Agents

The NetLogo world is made up of agents. Agents are beings that can follow instructions. There are four types of agents:

- **Turtles**: Turtles are agents that move around in the world.
- **Patches**: The world is two dimensional and is divided up into a grid of patches. Each patch is a square piece of "ground" over which turtles can move.
- **Links**: Links are agents that connect two turtles. Links can be directed (from one turtle to another turtle) or undirected (one turtle with another turtle).
- **The observer**: The observer doesn't have a location - you can imagine it as looking out over the world of turtles, links and patches.

Instructions

Instructions tell agents what to do. There are three characteristics that are useful to remember about instructions:

a) Whether the instruction is implemented by the user (procedures), or whether it is built into NetLogo (primitives). Once you define a procedure, you can use it elsewhere in your program. The NetLogo Dictionary has a complete list of built-in instructions (i.e. a Primitives Dictionary).

```netlogo
to setup
  clear-all
  create-turtles 10
end
```

In this program, **setup** is a procedure (since it is implemented by us), whereas **clear-all** and **create-turtles** are both primitives (they are built into NetLogo).

b) Whether the instruction produces an output (reporters) or not (commands).

- A reporter computes a result and reports it. Most reporters are nouns or noun phrases (e.g. "average-wealth", "most-popular-girl"). These names are preceded by the keyword to-report. The keyword end marks the end of the instructions in the procedure.

```netlogo
to-report average-wealth
  report mean [wealth] of turtles ;; average wealth in the population of turtles
end
```

- A command is an action for an agent to carry out. Most commands begin with verbs (e.g. "create", "die", "jump", "inspect", "clear"). These verbs are preceded by the keyword to (instead of to-report). The keyword end marks the end of the instructions in the procedure.

```netlogo
to go
  ask turtles [ forward 1 ;; all turtles move forward one step
    right random 360 ] ;; ...and turn a random amount
end
```

c) Whether the instruction takes an input (or several inputs) or not. Inputs are values that the instruction uses in carrying out its actions.

```netlogo
to-report absolute-value [number] ;; number is the input
  ifelse number >= 0 ;; if number is already non-negative
    [ report number ] ;; return number (a non-negative value).
    [ report 0 - number ] ;; Otherwise, return the opposite, which is then necessarily positive.
end
```
Variables

Variables are places to store values (such as numbers). A variable can be a global variable, a turtle variable, a patch variable, a link variable, or a local variable (local to a procedure). To change the value of a variable you can use the set command (If you don't set the variable to any value, it starts out storing a value of zero).

a) Global variables: If a variable is a global variable, there is only one value for the variable, and every agent can access it. You can make a global variable by adding a switch, a slider, a chooser or an input box to your model, or by using the globals keyword at the beginning of your code, like this:

```
globals [ number-of-trees ]
```

b) Turtle, patch, and link variables: Each turtle has its own value for every turtle variable, each patch has its own value for every patch variable, and each link has its own value for every link variable. Turtle, patch, and link variables can be built-in or defined by the user.

- Built-in variables: For example, all turtles and all links have a color variable, and all patches have a pcolor variable. If you set this variable, the corresponding turtle, link or patch changes color. Other built-in turtle variables are xcor, ycor, and heading. Other built-in patch variables include pxcor and pycor. Other built-in link variables are end1, end2, and thickness. You can find the complete list in the NetLogo Dictionary.

- User-defined turtle, patch and link variables: You can also define new turtle, patch or link variables using the turtles-own, patches-own, and links-own keywords respectively, like this:

```
turtles-own [ energy ] ;; each turtle has its own energy
patches-own [ roughness ] ;; each patch has its own roughness
links-own [ strength ] ;; each link has its own strength
```

c) Local variables: A local variable is defined and used only in the context of a particular procedure or part of a procedure. To create a local variable, use the let command. You can use this command anywhere. If you use it at the top of a procedure, the variable will exist throughout the procedure. If you use it inside a set of square brackets, for example inside an ask, then it will exist only inside those brackets.

```
to swap-colors [turtle1 turtle2] ;; turtle1 and turtle2 are inputs
  let temp ([color] of turtle1) ;; store the color of turtle1 in temp
  ask turtle1 [ set color ([color] of turtle2) ] ;; set turtle1’s color to turtle2’s color
  ask turtle2 [ set color temp ] ;; now set turtle2’s color to turtle1’s (original) color
end ;; (which was conveniently stored in temp).
```

Setting and reading the value of variables

Global variables can be read and set at any time by any agent. Every agent has direct access to her own variables, both for reading and setting. Sometimes you will want an agent to read or set a different agent’s variable; to do that, you can use ask (which is explained in detail a bit later):

```
ask turtle 5 [ show color ] ;; print turtle 5’s color
ask turtle 5 [ set color blue ] ;; turtle 5 becomes blue
```

You can also use of to make one agent read another agent’s variable. of is written in between the variable name and the relevant agent (i.e. [reporter] of agent). Example:

```
show [color] of turtle 5 ;; same as the first line in the code above
```

Finally, a turtle can read and set patch variables of the patch it is standing on directly, e.g.

```
ask turtles [ set pcolor red ]
```
The code above causes every turtle to make the patch it is standing on red. (Because patch variables are shared by turtles in this way, you cannot have a turtle variable and a patch variable with the same name – e.g. that is why we have color for turtles and pcolor for patches—).

**Ask**

NetLogo uses the ask command to specify commands that are to be run by turtles, patches or links. Usually, the observer uses ask to ask all turtles or all patches to run commands. Here's an example of the use of ask syntax in a NetLogo procedure:

```plaintext
to setup
  clear-all       ;; clear the world
  create-turtles 100 ;; create 100 new turtles with random heading
  ask turtles
    [ set color red ;; turn them red
      forward 50 ] ;; make them move 50 steps forward
  ask patches
    [ if (pxcor > 0) ;; patches with pxcor greater than 0
      set pcolor green ] ;; turn green
end
```

You can also use ask to have an individual turtle, patch or link run commands. The reporters turtle, patch, link, and patch-at are useful for this technique. For example:

```plaintext
to setup
  clear-all       ;; clear the world
  create-turtles 3 ;; make 3 turtles
  ask turtle 0 [ fd 10 ] ;; tell the first one to go forward 10 steps
  ask turtle 1
    [ set color green ] ;; ...to become green
  ask patch 2 -2
    [ set pcolor blue ] ;; ...to become blue
  ask turtle 0
    [create-link-to turtle 2] ;; to link to turtle with who number 2
  ask link 0 2
    [ set color blue ] ;; ...to become blue
  ask turtle 0
    [ ask patch-at 1 0 ;; ...to ask the patch to her east
      set pcolor red ] ;; ...to become red
end
```

**Lists**

In the simplest models, each variable holds only one piece of information, usually a number or a string. The list feature lets you store multiple pieces of information in a single variable by collecting those pieces of information in a list. Each value in the list can be any type of value: a number, a string, an agent, an agentset, or even another list.

**Constant lists**

You can make a list by simply putting the values you want in the list between brackets, e.g.:

```
set my-list [2 4 6 8]
```

**Building lists on the fly**

If you want to make a list in which the values are determined by reporters, as opposed to being a series of constants, use the list reporter. The list reporter accepts two other reporters, runs them, and reports the results as a list.

```
set my-random-list list (random 10) (random 20)
```

To make shorter or longer lists, you can use the list reporter with fewer or more than two inputs, but in order to do so, you must enclose the entire call in parentheses, e.g.:

```
show (list random 10)
show (list random 10 random 20 random 30)
```
The of primitive lets you construct a list from an agentset (i.e. a set of agents). It reports a list containing each agent's value for the given reporter (syntax: [reporter] of agentset).

```
set fitness-list ([fitness] of turtles) ;; list containing the fitness of each turtle (in random order)
show [pxcor * pycor] of patches
```

See also: n-values, and sentence.

**Changing list items**
Technically, lists cannot be modified, but you can construct new lists based on old lists. If you want the new list to replace the old list, use set. For example:

```
set my-list [2 7 5 Bob [3 0 -2]] ;; my-list is now [2 7 5 Bob [3 0 -2]]
set my-list replace-item 2 mylist 10 ;; my-list is now [2 7 10 Bob [3 0 -2]]
```

See also: lput, fput, but-last, and but-first. (Note that fput is much faster than lput)

**Iterating over lists**
To apply a function on each item in a list, you can use foreach or map. foreach is used to run a block of commands on each item in a list. It takes as inputs the list and a block of commands, e.g.:

```
foreach [2 4 6]
[ 
  show (word "created " ? " turtles")
]
=> created 2 turtles => created 4 turtles => created 6 turtles
```

map is similar to foreach, but it is a reporter (it returns a list). It takes as inputs the input list and a reporter, and returns an output list containing the results of applying the reporter to each item in the input list. As when using foreach, the keyword ? refers to the current item in the list.

```
show map [round ?] [1.2 2.2 2.7] ;; prints [1 2 3]
```

Both foreach and map can take multiple lists as input; in that case, the block of commands (in foreach), or the input reporter (in map), is run for each group of items from each list, i.e. it is run once for the first items, once for the second items, and so on. In the block of commands (if you are using foreach), or in the input reporter (if you are using map), write ?1 through ?n to refer to the current item of each list. ?1 refers to an item from the first list, ?2 an item from the second list...

```
show (map [?1 + ?2] [1 2 3] [100 200 300]) ;; prints [101 202 303]
```

See also: repeat and while.

**Agentsets**
An agentset is a set of agents; all agents in an agentset must be of the same type (i.e. turtles, patches, or links). An agentset is not in any particular order. In fact, it’s always in a random order.

What’s powerful about the agentset concept is that you can construct agentsets that contain only some turtles, some patches, or some links. For example, all the red turtles, or the patches with pxcor evenly divisible by five, or all the links departing from a certain agent. These agentsets can then be used by ask or by various reporters that take agentsets as inputs.

Simple built-in agentsets are given by turtles-here (which contains only the turtles on my patch) and by turtles-at (only the turtles on some other particular patch). The primitive with is very useful to build agentsets. Here are some examples of how to make agentsets:

```
turtles with [color = red] ;; all red turtles
other turtles-here with [color = red] ;; all other red turtles on my patch
patches with [pxcor > 0] ;; patches with positive pxcor
patches in-radius 3 ;; all turtles three or fewer patches away
patches at-points [[1 0] [0 1] [-1 0] [0 -1]] ;; the four patches to the east, north, west, and south
neighbors4 ;; shorthand for those four patches
[my-links] of turtle 0 ;; all the links connected to turtle 0
```

1 If you want agents to do something in a fixed order, you can make a list of the agents instead.
Once you have created an agentset, here are some simple things you can do:

- Use **ask** to make the agents in the agentset do something.
- Use **any?** to see if the agentset is empty.
- Use **all?** to see if every agent in an agentset satisfies a condition.
- Use **count** to find out exactly how many agents are in the set.

Here are some more complex things you can do:

```plaintext
ask one-of turtles [ set color green ]
;; one-of reports a random agent from an agentset
ask (max-one-of turtles [wealth]) [ donate ]
;; max-one-of agentset [reporter] reports an agent in the
;; agentset that has the highest value for the given reporter
show mean ([wealth] of turtles)
;; Use of to make a list of values, one for each agent in the agentset.
show (turtle-set turtle 0 turtle 2 turtle 9)
;; Use turtle-set, patch-set and link-set reporters to make new
;; agentsets by gathering together agents from a variety of sources
show turtles = patches
;; Check whether two agentsets are equal using = or !=
show member? turtle 0 turtles
;; Use member? to see if an agent is a member of an agentset.
if all? turtles [color = red]
;; use ?all to see if every agent in the
[ show "every turtle is red!" ]
;; agentset satisfies a certain condition
ask turtles [create-links-to other turtles-here]
;; on same patch as me, not me,
with [color = [color] of myself]
;; and with same color as me.
show [[(color] of end1) - ([color] of end2)] of links
;; check everything’s OK
```

### Synchronization

When you **ask** a set of agents to run more than one command, each agent must finish before the next agent starts. One agent runs all of the commands, then the next agent runs all of them, and so on. As mentioned before, the order in which agents are chosen to run the commands is random. To be clear, consider the following code:

```plaintext
ask turtles
[ forward random 10 ]
wait 0.5
set color blue ]
;; move forward a random number of steps (bt 0 and 9)
;; wait half a second
;; set your color to blue

The first (randomly chosen) turtle will move forward some steps, she will then wait half a second, and she will finally set her color to blue. Then, and only then, another turtle will start doing the same; and so on until all turtles have run the commands inside ask without being interrupted by any other turtle. The order in which turtles are selected to run the commands is random. If you want all turtles to move, and then all wait, and then all become blue, you can write it this way:

```plaintext
ask turtles [ forward random 10 ]
ask turtles [ wait 0.5 ]
;; note that you will have to wait
ask turtles [ set color blue ]
;; (0.5 * number-of-turtles) seconds
```

Finally, you can make agents execute a set of commands in a certain order by converting the agentset into a list. There are two primitives that help you do this, **sort** and **sort-by**.

```plaintext
;; This sets my-list-of-agents to a list of turtles sorted in
;; ascending order by their turtle variable size.
foreach my-list-of-agents [ ask ? [ forward random 10
wait 0.5
set color blue
]]
;; each agent undertakes the list of commands
;; (forward, wait, and set) without being
;; interrupted, i.e. the next agent does not
;; start until the previous one has finished.
```

See also: **ask-concurrent** and **without-interruption**.
Tick counter

In many NetLogo models, time passes in discrete steps, called "ticks". NetLogo includes a built-in tick counter so you can keep track of how many ticks have passed. The current value of the tick counter is shown above the view (the black window where you can see patches and turtles).

In code, to retrieve the current value of the tick counter, use the ticks reporter. The tick command advances the tick counter by 1. The clear-all command resets the tick counter to 0. If you want to reset the counter to 0 without clearing everything, use the reset-ticks command.

Skeleton of many NetLogo Models

```plaintext
globals [ ... ] ;; global variables (also defined with sliders, ...) 
turtles-own [ ... ] ;; user-defined turtle variables (also <breeds>-own) 
patches-own [ ... ] ;; user-defined patch variables 
links-own [ ... ] ;; user-defined link variables (also <breeds>-own) 

... to setup
  clear-all ... setup-patches ... setup-turtles ... setup-graphs ...
end

... to go
  conduct-observer-procedure ...
  ask turtles [conduct-turtle-procedure] ...
  ask patches [conduct-patch-procedure] ...
  tick
  update-graphs ;; this may include update-view, update-plots...
end

... to update-plots
  set-current-plot "my-plot"
    set-current-plot-pen "my-pen"
    plot statistics ...
  set-current-plot "my-XY-plot"
    plotxy ticks (count turtles) ...
  set-current-plot "my-histogram"
    histogram [age] of turtles ...
end

... to-report statistics
  ... report the-result-of-some-formula
end
```

Common Primitives

**Turtle-related**: die, forward (fd), move-to, my-links, myself, nobody, of, other, other-end, patch here, right (rt), self, setxy, turtle, turtle-set, turtles, turtles-at, turtles-here, turtles-on, turtles-own.

**Patch-related**: clear-patches (cp), distance, import-pcolors, myself, neighbors, neighbors4, nobody, of, other, patch, patch-at, patch-set, patches, patches-own.

**Link-related**: both-ends, clear-links, create-link-from, create-link-to, create-link-with, in-link-neighbor?, in-link-neighbors, in-link-from, is-directed-link?, link, link-length, link-neighbors, link-set, link-with, my-in-links, my-links, other-end, out-link-neighbor?, out-link-neighbors, out-link-to, tie.

**Agentset primitives**: all?, any?, ask, ask-concurrent, count, in-radius, is-agentset?, max-one-of, min-one-of, n-of, neighbors, of, one-of, other, sort, sort-by, with, with-max, with-min.

**Control flow and logic primitives**: and, ask, foreach, if, ifelse, ifelse-value, let, loop, map, not, or, repeat, report, set, stop, startup, wait, while, with-local-randomness, without-interruption, xor.

**World primitives**: clear-all (ca), clear-patches (cp), clear-turtles (ct), display, max-pxcor, min-pxcor, no-display, random-pxcor, reset-ticks, tick, ticks, world-width, world-height.