.py`; both the files are in the project folder.

To reply to the open question about the random seed, we have to enter an integer number (positive or negative; 0 is a valid input) to trigger the sequence of the random numbers used internally by the simulation code. If we reply 1, the seed—used to start the generation of the random series—comes from the internal value of the clock at that instant of time. So it is different anytime we start a simulation. This kind of reply is useful to repeat the simulated experiments with different conditions. If we chose a number different from 1, the same random sequence would be repeated anytime we will use that seed. This solution is useful (1) while debugging, when we need to repeat exactly the sequence generating some error, but also (2) to give to the user the possibility of replicating exactly an experiment.

The `running in Python` sentence signals the we are running the program in plain Python. Alternatively, the message could be `running in IPython`, as explained in Chapter 6.

The `SLAPP started from a terminal` signals this info relatively to the graphical behavior of SLAPP. For a technical explanation look at the content of the folder `matplotlib_aQuestForAFewGraphicCapabilities` in the main SLAPP folder.

- Then the code asks us to enter the number of unspecified agents; this is related to the AESOP (Agents and Emergencies for Simulating Organizations in Python) perspective, introduced below as an abstract layer upon SLAPP. There we have both well-defined agents (*tasty*) and unspecified ones (*bland*).

How many 'bland' agents?

Finally, after a few information, we have to enter the number of the cycles we want:

```
X size of the world? 50
Y size of the world? 50
How many cycles? (0 = exit)
```

Replying 3 as the number of bland agents and 3 as the number of cycles, we obtain the output reported (only the final part) in Figure 1.2. The random seed, as in Figure 1.1, was 3.

## 2.2 Scheduling

We introduce now time management, split into several (consistent) levels of scheduling.

The general picture is that of Figure 2.1: in an abstract way we can imagine to have a clock opening a series of containers or boxes. Behind the metaphor of the boxes, in SLAPP, as it was in Swarm, we have the *action groups*, where we store the information about the actions to be done.<sup>17</sup>

Imagining the events as objects, in the object-oriented programming perspective, is one of the key points of success in the original Swarm system. We implement the same idea in SLAPP.

---

<sup>17</sup>The structure is highly dynamical because we can associate a probability to an event, or an agent of the simulation can be programmed to add or eliminate one or more events into the *boxes* or, better, into the *action groups*.

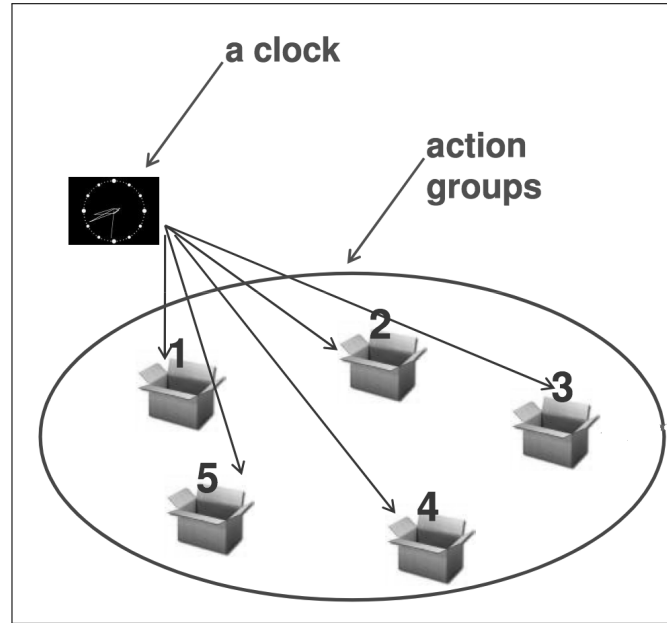


Figure 2.1: The representation of the schedule

In SLAPP, we have the following three schedule mechanisms, or processes, driving the events.

- The first mechanism is at the level of the Observer (Section 2.2.1) and the second one at the level of the Model (Section 2.2.2), both with recurrent sequences of action to be done.<sup>18</sup> We will introduce the third mechanism, more detailed, in Section 2.2.3.
- In our basic code, these sequences are reported in the files `observerActions.txt` and `modelActions.txt` in folder `basic`.<sup>19</sup>

## 2.2.1 The scheduling mechanism at the level of the Observer

<sup>18</sup>The level of the Observer is our level, where the experimenter looks at the model (the level of the Model) while it runs. This structure is a key feature in Swarm, and so we reproduce it in SLAPP. Other simulation shells follows the same scheme: as an example, the observer is a key feature in NetLogo <https://ccl.northwestern.edu/netlogo/>.

<sup>19</sup>Within folder 6 *objectSwarmObserverAgents\_AESOP\_turtleLib\_NetworkX*.



- To discover the first schedule mechanism, we refer to the first file (`observerActions.txt`), containing (row changes are not relevant):

```
modelStep ask_all clock
modelStep ask_one clock
modelStep ask_one clock
```

The interpretation is the following.

- First of all, we have to take into consideration that the execution of the content of the file is “with repetition”, until an **end** item will appear (see page 24). If we do not need differentiations within the repetition cycle, also a content as the following should work:

```
modelStep ask_all clock
```

However, the content can be as articulated as we need.

- **modelStep** orders to the model to make a step in time. The order acts via the code of the file `ObserverSwarm.py`<sup>20</sup>, where we have (example *i*) a simple rule ordering to the Model code to make a step.
- **ask\_all** orders to all the agents to talk. In this case, always in `ObserverSwarm.py`, we have (example *ii*) one of the four stable instances<sup>21</sup> of the class `ActionGroup` within the Observer. That related to **ask\_all** contains the `do2a` variable, linking a method which is specified as a function in the file `oActions.py` of folder `basic`. In this way, the application of the basic method **ask\_all** can be flexibly tailored to the specific applications.
- **clock** ask the clock to increase its counter of one unit. When the count will reach the value we have entered replying to the **How many cycles?** query, the `ActionGroup` instance (example *iii*) related to the clock (`actionGroup1` in `ObserverSwarm.py`) will add the **end** item into the sequence of the file `observerActions.txt`. The item is placed immediately after the **clock** call. The **end** item stops the sequence contained in the file.

It is possible to automatically start a concluding sequence after **end**, if we declare a string of Python instructions in a quoted way, as content of the `toBeExecuted` variable to be added in the `commonVar.py` file.

---

<sup>20</sup>Which is in the “`$$$lapp$$`” folder (see above in this Chapter).

<sup>21</sup>The instances of the class `ActionGroup` contained in the file `ActionGroup.py` in folder “`$$$lapp$$`” are related to: “clock”; “visualizeNet”, used with network analysis; “ask\_all”; and “ask\_one”.

Have a look to the example in the `basic` project, producing the final Goodbye message.

For advanced users:

- \* in a similar way, as we can see in project `basicSpecialAction`, we can declare a string of Python instructions in a quoted way, as content of the `specialAction` variable in the `commonVar.py` file;
- \* in the `basicSpecialAction` project, the content of that variable is `"makeSpecialAction()"`; we have to develop this function into the file `oActions.py` (we can change the name, but the modification is related to the content both of the variable `basicSpecialAction` and of the function in `oActions.py`);
- \* in this example, look at `observerAction.txt`; running the project for at least 3 cycles, we see the effect of the `specialAction` at time 1 or 2, not successively; indeed, in `oActions.py` the code is:

```
# an example of special action code, to be activated if the time
# (cycle) is equal to ...

# to pass variables to the function, simply use the common area
def makeSpecialAction():

    if common.cycle == 1:
        print "doing a special action at time =", common.cycle

    if common.cycle == 2:
        print "doing a special action at time =", common.cycle

    # etc.
```

- `ask_one` orders to the first component of the agent collection to talk. As above (example *ii*, being this the example *iv*), we have an instance of the class `ActionGroup` within the `Observer`. That related to `ask_one` contains the `do2b` variable, linking a method which is specified as a function in the file `oActions.py` in the folder `basic`. In this way, the application of the basic method `ask_one` can be flexibly tailored to the specific applications.

- It is useful to underline that the example (*i*) has no reference in the file `oActions.py`. We can add similar items for the scheduling, directly “wiring” them via the function

```
def otherSubSteps(subStep, address):
```

in `oActions.py`, without modifying

`ObserverSwarm.py` in `$$slapp$$`

(look at the `production` project to see how, with `pause` and `prune`).

- The examples (*ii*), (*iii*), and (*iv*) use the double structure of the instance of the class `ActionGroup` and of the related method<sup>22</sup> construction that we have

---

<sup>22</sup>Technically, our *pseudo*-methods—that we pass to the instance via a variable—are always

in `ObserverSwarm.py` (in `$$$slapp$$`), with the definition in `oActions.py` of the folder `basic` (in our current case). It is a more complicated structure, but very flexible.

- Looking at the `oActions.py` files of the other projects (currently, `debug`, `basic2classes`, `basicSpecialActions`, `school` and `production`<sup>23</sup>), you can analyze the different ways of using the options (i), (ii), (iii), and (iv).
- If we use a missing keyword in the files collecting the first two levels of scheduling, i.e. `observerActions.txt` or `modelActions.txt`—maybe in error or referring to a not yet implemented item—we receive a *warning*. See: `Warning: step ask_on not found in Observer` where the item `ask_one` is misspelled,

### 2.2.2 The scheduling mechanism at the level of the Model

- The file `modelActions.txt`, quoted above at the beginning of Section 2.2, is related to the second schedule mechanism: that of the Model. (About the Observer/Model dualism, the reference is to note 18.)

The file contains (unique row, remembering that row changes are not relevant in this group of files):

```
reset move read_script
```

The interpretation is the following.

- Also at the Model level, we have to take into consideration that the execution of the content of the file is “with repetition”, never ending. It is the Observer that stops the experiment, operating at its level.
- `reset` orders to the agents to make a reset, related to their variables. The variables can be specified as explained in the next few rows. The order acts via the code of the file `ModelSwarm.py`<sup>24</sup>. In this case, always in `ModelSwarm.py`, we have (example *I*) one of the three stable instances<sup>25</sup> of the class `ActionGroup` within the Model.

---

functions. So, we have to manage explicitly the value of the usual *self* value. To avoid any possible confusion, the term used in these cases—into the SLAPP code—is *address*.

<sup>23</sup> *debug* and, most of all, *basic2classes* and *basicSpecialActions*, are introducing technical details.

<sup>24</sup> That is in the “`$$$slapp$$`” folder (see above in this Chapter).

<sup>25</sup> The instances of the class *ActionGroup* contained in the file *ActionGroup.py* in folder `$$$slapp$$` are related to: “reset”; “move”; and “read\_script”

The item `reset` contains the `do0` variable, linking a method that is specified as a function in the file `mActions.py` in the folder `basic`. The application of the basic method `reset` can so be flexibly tailored to the specific applications, defining which are the variables we are resetting.

In our specific case, the content of the `do0` function in `mActions.py` asks all the agents to execute the method `setNewCycleValues`. The method is defined in an instrumental file (`agTools.py` in `$$$slapp$$$`) and it is, as default, doing nothing. We can redefine it in `Agent.py` in the project folder. Into the `basic` project, `reset` is not operating, but it is reported above as a memo for future uses.

The case is strictly similar to the examples *ii*, and subsequent ones, introduced above (Section 2.2.1).

- `move` orders to the agents to move. The order acts via the code of file `ModelSwarm.py`. We have here (example *II*) the second of the three stable instances of the class `ActionGroup` within the `Model`. That related to `move` contains the `do1` variable, linking a method that is specified as a function in the file `mActions.py` in the folder `basic`. In this way, the application of the basic method `move` can be flexibly tailored to the specific applications, defining what kind of movement (if any) we order to the agents.

In our specific case, the content of the `do1` function in `mActions.py` asks all the agents to execute the method `randomMovement`. We defined that method in the file `Agent.py`, in the project folder.

The case is strictly similar to the examples *ii*, and subsequent ones, introduced above (Section 2.2.1).

The structure managing the movement is quite complicated, just to propose a not trivial example.

The Python code (in `mActions.py`) determining the movement is:

```
askEachAgentInCollectionAndExecLocalCode \  
    (address.agentListCopy, Agent.randomMovement,  
     jump=random.uniform(0,5))
```

A few details:

- \* `address` substitutes the implicit usual *self* as explained above in Section 2.2.1;
- \* `agentListCopy` is a shuffled copy to ask the agent to move in an ever-changing sequence;

\* `Agent.randomMovement` is the address (within the class) of the method that we send to the agent list;  
an example helps to clarify (we are here using Python interactively, in a shell):

```
>>> class A:
    def __init__(self,b):
        self.b=b
    def prnt(self):
        print self.b
>>> a=A(10)
>>> aa=A(100)
>>> a.prnt()
10
>>> aa.prnt()
100
>>> A.prnt
<unbound method A.prnt>
>>> A.prnt(a)
10
>>> A.prnt(aa)
100
```

\* `jump=random.uniform(0,5)` is optional; if it is placed there, it assigns a random value to a dictionary key named *jump*.

The method `randomMovement`, reported in `Agent.py` in folder `basic` (this example), is defined with an optional<sup>26</sup> dictionary in the head, as:

```
def randomMovement(self,**k):
```

---

<sup>26</sup> The optional dictionary works as in the following example (created interactively in a Python shell):

```
>>>class A:
    def test(self,**k):
        self.b=1
        if k.has_key('b'):self.b=k['b']
>>> a=A()
>>> a.test()
>>> a.b
1
>>> a.test(b=10)
>>> a.b
10
```

The call to the method assigns a default value to the key `jump`; the method verifies its existence;

```
self.jump=1
    if k.has_key("jump"): self.jump=k["jump"]
```

The value of the `jump` multiplies the length of the movement.

- `read_script` orders to the Model to open a new level of scheduling, described in Section 2.2.3. The order acts via the code of file `ModelSwarm.py`. We have here (example *III*) the third of the stable instances of the class `ActionGroup` within the Model. The `ActionGroup` related to `read_script` item is the `actionGroup100` that contains the `do100` function, used internally within `ModelSwarm.py` to manage the script reported into the `schedule.xls` file (or directly into the `schedule.txt` one).

- We have also the feature of the *local code execution*.

We use the function `setLocalCode` of `Tools.py` (in `$$$slapp$$$`) to define a code to be executed “on the fly” via the function `askEachAgentInCollectionAndExecLocalCode` of `Tools.py`. The function simply executes:

```
exec(localCode)
```

having received the code to be executed (if you want to replicate this kind of code implementation, have a look both to the function and to its internal links). In this way, we have a flexible and powerful way for adding activities in our agents.

We can employ the *local code execution* also within the agents’ methods used in the third scheduling mechanism described in Section 2.2.3.

- The cases (*I*), (*II*) shown above are conceptually similar to the cases (*ii*) and subsequent ones, seen above (Section 2.2.1); instead, the case (*III*) is quite special.

We can also have schedule structures as the *i* above (always Section 2.2.1, adding the code after

```
def otherSubSteps(subStep, address):
    in mActions.py.
```

We have an example of this solution in the project `production`.

### 2.2.3 The detailed scheduling mechanism within the Model (AESOP level), with *WorldState* and *macros*

- The third scheduling mechanism, as anticipated in Section 2.2, is based on a detailed script system that the Model executes while the time is running. The time is managed by the `clock` item in the sequence of the Observer.<sup>27</sup> The script system is activated by the item `read_script` in the sequence of the Model.
- This kind of script system does not exist in Swarm, so it is a specific feature of SLAPP, introduced as implementation of the AESOP (Agents and Emergencies for Simulating Organizations in Python) idea: a layer that describes in a fine-grained way the actions of the agents in our simulation models.
- Let us deepen the scheduling hierarchy, with the three levels:
  - at the Observer level (via the file `observerActions.txt`) we run a high level sequence of events: (a) one of the events is the request to the Model of making a step (`modelstep`) and (b) another is the request to the `clock` to go by;
  - at the Model level (via the file `modelActions.txt`) we run an intermediate level sequence of events (a sub-cycle within the previous one): one of the events is now the request (`readScript`) to the fine-grained scripting system (if any) to execute the *action container* (see below, next bullet point) related to the time step we are in;
  - at the AESOP level (i.e. within the Model detailed scripting system) we activate the set of rules and actions introduced in this Section.
- At the AESOP layer, the *action containers* are specified—with the `#` indicators (see below)—upon the time; we put them into a spreadsheet describing both the acting agents and their actions.

We adopt the spreadsheet formalism because it is well known and diffused, but you can bypass it creating directly a text file containing the same elements, as explained below.

Other details are in the files

SLAPP 6 `objectSwarmObserverAgents.txt`

and

---

<sup>27</sup>Increasing the time counter of one unit for each call to the *clock* item.

a\_note\_on\_AESOP.txt

in the usual folder

6 objectSwarmObserverAgents\_AESOP\_turtleLib\_NetworkX.

- Now let us have a close look to an example of AESOP layer schedule (the third layer), i.e. the timetable where we describe minutely the actions that the agents are doing at each time step.
- The file `schedule.xls` can be composed of several sheets, with: (a) the first one with the mandatory name `schedule`; (b) the other ones with any name (those names are *macro* names.)<sup>28</sup> We can recall the macro instructions in any sheet, but not within the sheet that creates the macro (that one with the same name of the macro), to avoid infinite loops.

We start with the sheet in `scheduleBase.xls` of folder `basic`. To use a sheet, you have to rename it to `schedule.xls` (keep safe the original file).

Within the sheet, we have the *action containers* as introduce above, starting with the sign #.

In `scheduleBase.xls` we have (comments start at column E):

			COL. E
			comments here or in successive columns
#	1		standard (background) actions, like move, are applied to "all"
bland	eat		bland agents are those not specified in dedicated .txt files,
bland	dance		with the related names reported in the agTypeFile.txt file
#	2		
#	4		
all	0.5	dance	all agents acting
tasteC	eat		tasteC agents acting
#	5		
all	eat		
all	dance		
#	7		
tasteA	0.5	dance	tasteA agents acting
#	8		
tasteB	dance		tasteB agents acting (no agents of this type exist here)

- In column A, we can place: (i) the sign # or (ii) the word `macro` or (iii) a *name* identifying a group of agents (the number of the agents in the group can vary from 1 to any dimension; about the groups, see Section 2.3.1):
  - with # (action container), we state that, when the clock reaches the time (in units of the conventional internal clock of the model) specified in column B, SLAPP will execute the content of the rows<sup>29</sup> following that containing the # sign and until the next similar sign;

---

<sup>28</sup>We deeply use *macros* within the *school* project.

<sup>29</sup>Maybe none, or: (i) one or more, empty; (ii) one or more, with operating contents.



- with a *macro* name in column A, we indicate that at a given time, SLAPP will activate the set of instructions reported in a specific sheet; see below for macros;
- with a *name* identifying a set of agents<sup>30</sup> in column A, we send to this/these agent/s the method set in column B, as an action to be done in a deterministic way; if in column B we have a number, it represents the probability (upon 1) of execution of the method reported, in this case, in column C; the probability can be close to 0, but always  $\geq 0$ :<sup>31</sup>  
<sup>32</sup>
  - \* the probability can be interpreted both as: (a) the share of the set of agents—recalled in column A—requested to act; or (b)—which is quite the same—as the probability of each individual of set of executing the action;
  - \* if the number in column B is both less than "0" and integer, exactly that number (multiplied times  $-1$  to have it as positive) of agents is asked to execute the actions; the agents are randomly extracted from the list. We use this feature both in the project **school** and in the project **production**.
- The containers of action identified by the # sign can be also introduced in a nonsequential way into each sheet of the spreadsheet. If we repeat the same “# *time*” sequence in the same sheet, only the last one is considered. The # sign can be employed into the macro sheets (also repeating a “# *time*” sequence existing in another sheet); when the macro is called, the content of its time blocks is properly placed in the related time steps, orderly (e.g., a block coming from the third sheet will be placed after a block coming from the second and before a block coming from the fourth sheet).
- *Time loops*: we can also manage time loops using a block such as

```
#      1      3
somethingA
somethingB
```

that will be internally transformed to

---

<sup>30</sup>Both coming from the `agTypeFile.txt` list or from the `agOperatingSets.txt` one; in this example, we do not have operating sets.

<sup>31</sup>This is due to the internal use of the  $\theta$  value to discriminate cases in which the second column directly contains a method.

<sup>32</sup>SLAPP displays—into the text output—a dictionary of the methods and of their probabilities, if at least one method has a linked probability.

```
#      1
somethingA
somethingB
#      2
somethingA
somethingB
#      3
somethingA
somethingB
```

It is not possible to insert the *time loops* feature operating into the `schedule.txt` file. Sure, we can repeat block using copy and paste and modifying by hand the variable parts.

- We also have a more complicated schedule in the file `scheduleBaseWithMacros_WorldState.xls` (to be copied to `schedule.txt` for the use), where we employ both the *WorldState* feature and the *macros*.

### 2.2.3.1 The *WorldState* feature as a variable repository

Running a project, at the beginning of the output, we read:

```
World state has been created.
```

What does it mean?

The `WorldState` class interacts with the agents; we use a unique instance of that class.

The variables managed via `WorldState` have to be added, with their methods, within the instance, following the existing example in project 'basic', where `WorldState` has set/get methods for the variable `generalMovingProb`.

In `Agent.py` of `basic` the method `randomMovement` asks to the `WorldState` the probability threshold, to be compared with a random value to decide to move. By construction (in `WorldState.py` into the `basic` project) the default threshold is 1 (move always); if we modify it to 0.1, as in the example below, movements will be a lot less frequent. The probability can also be 0.

This use of *WorldState* has in Col. B a value and in Col. C the method used to set that value into the WorldState; it will be retrieved by the agents using a symmetric get method.<sup>33</sup>

In `scheduleBaseWithMacros_WorldState.xls` we have:

– in sheet *schedule* (comments not reported here)

```
# 1
bland eat
bland dance
# 2
# 4
all 0.5 dance
tasteC eat
WorldState 0.1 setGeneralMovingProb
# 5
all eat
all dance
# 7
tasteA 0.5 dance
# 8
tasteB dance
```

```
macro repeat
```

```
macro dancing
```

– in sheet *dancing*

```
# 9
all dance
# 10
tasteC dance
```

– in sheet *repeat*

```
# 5 10
tasteA eat
```

### 2.2.3.2 The *WorldState* feature as a computational tool

If in Col. B we have the expression `computationalUse`<sup>34</sup>, the content of Col. C is a special method making *world calculations*.

You can find significant examples of the computational use of the *WorldState* feature (with their code) within the project *oligopoly* at <https://github.com/terna/oligopoly>.

### 2.2.3.3 The *macros*

- In the example we also have the use of *macros* (as for *time loops*, macros cannot be programmed directly in the file `schedule.txt`).

---

<sup>33</sup>These methods have to be implemented by the user, see the example in the *basic* project.

<sup>34</sup>the expression *specialUse* is still working, but it is deprecated.

In the example above we have two macros, defined in the sheets **dancing** and **repeat**. The effect in file **schedule.txt** follows. (NB, the call to a macro can be usefully repeated, mainly if the macro has no time reference via **#** sign).

- The content of the file **schedule.txt** with the effects of the macros:

```
# 1
bland eat
bland dance
# 2
# 4
all 0.5 dance
tasteC eat
WorldState 0.1 setGeneralMovingProb
# 5
all eat
all dance
tasteA eat
# 6
tasteA eat
# 7
tasteA 0.5 dance
tasteA eat
# 8
tasteB dance
tasteA eat
# 9
tasteA eat
all dance
# 10
tasteC dance
```

#### 2.2.3.4 The *if* structure

- The **if** structure can be easily implemented, as in the **school** project, where the file **scheduleIf.xls** (in sheet **checkToObtainAttention**) contains the row:

```
saPupil shakeIf_greenPupil
```

The method **shakeIf\_greenPupil** is developed in **Agent.py** in folder **school**. It orders to the agents of type **saPupil** (a unique one, in our case) to shake, but only if at least one of the agents of the group **greenPupil** has been shaking (verified checking their last executed method).

#### 2.2.3.5 Agents adding and eliminating tasks into the detailed schedule

- The agents' capability of adding and eliminating tasks into the detailed schedule of the file **schedule.xls** can be useful to add a high dynamic

perspective to the sequence of the events

Look at the content of the file `scheduleAddElim.xls`<sup>35</sup> within the `basic` project.

The differences with `scheduleBase.xls` are visible thanks this sequence of `\\` signs, which are not part of the file.

```

COL. E

#          1          comments here or in successive columns
bland  eat          standard (background) actions, like move, are applied to "all"
bland  dance        bland agents are those not specified in dedicated .txt files,
                    with the related names reported in the agTypeFile.txt file

tasteA -1          addTask \\adding a task

#          2

all -1          elimTask \\eliminating a task

#          4
all  0.5  dance  all agents acting
tasteC eat          tasteC agents acting
#          5
all  eat
all  dance
#          7
tasteA 0.5  dance  tasteA agents acting
#          8
tasteB dance          tasteB agents acting (no agents of this type exist here)

```

- With “`tasteA -1 addTask`” we order to exactly one of the agents of type `tasteA` to add, at time  $t+1$ , the item “`all dance`” (look at the file `Agent.py` within the `basic` project. This is an unconditional action, but it is quite easy to imagine to introduce probabilities or *if* structures into the method `addTask` of `Agent.py`, as well as different time delays, also in a dynamic way.
- With “`all -1 elimTask`” we order to exactly one of the agents of type `all` to eliminate, at time  $t+2$ , the item “`tasteC eat`” (look at the file `Agent.py` within the `basic` project. This is an unconditional action, but again it is quite easy to imagine to introduce probabilities or *if* structures into the method `elimTask` of `Agent.py`, as well as different time delays, also in a dynamic way.
- You can see the effect into the simulation output below, having the adding or eliminating steps producing: (i) the immediate outputs signed with `<=====` and (ii) the delayed effects identified with `<====`.

SLAPP 3.0 build 20170513  
Project name? basic

---

<sup>35</sup>Rename it *schedule.xls* if you want use it.

```
SLAPP started from a terminal
running in Python
debug = False

Project version 1.2
random number seed (1 to get it from the clock) 3
How many 'bland' agents? 3
X size of the world? 50
Y size of the world? 50
How many cycles? (0 = exit) 5
Warning: operating sets not found.
World state has been created.
agent bland # 0 has been created at -14 , 2
agent bland # 1 has been created at -7 , 5
agent bland # 2 has been created at 6 , -22

No tasteB agents: lacking the specific file tasteB.txt

agent tasteA # 111 has been created at -25 , 16
agent tasteA # 222 has been created at -13 , -14
agent tasteC # 1111 has been created at 24 , -2

#### Project ./basic starting.

Time = 1
agent tasteA # 111 moving
agent bland # 1 moving
agent bland # 0 moving
agent tasteA # 222 moving
agent bland # 2 moving
agent tasteC # 1111 moving
methodProbabilities = {'dance': 0.5}
I'm bland agent # 1:  nothing to eat here!
I'm bland agent # 0:  nothing to eat here!
I'm bland agent # 2:  nothing to eat here!
I'm bland agent # 1:  it's not time to dance!
I'm bland agent # 2:  it's not time to dance!
I'm bland agent # 0:  it's not time to dance!
agent 222 adding a task for cycle 2                                <=====
Time = 1 ask all agents to report position
bland agent # 0  is at X = -18.3402265357  Y =  6.34022653572
bland agent # 1  is at X = -2.65977346428  Y =  5.0
bland agent # 2  is at X = 10.3402265357  Y = -22.0
tasteA agent # 111 is at X = -20.6597734643  Y = 20.3402265357
tasteA agent # 222 is at X = -8.65977346428  Y = -9.65977346428
tasteC agent # 1111 is at X = 24.0  Y = 2.34022653572
Time = 2
agent tasteC # 1111 moving
agent bland # 0 moving
agent tasteA # 222 moving
agent bland # 2 moving
agent bland # 1 moving
agent tasteA # 111 moving
I'm bland agent # 1:  it's not time to dance!                    <====
I'm tasteA agent # 111:  I'm an A, nice to dance here!          <====
I'm bland agent # 2:  it's not time to dance!                    <====
I'm bland agent # 0:  it's not time to dance!                    <====
I'm tasteA agent # 222:  I'm an A, nice to dance here!          <====
I'm tasteC agent # 1111:  I'm a C, why to dance here?           <====
agent 0 eliminating a task for cycle 4                            <=====
Time = 2 ask first agent to report position
bland agent # 0  is at X = -15.4148559987  Y =  9.26559707274
```

```
Time = 3
agent bland # 1 moving
agent tasteA # 222 moving
agent bland # 0 moving
agent tasteA # 111 moving
agent bland # 2 moving
agent tasteC # 1111 moving
Time = 3 ask first agent to report position
bland agent # 0 is at X = -18.4136700427 Y = 12.2644111167
Time = 4
agent bland # 1 moving
agent tasteC # 1111 moving
agent bland # 0 moving
agent tasteA # 222 moving
agent bland # 2 moving
agent tasteA # 111 moving
I'm bland agent # 2: it's not time to dance!
I'm bland agent # 0: it's not time to dance!
I'm tasteA agent # 111: I'm an A, nice to dance here!
I'm bland agent # 1: it's not time to dance!
I'm tasteC agent # 1111: I'm a C, why to dance here?
tasteC modified to dummy in: ['tasteC', 'eat'] <====
agent dummy does not exist <====
Time = 4 ask all agents to report position
bland agent # 0 is at X = -18.4136700427 Y = 7.37542453448
bland agent # 1 is at X = -8.58395804529 Y = 3.03638395478
bland agent # 2 is at X = 18.154583655 Y = -17.1110134178
tasteA agent # 111 is at X = -25 Y = 22.3038425809
tasteA agent # 222 is at X = -14.5839580453 Y = -4.844230389
tasteC agent # 1111 is at X = 24.0 Y = 3.37542453448
Time = 5
agent bland # 1 moving
agent tasteC # 1111 moving
agent tasteA # 111 moving
agent bland # 2 moving
agent bland # 0 moving
agent tasteA # 222 moving
I'm bland agent # 1: nothing to eat here!
I'm tasteA agent # 222: nothing to eat here!
I'm bland agent # 2: nothing to eat here!
I'm tasteC agent # 1111: nothing to eat here!
I'm bland agent # 0: nothing to eat here!
I'm tasteA agent # 111: nothing to eat here!
I'm tasteC agent # 1111: I'm a C, why to dance here?
I'm bland agent # 0: it's not time to dance!
I'm tasteA agent # 111: I'm an A, nice to dance here!
I'm bland agent # 2: it's not time to dance!
I'm tasteA agent # 222: I'm an A, nice to dance here!
I'm bland agent # 1: it's not time to dance!
Time = 5 ask first agent to report position
bland agent # 0 is at X = -18.4136700427 Y = 8.48811374845
Time = 6
enter to conclude
Goodbye
```

## 2.3 The agents and their sets

We have files containing the agents of the different types. These files are listed in a master file, with name `agTypeFile.txt`: in our case, it simply contains the record `tasteA tasteB tasteC` (the names are just an example).

- `tasteA.txt` lists the agents of type (“taste”) A; in our case it reports only the identification numbers:

```
111
222
```

- `tasteC.txt` lists the unique agent of type (“taste”) C, with the identifying number:

```
1111
```

- `tasteB` agent are missing, so we have no file `tasteB.txt`; lacking the file, we receive the message:

```
No tasteB agents: lacking the specific file tasteB.txt
```

The presence in the folder of an empty `tasteB.txt` file (zero bytes or containing a few spaces), would eliminate the message above. The SLAPP program raises no errors in the execution in any case.

The agents are created by `ModelSwarm.py` (in folder `$$$slapp$$$`) via the application specialized file `mActions.py` (in folder `basic`).

```
def createTheAgent(self,line,num,agType):
    #explicitly pass self, here we use a function

    if len(line.split())==1: # weak control, can be improved
        anAgent = Agent(num, self.worldState,
                        random.randint(self.leftX,self.rightX),
                        random.randint(self.bottomY,self.topY),
                        self.leftX,self.rightX,self.bottomY,self.topY,agType=agType)
        self.agentList.append(anAgent)

    else:
        print "Error in file "+agType+".txt"
        os.sys.exit(1)
```

*self*, in the code above, is the address of `ModelSwarm.py`, where we define the variables that here we address with the prefix *self*.

Each project has an analogue structure dedicated to its agents. The following bullets describe how this code works.



- As an ex-ante information, the identifying number of the agent is read outside this function, as a mandatory first element in the lines of any file containing agent descriptions. Also the content of the `agType` variable is coming from outside, being the name of the agent file currently open.
- We check the input file, which—in the case of the project *basic*—has to contain a unique datum per row.

Other projects can have several data in each row, related to multiple attributes of each agent. <sup>36</sup>

Each agent is added to the `agentList`.

### 2.3.1 Sets of agents

The files containing the agents are of two families:

- files listing the agents with their characteristics (if any): in folder **basic** we have the files `tasteA.txt` and `tasteC.txt`;
- files defining groups of agents:
  - the list of the types of agents (mandatory); from this list SLAPP searches for the file describing the agents (first bullet here above); in folder **basic** we have the file `agTypeFile.txt` (also the name of this file is mandatory), containing:

```
tasteA tasteB tasteC
```

- the list of the operating sets of agents (optional); in folder **basic** this file is missing. Indeed we receive the message  
`Warning: operating sets not found.`  
In **school** project we have the file `agOperatingSets.txt` (the name of this file is mandatory), with content:

```
threeGreen leftS rightS r1l r2l r3l r1r r2r r3r r4r  
1Row cZone 1Pupil 2Pupil 3Pupil 4Pupil 5Pupil  
6Pupil 7Pupil 8Pupil 9Pupil 10Pupil 11Pupil  
12Pupil 13Pupil 14Pupil 15Pupil 16Pupil 17Pupil  
18Pupil 19Pupil 20Pupil 21Pupil
```

---

<sup>36</sup>The files defining each set of agents can also have the extension `.txtx`. In case, they will be translated in regular `.txt` file, as explained in Section 2.3.2.

All the names contained in the file are related to other `.txt` files reporting the identifiers of agents specified in the lists of the previous bullet. The goal of this feature is that of managing clusters of agents, recalling them as names in Col. A in Section 2.2.3.

### 2.3.2 The use of files `.txtx` to define the agents

The files with extension `.txtx` (*txt eXtended*) are used to define the agents in a flexible way.

An example is reported in the `basic` project where the `tasteA.txt` file contains:

```
111
222
```

and generates the taste A agents with id (number) 111 and 222.

Redefining `tasteA.txtx_` to `tasteA.txtx` the mechanism activated by the files `.txtx` operates. The file `tasteA.txtx` contains:

```
111@120
```

and produces a file `tasteA.txt` containing:

```
111
112
113
114
115
116
117
118
119
120
```

(to roll back copy `tasteA.txt_` to `tasteA.txt`). Definitions:

- $n$  and  $v$  are mandatory names;
- $n$  is the value in first position of the record (cannot be a formula);
- $v$  is the result of the calculation in a formula;
- `&` starts and concludes a formula.

We can have more than one formula in a row of the `.txtx` file. Typical complex rows (with the extension `.txtx` as *eXtended txt*) are:

```
1@3 1 2 &if n==1:v=100#else:v=10& 3
```

with effect:

```
1 1 2 100 3
2 1 2 10 3
3 1 2 10 3
```

or (with the same effect):

```
1@3 1 2 &if n==1:v=100#if n>1:v=10& 3
```

or

```
1@3 &v=100*n& 3
```

```
1@3 1 2 3
```

The sign ;—as separator of instruction in the same row—is quite complicated to be used within an `if` construct, so we use `#` as special sign, internally translated to `\n` (change row sign).

Details (NB, *a* and *b* are only used for the examples):

```
exec("a=2; if a< 3: print 'phew'") rises and error
exec("a=2\nif a< 3: print 'phew'") works
exec("a=2\n if a< 3: print 'phew'") rises and error again
```

The sign ; is not forbidden, but we suggested to use it (carefully) to build blocks after an `if` structure; e.g.:

```
exec("a=2\nif a<2:b=11;print b\nelse: b=22;print b")
```

### 2.3.3 Future developments about agents

Until v.1.36 we had a unique `Agent` class for each project, containing heterogeneous methods addressed to different types of agents. The agent types are specified as in Section 2.3.1, so all is working, but from the coding point of view, this construction is a bit annoying.

Look at project "production" to see how we logically subdivide the unique class of the agents.

SLAPP 1.36 and most of all 1.4 introduces the possibility of having any number of classes for the agents, with methods used: (a) in the AESOP mechanism (`read_script` execution), but also (b) at the higher level of Model scheduling, when a function like `askEachAgentInCollection` (of `Tools.py`) is internally used maybe in `mActions.py`. For details look at Section 3.

## Chapter 3

# SLAPP multi-class: the *basic2classes* example

### 3.1 Introducing the example

The project `basic2classes` shows how to use the multi-class capability of SLAPP.<sup>37</sup>

- The file `agClassFile.txt` reports the links between the names of each type of instance and its class; in this example, the names are the same of the project `basic` plus a new one, which has a class on its own (`OtherAgent`):

```
tasteA Agent
tasteB Agent
tasteC Agent
tasteD OtherAgent
```

We make automatically a consistency control between the contents of `agClassFile.txt` and `agTypeFile.txt`.

We create, as above, two agents of type `tasteD` via the file `tasteD.txt`, containing:

```
11111
22222
```

- To create, in `ModelActions.py`, the new kind of agents introduced in this Section, having class different from `Agent`, we call the general function

---

<sup>37</sup>If you are not aware of classes and how use them in Python, have a look to <https://docs.python.org/2/tutorial/classes.html>.

`createTheAgent_Class(self, line, num, agType, agClass)`, implemented in `mActions.py` of the specific project we use, as in the example of the project `basic2classes`.

The parameters `line`, `num`, `agType`, `agClass` are fixed. We can recall other parameters (e.g., those related to the world limits if we are considering the space or the address of the meta-agent `WorldState`) prefixing *self*—which is here a pointer to `ModelSwarm.py` in `$$$slapp$$` folder—to their names. Have a look to the examples of the projects included into the distribution of SLAPP.

`line` is a special parameter containing the current line coming from the file `.txt` in which we define the agents of the class under construction.

- All the class of the agents have to inherit from the class *Agent*; *Agent* inherits from *SuperAgent*, which inherits from *object*.<sup>38</sup> Bland agents automatically inherits from the *Agent* class.
- We can place into the *Agent* class all the common methods; if necessary, we can redefine them into the inhering classes.
- In the example presented here we have agents created directly as instances of the *Agent* class and agents created as instances of the *otherAgent* class; this situation is related to the example: we can have all the agents derived from classes different from the *Agent* one. As stated above, anyway all inheriting from that class.

N.B. to simplify the structure of SLAPP, the name of a class and the name of the file containing it have to be the same.

The example is based on the *schedule* below, derived from the *base schedule* of the project `basic`, modified adding `all sleep` after `# 2` and `tasted sleep` after `# 3`.

The method `sleep` is defined only in the class `OtherAgent`. In `OtherAgent` is also redefined the method `randomMove`, with the agents refusing to move!

```
COL. E
      comments here or in successive columns
#      1      standard (background) actions, like move, are applied to "all"
bland  eat    bland agents are those not specified in dedicated .txt files,
bland  dance  with the related names reported in the agTypeFile.txt file
#      2
all    sleep
#      3
tasted sleep
```

---

<sup>38</sup> *WolrdState* inherits directly from *object*.

```
#      4
all    0.5    dance    all agents acting
tasteC eat          tasteC agents acting
#      5
all    eat
all    dance
#      7
tasteA 0.5    dance    tasteA agents acting
#      8
tasteB dance          tasteB agents acting (no agents of this type exist here)
```

Starting the project, we have chosen 3 as seed of the random sequence and declared 3 as number of *bland* agents.

The results specifically related to the `sleep` method), at time 2 and 3 are:

```
Time = 2
...
Warning, class Agent (or above) of agent tasteA does not have the method sleep
Warning, class Agent (or above) of agent bland does not have the method sleep
Warning, class Agent (or above) of agent tasteA does not have the method sleep
I'm tasteD agent # 22222: happy to sleep!
Warning, class Agent (or above) of agent bland does not have the method sleep
I'm tasteD agent # 11111: happy to sleep!
Warning, class Agent (or above) of agent tasteC does not have the method sleep
Warning, class Agent (or above) of agent bland does not have the method sleep
...
Time = 3
...
I'm tasteD agent # 11111: happy to sleep!
I'm tasteD agent # 22222: happy to sleep!
...
```

At  $t = 2$  all the agents are reacting, both signaling that their class is lacking of the method `sleep` or replying correctly.

At  $t = 3$  only the `tasteD` agents are activated and reply in the due way.

Looking at the effect of the order move in `modelActions.txt`, activating the method `randomMove` (defined in class *Agent*, bur redefined in class *OtherAgents*), we see below that at  $t = 1$  (and the same for successive steps) `TasteD` agents react in a special way.

```
Time = 1
agent tasteA # 222 moving
agent bland # 1 moving
agent bland # 0 moving
agent bland # 2 moving
I'm tasteD agent # 22222: absolutely not moving!!!
agent tasteA # 111 moving
I'm tasteD agent # 11111: absolutely not moving!!!
agent tasteC # 1111 moving
```

## Chapter 4

# Debugging a new project: the *debug* project as a micro tutorial

Running the project `debug`<sup>39</sup> with the option `debug` set to `False` in its `commonVar.py` file or missing at all, we have:

```
(...)
Project name? debug
debug = False
(...)

Time = 1
I'm bland agent # 2:  clock is at  1
I'm bland agent # 0:  clock is at  1
I'm bland agent # 1:  clock is at  1
I'm A agent # 11:   clock is at  1
I'm A agent # 22:   clock is at  1
Time = 2
Warning, method checkCloc does not exist in class Agent      <=====
Time = 3
I'm bland agent # 1:  clock is at  cannot apply (case 0) method checkSuperClock to agent
                                number 1 of type  bland
I'm bland agent # 0:  clock is at  cannot apply (case 0) method checkSuperClock to agent
                                number 0 of type  bland
I'm A agent # 11:   clock is at  cannot apply (case 0) method checkSuperClock to agent
                                number 11 of type  A
I'm bland agent # 2:  clock is at  cannot apply (case 0) method checkSuperClock to agent
                                number 2 of type  bland
I'm A agent # 22:   clock is at  cannot apply (case 0) method checkSuperClock to agent
                                number 22 of type  A
Time = 4
enter to conclude
```

`checkCloc` effectively does not exist as a method (it is a typo in `schedule.xls`); the warning message perfectly identifies the problem; the program is anyway running.

Using SLAPP for Python 3 the message about `checkCloc` is more elaborated:

---

<sup>39</sup> *debug* project is a clone (with different methods in *Agent* class) of the *basic* one.

```
Warning, class Agent (or above) of agent bland does not have the method checkCloc
Warning, class Agent (or above) of agent A does not have the method checkCloc
Warning, class Agent (or above) of agent bland does not have the method checkCloc
Warning, class Agent (or above) of agent bland does not have the method checkCloc
Warning, class Agent (or above) of agent A does not have the method checkCloc
```

Then we have several message declaring that it is impossible to apply the method `checkSuperClock` to a few agents; the attempt of communicating that the “clock is at” fails.

But why? The SLAPP attempt of capturing the errors, is hiding here the Python messages explaining what is happening.

Correct the typo writing correctly `checkClock`; then set `debug` to `True` in `commonVar.py` and run again.

We have:

```
(...)
Project name? debug
debug = True
(...)

Time = 1
I'm A agent # 11: clock is at 1
I'm A agent # 22: clock is at 1
I'm bland agent # 0: clock is at 1
Time = 2
I'm A agent # 11: clock is at 2
I'm bland agent # 0: clock is at 2
I'm A agent # 22: clock is at 2
Time = 3
I'm bland agent # 0: clock is at
Traceback (most recent call last):
  File "start.py", line 39, in <module>
    observerSwarm.run()
  File "./$slapp$/ObserverSwarm.py", line 103, in run
    self.modelSwarm.step(common.cycle)
  File "./$slapp$/ModelSwarm.py", line 296, in step
    self.actionGroup100.do(self,cycle)
  File "./$slapp$/ModelSwarm.py", line 215, in do100
    self.applyFromSchedule(localList,task)
  File "./$slapp$/ModelSwarm.py", line 334, in applyFromSchedule
    if common.debug: exec "askEachAgentInCollection(localList,Agent"+"."+task[1]+" )"
  File "<string>", line 1, in <module>
  File "./$slapp$/Tools.py", line 38, in askEachAgentInCollection
    if common.debug: method(a,**k)
  File "./debug/Agent.py", line 46, in checkSuperClock
    print "clock is at ", common.cycles
AttributeError: 'module' object has no attribute 'cycles'    <=====
```

We discover that the error is due to the use of the attribute *cycles* supposed to be in `common`, while the correct name is *cycles*.

In the same way, setting `debug=True` in `commonVar.py` of any project, we can have the Python syntax emerging in case of coding errors. The misuse of method etc. is always signaled by SLAPP.



## Chapter 5

# Other existing and upcoming projects

The project `oligopoly` will be included in the SLAPP distribution. Temporary look at it at <https://github.com/terna/oligopoly>.

### 5.1 Adding *turtles*: the *school* project

We add here turtle graphical capabilities. TO BE DEVELOPED.

About the name (turtle), have a look at Appendix [D](#).

### 5.2 Adding networks: the *production* project

TO BE DEVELOPED.

### 5.3 New projects and extensions

#### 5.3.1 Connecting to R, via Rserve

TO BE DEVELOPED.

#### 5.3.2 Connecting to other applications, via Redis

TO BE DEVELOPED.

Redis is at <http://redis.io>.

We can—as an example—connect a SLAPP model to a NetLogo one (NetLogo address in Appendix [D](#)).

### 5.3.3 Parallel computations in SLAPP

We plan to introduce parallel computation capabilities in SLAPP relatively to the class of problems defined as *embarrassingly parallel*.<sup>40</sup>

The plan is of exploring the use of *ipyparallel* project.<sup>41</sup>

---

<sup>40</sup>[https://en.wikipedia.org/wiki/Embarrassingly\\_parallel](https://en.wikipedia.org/wiki/Embarrassingly_parallel).

<sup>41</sup><https://github.com/ipython/ipyparallel>.

## Chapter 6

# SLAPP in IPython/Jupyter

### 6.1 Running SLAPP in an IPython/Jupyter notebook

SLAPP runs in IPython.

To use it as a notebook go to the main SLAPP folder via a terminal, then start:

```
jupyter notebook
```

and load in `jupyter` the file `iRunShell.ipynb`.

We do not explain here how to install Jupiter (<https://jupyter.org>), but a short cut is

```
pip install jupyter
```

or

```
sudo pip install jupyter
```

Look at the contents of Appendix [A](#) about the use of `pip`.

As noted in opening this Reference, we have also a [running version of SLAPP](<https://beta.mybinder.org/>) via the wonderful [Binder project](<https://beta.mybinder.org/>): try it!

The module `graphicControl.py` in the `$$$slapp$$` folder operates the identification of the environment used to run SLAPP and of the related characteristics.

### 6.2 Size or the pictures in IPython/Jupyter notebook

The *width* and the *height* of the graphic pictures within an IPython notebook is defined in the module `graphicControl.py` (see page [50](#) and can be redefined

within the `commonVat.py` module of each application with:

`width = a value`

`height = a value`

The values are in inches, but ... on paper and on the screen the effect is related to the screen and printer pixel density.

The *width/height* suggested ratio is 3/2.

## 6.3 *Turtle* graphics and IPython

Turtle graphics does not work in an IPython notebook (maybe in the future, existing several projects in that direction); the turtle display is generate outside the notebook. To run SLAPP on line, a possible solution is that of opening a VNC<sup>42</sup> connection parallel to the notebook interaction. More to come.

---

<sup>42</sup>[https://en.wikipedia.org/wiki/Virtual\\_Network\\_Computing](https://en.wikipedia.org/wiki/Virtual_Network_Computing).

# Appendices

# Appendix A

## Libraries for SLAPP

To use SLAPP you need to install a few Python libraries.

An easy way to have anything installed at once is the Anaconda Scientific Python distribution. You can find it at <https://store.continuum.io/cshop/anaconda/>, with clear installing instructions. Anaconda contains installers for Python 2.x and 3.x: for SLAPP chose Python 3.x.

After the installation, your environment variable (PATH in Mac OS and Linux; PATH o path in Windows)<sup>43</sup> will contain the information to use Python and IPython from the **anaconda** folder (usually in the user home) and its subfolders.

If you do not want to use the Anaconda distribution, the *do it yourself* way is feasible.

---

<sup>43</sup>It is possible to see the content of the path from the *Terminal* (*Command Prompt* or *Windows PowerShell* in Windows) with:

*echo \$PATH*

in Linux/Mac OS terminal

*set path*

in *Command Prompt* of Windows

*\$env:Path*

in *Windows PowerShell* of Windows.

It is highly useful to familiarize with the Unix-like commands of the Linux/Mac OS Terminal and Windows PowerShell, e.g., at [https://en.m.wikipedia.org/w/index.php?title=Command-line\\_interface&redirect=no](https://en.m.wikipedia.org/w/index.php?title=Command-line_interface&redirect=no) and with the DOS-like commands of Command Prompt of Windows, e.g., at <http://pcsupport.about.com/od/termssc/p/command-prompt.htm>.

## A.1 Using Linux (via the Ubuntu distribution)

- Verify the Python version in your system (with `python --version`) and upgrade it if not recent (in the series of the version 3.x, at least 3.6.x).

A simple way to install Python from the terminal, is (sudo requires your password)<sup>44</sup>

```
sudo apt-get update
```

to update the list of the packages, then

```
sudo apt-get install python3
```

to upgrade (or to install, if Python is not there)

- In the Ubuntu distribution, sometimes Python is installed without the *Tkinter* module. *Tkinter* is required by *matplotlib*, see below. To check your system, start Python and order `import tkinter`,<sup>45</sup> if the reply signal the library as missing, install it with:

```
sudo apt-get install python3-tk
```

- If the program `pip` (Python Package Index) is not installed (try `pip` in the terminal), run (always in the terminal)

```
sudo apt-get install python-pip
```

From now on, `pip3` could be required instead of `pip`. Check with

```
which pip
```

to discover if in the resulting path you have a python3 three or not. If not, use `pip3`.

- Install the `xlrd`<sup>46</sup> library to read spreadsheet files (`.xls` extension) in Python, via terminal with  

```
sudo pip install xlrd
```
- Until here, we have been copying the requirements of file `WARNING.txt` of the folder 6 `objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.
- The tools above are sufficient to run the `basic` example, having no graphic output, or the `school` project, which is entirely based on the graphical capabilities of the Python `turtle` library (installed with Python). About *turtles* see the Appendix D.

---

<sup>44</sup>This note is related to Ubuntu distribution, but do not use strictly the Ubuntu Software Center, mainly to be a bit more general; anyway, if you use other Linux distributions, you are certainly cleverer than the author of this Reference Handbook.

<sup>45</sup>NB: for Python 3 we write *tkinter* with small “t”.

<sup>46</sup><https://github.com/python-excel/xlrd>

- If you want run the project `production`, graphically displaying networks, or the `oligopoly` project <sup>47</sup> the reference is `WARNING bis - Production required libraries.txt` in the same folder above.
- Before installing `matplotlib`, it is useful to install `scipy` via Ubuntu Software Center or via command line with  
`sudo apt-get install python3-scipy`  
In this way you have also `numpy` installed (`numpy` is required by `matplotlib`).
- Install `matplotlib` (<http://matplotlib.org>) via Ubuntu Software Center or via terminal with  
`sudo pip install matplotlib`
- Install `NetworkX` (<https://networkx.github.io>) with  
`sudo pip install networkx`
- Install <sup>48</sup> `pandas` (<http://pandas.pydata.org>) with  
`sudo pip install pandas`

## A.2 Using Mac OS X

- Verify the Python version in your system (with `python --version`) and upgrade it if not recent (in the series of the version 3.x at least 3.6.x).  
To install Python download the installer from <https://www.python.org>; with the current 3.6.1 version, the installer file is `python-3.6.1-macosx10.6.pkg`. Run it.
- `pip` (Python Package Index) is coming with recent versions of Python; anyway, upgrade it via terminal with  
`sudo pip install --upgrade pip`  
(root user password required).  
From now on, `pip3` could be required instead of `pip`. Check with  
`which pip`  
to discover if in the resulting path you have a python3 three or not. If not, use `pip3`.

---

<sup>47</sup><https://github.com/terna/oligopoly>

<sup>48</sup>For the *oligopoly* project or in any case in which you need a database within SLAPP.



- Install the `xlrd`<sup>49</sup> library to read spreadsheet files (`.xls` extension) in Python, via terminal with  
`sudo pip install xlrd`  
(root user password required).
- Until here, we have been copying the requirements of file `WARNING.txt` of the folder 6 `objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.
- The tools above are sufficient to run the `basic` example, having no graphic output, or the `school` project, which is entirely based on the graphical capabilities of the Python `turtle` library (installed with Python). About *turtles* see the Appendix D.
- If you want run the project `production`, graphically displaying networks, or the `oligopoly` project<sup>50</sup> the reference is `WARNING bis - Production required libraries.txt` in the same folder above.
- Before installing `matplotlib`, it is useful to install `scipy`, via terminal with:  
`sudo pip install scipy`  
(root user password required). In this way you have also `numpy` installed (`numpy` is required by `matplotlib`).
- Install `matplotlib` (<http://matplotlib.org>), via terminal with  
`sudo pip install matplotlib`  
(root user password required).
- Install `NetworkX` (<https://networkx.github.io>) with  
`sudo pip install networkx`  
(root user password required).
- Install<sup>51</sup> `pandas` (<http://pandas.pydata.org>) with  
`sudo pip install pandas`  
(root user password required).
- If you have IDLE or tinter troubles in Mac, carefully analyze <https://www.python.org/download/mac/tcltk/>.

---

<sup>49</sup><https://github.com/python-excel/xlrd>

<sup>50</sup><https://github.com/terna/oligopoly>

<sup>51</sup>For the *oligopoly* project or in any case in which you need a database within SLAPP.

## A.3 Using Windows (referring to Windows 10)

We refer here to Windows 10, but the following notes work also for the versions 7, 8, 8.1 (always supposing a 64 bits system).

Use the **Command Prompt** or the **Windows PowerShell**, introduced in note 43 above.

- Python 3.x, if installed, is in `C:\Python3x.y\`
- Verify the Python version in your system (with `python --version`) and upgrade it if not recent (in the series of the version 3.x at least 3.6.x).

From <https://www.python.org/> you can download an installer; e.g. for version 3.6.6 on a 64 bits system: `python-3.6.6.amd64.msi`. Run the file clicking on it.

If you run `python` from a terminal (**Command Prompt** or **Windows PowerShell**) and the reply is that the program does not exist. You have to run (substitute *x* below bwith the appropriate number):

```
\python3x\python
```

because the *path* of your system does not contemplate that folder as a repository for programs. As an example, it could be (using **Command Prompt**; for **Windows PowerShell** use `$env:Path` instead of `set path`):

```
>set path
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
C:\Windows\System32\WindowsPowerShell\v1.0\
```

You have to modify the *environment variables*: from **Settings** go to **System**, then to **About**, scroll down to find **System Info**, then proceed choosing **Advanced System Settings**, press the **Environment Variables** button. In **System variables**, chose **Path**, then **Edit** and add at the end of the path the string (pay attention to the initial semicolon):

```
;c:\Python27;c:\Python27\Scripts\.
```

Restart the terminal you were using to apply the new settings and the `python` command will work.

Now you have:

```
>set path
Path=C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
C:\Windows\System32\WindowsPowerShell\v1.0\;c:\Python3x\;
c:\Python3x\Scripts\
```

- `pip` (Python Package Index) is coming with recent versions of Python; anyway, upgrade it via terminal with  
`pip install --upgrade pip`  
From now on, `pip3` could be required instead of `pip`. Check (in the Power Shell) with  
`which pip`  
to discover if in the resulting path you have a python3 three or not. If not, use `pip3`.
- Install the `xlrd`<sup>52</sup> library to read spreadsheet files (`.xls` extension) in Python, via terminal with  
`pip install xlrd`
- Until here, we have been copying the requirements of file `WARNING.txt` of the folder `6 objectSwarmObserverAgents_AESOP_turtleLib_NetworkX`.
- The tools above are sufficient to run the `basic` example, having no graphic output, or the `school` project, which is entirely based on the graphical capabilities of the Python `turtle` library (installed with Python). About *turtles* see the Appendix [D](#).
- If you want run the project `production`, graphically displaying networks, the reference is `WARNING bis - Production required libraries.txt` in the same folder above.
- Before installing `matplotlib`, it is useful to install `scipy`. Very useful information are at <https://www.scipy.org/install.html>.  
In this way you have also `numpy` installed (`numpy` is required by `matplotlib`).  
If you want to avoid the `scipy` installation, you can install uniquely `numpy`:  
`pip install numpy`
- Install `matplotlib` (<http://matplotlib.org>), via terminal with  
`pip install matplotlib`
- Install `NetworkX` (<https://networkx.github.io>) with  
`pip install networkx`
- Install<sup>53</sup> `pandas` (<http://pandas.pydata.org>) with  
`pip install pandas`

---

<sup>52</sup><https://github.com/python-excel/xlrd>

<sup>53</sup>For the *oligopoly* project or in any case in which you need a database within SLAPP.

## Appendix B

### On SLAPP execution

SLAPP runs only via a terminal or in IPython (`jupyter notebook`), using `runShell.py` or `iRunShell.ipynb`.

In IPython, the magic command `%matplotlib inline` is internally added if missing; if `%matplotlib` is the explicit choice, the `inline` option is internally stated.

In the main folder now we have `runShell.py` to start the shell in Python and `iRunShell.ipynb` to start it in IPython (using `jupyter notebook`).

We do not explain here how to install Jupiter (<https://jupyter.org>), but a short cut is

```
pip install jupyter
```

or

```
sudo pip install jupyter
```

Look at the contents of Appendix [A](#) about the use of `pip`.

We stop the execution if starting from IDLE or Spyder, for compatibility with the graphic operations.<sup>54</sup>

---

<sup>54</sup>At <http://matplotlib.org/users/shell.html> we read “the python IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend”.

## Appendix C

### Problems with libraries

#### C.1 A warning about fonts coming from *matplotlib* 1.5.1

`matplotlib` produces an annoying warning about creating fonts; to avoid it.

Several hints online suggest to delete the folders `fontconfig` or `matplotlib` that you can find in folder `.cache` within your home.

Instead, in MacOS go to the folder `.matplotlib` in your home and delete the file `fontList.cache`.

The annoying warning will appear only one more time.

## Appendix D

### On *turtles*

The `turtle` library mimics the behavior both of NetLogo,<sup>55</sup> of OpenStarLogo,<sup>56</sup> and (partially) of StarLogo TNG<sup>57</sup> agent-based shells. The name *turtle* attributed to the agents in those shells (and in the Python related library) comes from Logo, a special language of the 1960s. At [http://el.media.mit.edu/logo-foundation/what\\_is\\_logo/logo\\_primer.html](http://el.media.mit.edu/logo-foundation/what_is_logo/logo_primer.html) we read that:

The most popular Logo environment has involved the Turtle, originally a robotic creature that moved around on the floor.

It can be directed by typing commands at the computer. The command forward 100 causes the turtle to move forward in a straight line 100 "turtle steps". Right 45 rotates the turtle 45 degrees clockwise while leaving it in the same place on the floor. Then forward 50 causes it to go forward 50 steps in the new direction.

With just the two commands forward and right, the turtle can be moved in any path across the floor. The turtle also has a pen which may be lowered to the floor so that a trace is left of where it has traveled. With the pen down, the turtle can draw geometric shapes, and pictures, and designs of all sorts.

(. . .)

The turtle migrated to the computer screen where it lives as a graphics object. Viewing the screen is like looking down on the mechanical turtle from above.

But . . . why the name *turtle*? In Epstein (2014, p.88) we have a nice and openly subjective explanation:

---

<sup>55</sup><https://ccl.northwestern.edu/netlogo/>

<sup>56</sup><http://web.mit.edu/mitstep/openstarlogo/index.html>

<sup>57</sup>[http://education.mit.edu/portfolio\\_page/starlogo-tng/](http://education.mit.edu/portfolio_page/starlogo-tng/)

*NetLogo*'s name for a generic agent is “turtle”. I choose to imagine that this is in honor of a famous exchange between Bertrand Russel and an audience member who told Russel that the earth was supported on the neck of a great turtle. Russel asked, ‘And what, pray tell, is supporting *that* turtle?’ The answer was immediate. “Oh, another turtle ...it’s turtles all the way down.”

My humble explanation is less fascinating: when I was told about Logo for the first time, in the 1970s, they explained me that the original robot-agent was named *turtle* ... because it was slowly moving and, above all, because—being Logo aimed to interact with boys and girls of the primary school—the name turtle was less intimidating than robot.

Anyway, what is crucial is that NetLogo and StartLogo TNG (deriving from what now is named OpenStarLogo) have their roots in Logo and turtles (in Fig. D.1 the *logo* of the Logo Foundation).



Figure D.1: The Logo Foundation, at <http://el.media.mit.edu/logo-foundation/>

# Bibliography

Boero, R., Morini, M., Sonnessa, M. and Terna, P. (2015). *Agent-based Models of the Economy Agent-based Models of the Economy – From Theories to Applications*. Palgrave Macmillan, Houndmills.

URL <http://www.palgrave.com/page/detail/agentbased-models-of-the-economy-/?K=9781137339805>

Downey, A. B. (2012). *Think Python*. How to Think Like a Computer Scientist. O'Reilly Media, Inc., Sebastopol, CA.

URL <http://www.greenteapress.com/thinkpython/>

Elkner, J., Downey, A. B. and Meyers, C. (2013). *Learning with Python: Interactive Edition 2.0*. How to Think Like a Computer Scientist. Runestone Interactive.

URL [http://interactivepython.org/runestone/default/user/login?\\_next=/runestone/default/index](http://interactivepython.org/runestone/default/user/login?_next=/runestone/default/index)

Epstein, J. M. (2014). *Agent\_Zero: Toward Neurocognitive Foundations for Generative Social Science*. Princeton University Press.

Minar, N., Burkhart, R., Langton, C. and Askenazi, M. (1996). *The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations*. In «SFI Working Paper», vol. 06(42).

URL <http://www.santafe.edu/media/workingpapers/96-06-042.pdf>

Sargent, T. and Stachurski, J. (2013). *Quantitative Economics*.

URL <http://quant-econ.net>



# Index

- .txtx files, 41
- \$\$slapp\$\$, 20
- action container, 30, 31
- adding or eliminating tasks, 35
- AESOP, 30
- agent creation, 39
- Anaconda, 53
- basicSpecialAction, 25
- classes in SLAPP, 43
- computationalUse in world state, 34
- debug, 46
- embarrassingly parallel problems, 49
- end, 24
- files .txtx, 41
- Future developments about agents, 42
- if structure, 35
- installing SLAPP, 53
- IPython, 50
- IPython notebook, 50
- jupyter notebook, 50
- Libraries for SLAPP, 53
- Libraries to use SLAPP, 4
- Linux, 54
- local code execution, 29
- Mac OS X, 55
- macros, 33, 34
- matplotlib 1.5.1 warning (fonts), 60
- Model, 23
- multi-class, 43
- Observer, 23
- oligopoly, 48
- online, 5
- operating sets of agents, 40
- parallel computations, 49
- parameters in classes, 44
- predefining a default project, 20
- problems with libraries, 60
- project reference, advanced use, 21
- running SLAPP, 19
- schedule, 23, 24, 26
- schedule.xls, 31
- scheduling hierarchy, 30
- set of agents, 40
- setting action probabilities, 32
- simulation engine, 20
- size of the graphic pictures in IPython notebook, 50
- Size or the pictures in IPython notebook, 50
- SLAPP execution, 59
- specialAction, 25
- specialUse in world state, 34
- spreadsheet formalism, 30
- starting SLAPP from a Jupiter QtConsole, 20
- starting SLAPP from a terminal with IPython, 20

starting SLAPP from a terminal with  
    jupyter notebook, [20](#)  
starting SLAPP from a terminal with  
    Python, [19](#)  
starting SLAPP from Spyder using the  
    IPython console, [20](#)  
Swarm, [5](#), [6](#), [22](#), [23](#), [30](#)  
swarmapps original file, [6](#)  
  
time loops, [32](#)  
toBeExecuted, [24](#)  
turtle graphics and IPython, [51](#)  
turtles, [61](#)  
types of agents, [40](#)  
  
Ubuntu, [54](#)  
  
width and the height of the graphic pic-  
    tures in IPython notebook, [50](#)  
Windows, [57](#)  
world state, [33](#)  
WorldState, [33](#)