Simulating Language: Lab 2 Worksheet
Code Walkthrough

This document gives a line-by-line walkthrough of the code in the first file we looked at (signalling1.py), which measures the communicative accuracy between a production and reception system.

Data Structures: a signalling matrix represented as a list of lists

A production system can be thought of as a matrix which maps meanings to signals. We are representing this as a list. Each member of the list is itself a list containing the association strengths for *one particular meaning.* In the example here, a production system called **`psys`** is defined: it has three members, representing the three meanings. The length of the system is equivalent to the number of meanings in the system. `psys[0]` contains the association strengths for the meaning **m1**, `psys[1]` contains the association strengths for the meaning **m2**, and so on (remember that indexes in Python start from zero!). Each of these sub-lists has three members, representing the three possible signals. So `psys[0][0]` is the strength of association between meaning **m1** and signal **s1**.

```
>>> psys = [[1, 0, 0],[1, 2, 1],[3, 4, 4]]
>>> len(psys)
3
>>> psys[0]
[1, 0, 0]
>>> psys[0][0]
1
```

We can do the same thing to model a reception system, but in this case we are dealing with a system which maps from signals to meanings: so if `rsys` is a reception system then each member of `rsys` is itself a list which contains the association strength between a signal and three meanings.

```
>>> rsys = [[0, 0, 1],[0, 1, 0],[3, 1, 2]]
>>> len(rsys)
3
>>> rsys[2]
[3, 1, 2]
>>> rsys[2][1]
1
```

Create a variable containing the following production matrix:

	s_1	s_2	s_3
m_1	1	0	2
m_2	2	2	0
m_3	0	1	3

Print the weights for meaning m1
Print the weight of the connection between meaning m2 and signal s3

Create a variable containing the following reception matrix:

	m_1	m_2	m_3
s_1	1	2	0
s_2	0	2	1
s_3	2	0	3

Print the weights for signal s3
Print the weight of the connection between signal s1 and meaning m2

The code proper

The code begins by importing the random and pyplot modules; this allows us to use Python's built-in random number generator and plotting functions (see the worksheet for lab 1).

```
import random
import matplotlib.pyplot as plt
```

Function wta

The function wta ("winner takes all") takes a list of numbers (**`items`**) as its parameter; this represents a row of a production or reception matrix. The function returns the index of the largest number in the list `items`. If there multiple equally large numbers, then one of them is chosen at random.

```
def wta(items):
    maxweight = max(items)
    candidates = []
    for i in range(len(items)):
        if items[i] == maxweight:
            candidates.append(i)
    return random.choice(candidates)
```

`maxweight = max(items)` uses the built-in function `max` to calculate the maximum value of `items` and allocates this value to `maxweight`.

`candidates = []` creates an empty list.

`for i in range(len(items)):`
`range(len(items))` creates a sequence of numbers from 0 to (not including) the length of `items`. These represent each possible index of `items`, and in the for loop we go through each in turn, allocating it to the variable **`i`**, and then carrying out everything in the next code block for each value of **`i`**:

```
    if items[i] == maxweight:
```

candidates.append(i)

This checks each member of `items` in turn; if its value is equal to `maxweight`, then the index (`i`) is appended to (added to) the list of `candidates`.

After this loop has been completed, **`candidates`** will contain the indexes of all the largest numbers.

`return random.choice(candidates)` returns a random choice from the numbers in `candidates`. If there is only one number in `candidates`, then this is returned.

Using wta and the variables you created above to store the production and reception matrices:
find the preferred signal for each meaning in turn
find the preferred meaning for each signal in turn

Are the results as you would expect?

Function communicate

The function communicate plays a communication episode; it takes three parameters:

- **`speaker_system`**, the production matrix of the speaker;
- **`hearer_system`**, the reception matrix of the hearer, and
- **`meaning`**, the index of the meaning which is to be communicated.

```
def communicate(speaker_system, hearer_system, meaning):
    speaker_signal = wta(speaker_system[meaning])
    hearer_meaning = wta(hearer_system[speaker_signal])
    if meaning == hearer_meaning:
        return 1
    else:
        return 0
```

In a communication episode, the speaker chooses the signal it uses to communicate `meaning`, and expresses this signal to the hearer; the hearer then chooses the meaning it understands by the speaker's signal. If the hearer's meaning is the same as the speaker's meaning, then the communication episode succeeds, otherwise it fails.

`speaker_signal = wta(speaker_system[meaning])` uses `speaker_system[meaning]` to extract a list of association strengths from the speaker's production matrix (`speaker_system`) for `meaning`, and then uses `wta` (see above) to find the index corresponding to the largest of these weights. This value is then stored in the variable `speaker_signal`.

`hearer_meaning = wta(hearer_system[speaker_signal])` uses `hearer_system[speaker_signal]` to extract a list of association strengths from the hearer's reception matrix (`hearer_system`) for `speaker_signal`, and then uses `wta` (see above) to find the index corresponding to the largest of these weights. This value is then stored in the variable `hearer_meaning`.

```
if meaning == hearer_meaning:
    return 1
else:
    return 0
```

If the hearer's interpretation of the speaker's signal (`hearer_meaning)` equals the original value of `meaning` (i.e. the meaning the speaker was trying to convey) and thus the communication episode succeeds, then the function `returns` 1, otherwise (`else`) it `returns` 0.

Using the same matrices you created earlier, find out which of the meanings can be successfully communicated using these production and reception matrices.

Function ca_monte

The function ca_monte ("**c**ommunicative **a**ccuracy **Monte** Carlo") is the main function in this program. It performs a Monte Carlo simulation, which runs a set number of communication episodes between a production system and a reception system, calculates how many of them were communicatively successful, and returns a trial-by-trial list of results. It takes three parameters:

- **`speaker_system`**, the production matrix of the speaker;
- **`hearer_system`**, the reception matrix of the hearer, and
- **`trials`**, the number of trials of the simulation, or the number of communicative episodes over which communicative accuracy should be calculated.

```
def ca_monte(speaker_system, hearer_system, trials):
    total = 0.
    accumulator = []
    for n in range(trials):
        total += communicate(speaker_system, hearer_system,
                             random.randrange(len(speaker_system)))
        accumulator.append(total/(n+1))
    return accumulator
```

`total = 0.` creates a variable called total, which will store the number of successful communicative episodes. Note the trailing decimal point, which tells Python that this number should be stored as a floating-point number.

`accumulator = []` creates a variable called accumulator, which will be used to build up a list of trial-by-trial success rates. We initialise accumlator with an empty list: before we have conducted any trials, we don't have any results for success or failure.

`for n in range(trials):`

range(trials) creates a sequence of numbers from 0 to (not including) `trials`, which is then traversed in the for loop.

```
total += communicate(speaker_system, hearer_system,
                     random.randrange(len(speaker_system)))
```

On each communicative episode, we choose a random meaning (`random.randrange (len(speaker_system))` from the speaker's signalling system, then use the function `communicate` to see whether the speaker can successfully communicate this meaning to the hearer (`hearer_system`). We add the value returned by `communicate` (i.e. 0 or 1) to the existing value in `total`, which therefore contains the number of successful communicative episodes.

```
accumulator.append(total/(n+1))
```

We want to build up an exposure-by-exposure list of the proportion of communicative episodes so far which have been successful. `total/n+1` gives the proportion of events to date that have been successful: this is the number of successful events (which we are storing in `total`), divided by the number of trials we have conducted up to this point, which is `n +1`. Note that the number of trials conducted so far is `n+1`, not just `n`: because of the way `range` works, the first trial is `n=0`, the second trial is `n=1`, and so on, so we have to add 1 to this number to get the actual number of trials completed. We then use `append` to add this value to `accumulator`, which is our building list of trial-by-trial successes.

return accumulator returns the list of trial-by-trial list giving proportion of successful communicative events. Note that this line of code is outside the for loop: `accumulator` is only returned once the for loop has run the necessary number of trials.

What is the overall communicative accuracy for the matrices you defined earlier?

Change the ca_monte function so that the trailing decimal point is removed from the definition and run it again. What happens?

Create another matrix (maybe with more meanings and/or signals). What is its communicative accuracy?