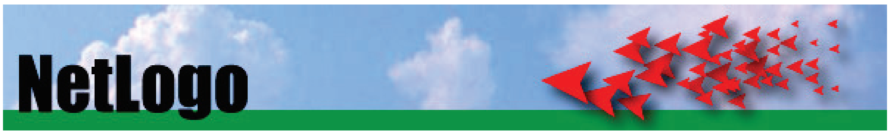
NetLogo

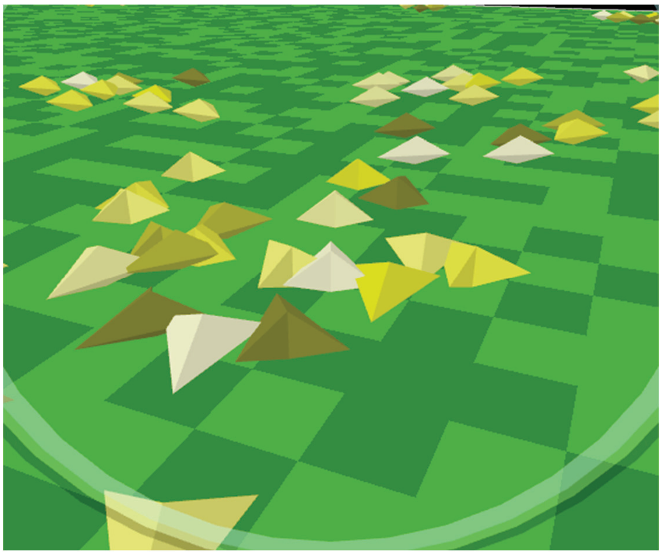
- **Corso di laurea in Informatica**
  (anno accademico 2024/2025)
  - Insegnamento: Apprendimento ed evoluzione
    in sistemi artificiali
  - Docente: Marco Villani

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
Dipartimento di
Scienze Fisiche,
Informatiche
e Matematiche

NetLogo

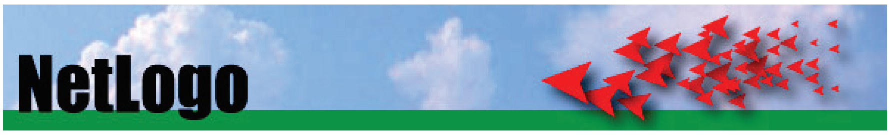


- **NetLogo is an agent-based programming language and integrated modeling environment**
- **NetLogo was designed, in the spirit of the Logo programming language, to be "low threshold and no ceiling"**
  - **It teaches programming concepts using agents in the form of turtles, patches, links, and the observer**

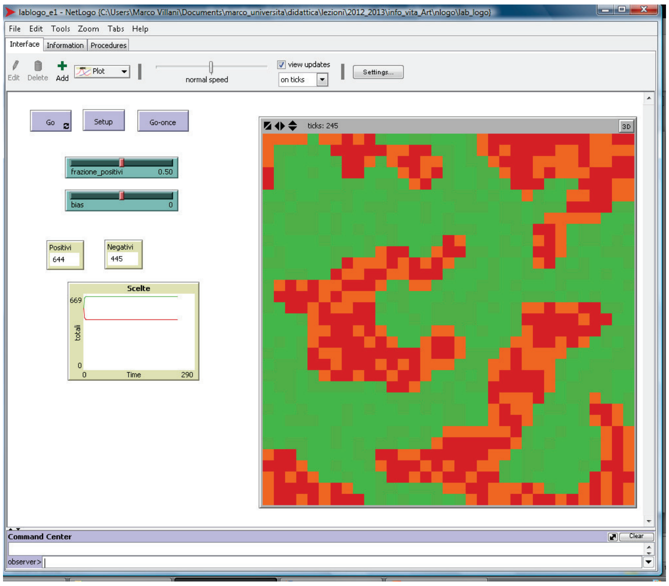


- **http://ccl.northwestern.edu/netlogo/**

NetLogo



- The NetLogo environment enables exploration of emergent phenomena
  - It comes with an extensive models library including models in a variety of domains, such as economics, biology, physics, chemistry, psychology, system dynamics
  - NetLogo allows exploration by modifying switches, sliders, choosers, inputs, and other interface elements
  - Beyond exploration, NetLogo allows authoring of new models and modification of existing models


lablogo_e1 - NetLogo {C:\Users\Marco Villani\Documents\marco_universita\didattica\lezioni\2012_2013\info_vita_Art\nlogo\lab_logo}

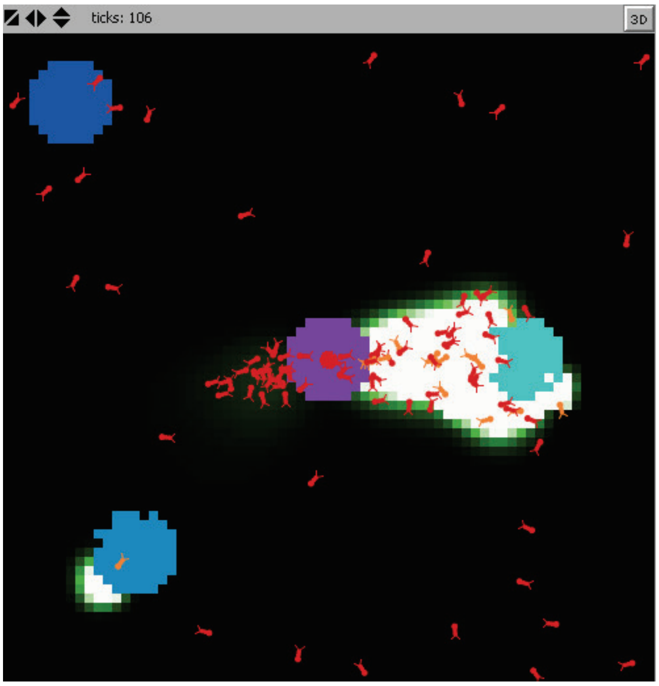
frazione_positivi 0.50

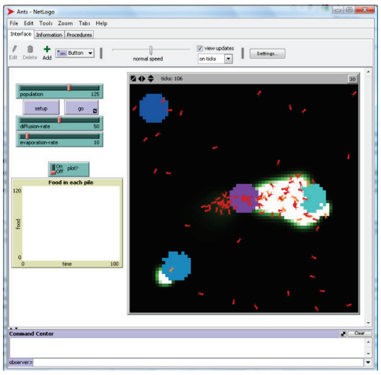
bias 0

Positivi 644

Negativi 445

Agents




Ants - NetLogo

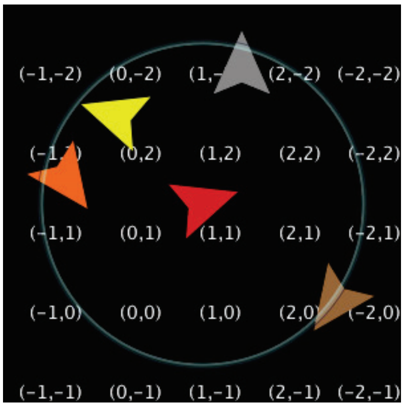
population 125

diffusion-rate 50

evaporation-rate 10

On Off plot?

Agents

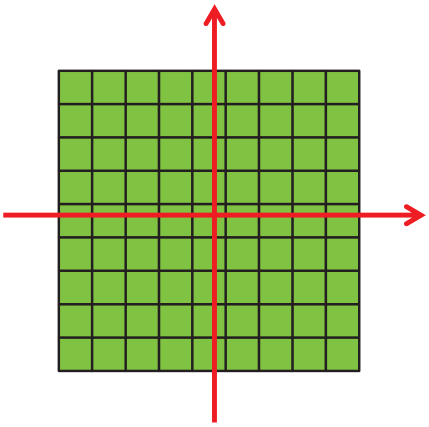


- The NetLogo world is made up of agents
  - Agents are beings that can follow instructions
  - Each agent can carry out its own activity, all simultaneously
- There are four types of agents:

  turtles, patches, links, and the observer
  - Turtles are agents that move around in the world
  - 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 are agents that connect two turtles
  - The observer doesn't have a location -- you can imagine it as looking out over the world of turtles and patches
  - The observer can make new turtles. Patches can make new turtles too.

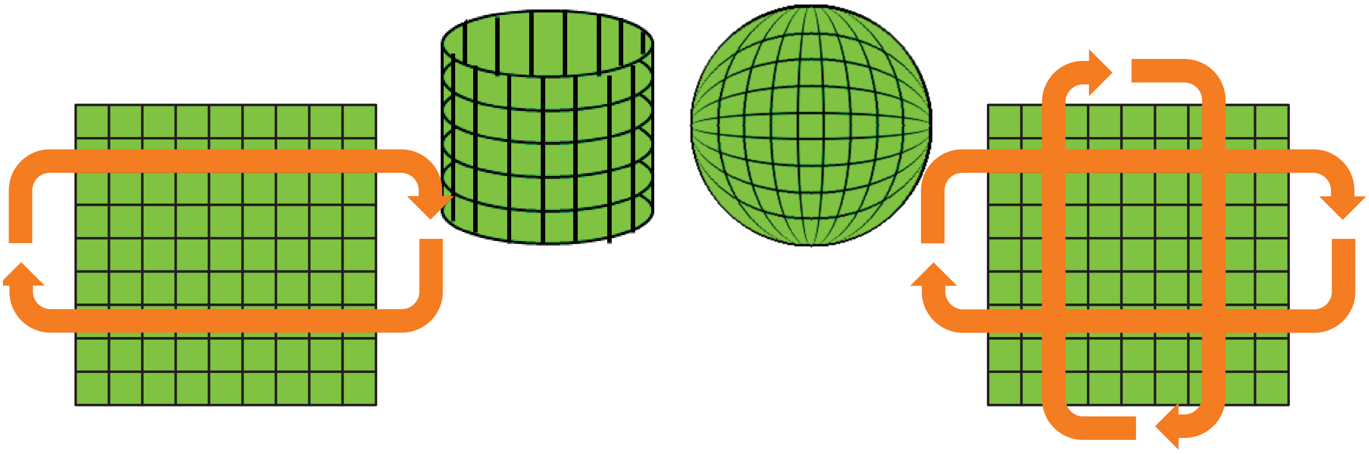
Patches

- Patches have coordinates
  - The patch at coordinates (0, 0) is called the origin and the coordinates of the other patches are the horizontal and vertical distances from this one
  - We call the patch's coordinates pxcor and pycor. Just like in the standard mathematical coordinate plane, pxcor increases as you move to the right and pycor increases as you move up
  - The total number of patches is determined by the settings min-pxcor, max-pxcor, min-pycor, and max-pycor (defaults are respectively -16, 16, -16, and 16, for a total of 1089 patches total)
  - Patch's coordinates are always integers



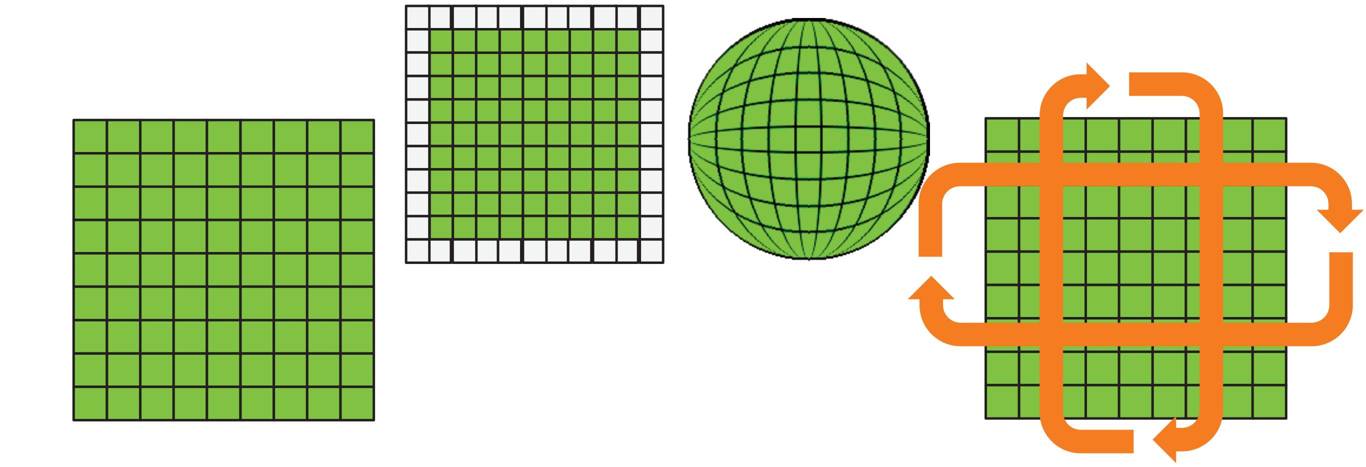
Patches

- The way the world of patches is connected can change
  - By default the world is a torus which means it isn't bounded, but "wraps“
  - However, you can change the wrap settings with the Settings button. If wrapping is not allowed in a given direction then in that direction (x or y) the world is bounded
  - Patches along that boundary will have fewer than 8 neighbors and turtles will not move beyond the edge of the world



Patches

- The way the world of patches is connected can change
  - By default the world is a torus which means it isn't bounded, but "wraps“
  - However, you can change the wrap settings with the Settings button. If wrapping is not allowed in a given direction then in that direction (x or y) the world is bounded
  - Patches along that boundary will have fewer than 8 neighbors and turtles will not move beyond the edge of the world



Patches

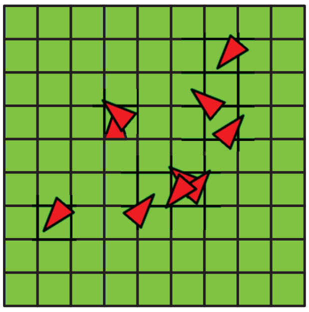
- List of all built-in patch variables:
  - pcolor - It holds the color of the patch
  - plabel - The patch appears with the given value "attached" to it as text
  - plabel-color - Determines what color the patch’s label appears in
  - pxcor pycor - It holds the current x (y) coordinate of the patch

- pcolor is a built-in patch variable: it holds the color of the patch
  - You can set this variable to make the patch change color
  - Color can be represented either as a NetLogo color (a single number) or an RGB color (a list of 3 numbers)

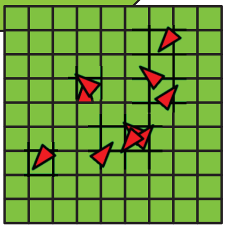
Turtles

- Turtles have coordinates
  - We call the turtle’s coordinates xcor and ycor.
  - A turtle's coordinates can have decimals. This means that a turtle can be positioned at any point within its patch; it doesn't have to be in the center of the patch

- All patch variables can be directly accessed by any turtle standing on the patch



Turtles



- List of all built-in turtle variables:
  - breed - It holds the type of the turtle
  - color - It holds the color of the turtle
  - heading - It indicates the direction the turtle is facing (degrees)
  - hidden? - It holds a boolean (true or false) value indicating whether the turtle is currently hidden (i.e., invisible)
  - label - The turtle appears with the given value "attached" to it as text
  - label-color - Determines what color the turtle's label appears in
  - pen-mode - You set the variable to draw lines, erase lines or stop either of these actions
  - pen-size - It holds the width of the line, in pixels
  - shape - It holds a string that is the name of the turtle current shape
  - size - It holds a number that is the turtle's apparent size (default=1)
  - who - It holds the turtle's ID number (“who number”), an integer >= 0
  - xcor ycor - It holds the current x (y) coordinate of the turtle

Turtles

- pen-mode is a built-in turtle variable; it holds the state of the turtle's pen
  - You can set the variable to draw lines, erase lines or stop either of these actions. Possible values are "up", "down", and "erase“
  - The built-in turtle variable pen-size holds the width of the line, in pixels, that the turtle will draw (or erase) when the pen is down (or erasing)
- pen-down (pd), pen-erase (pe), pen-up (pu)
  - The turtle changes modes between drawing lines, removing lines or neither
  - The lines will be displayed on top of the patches and below the turtles
  - When a turtle's pen is down, all movement commands cause lines to be drawn, including jump, setxy, and move-to

Context

- The observer
  - is “the God” of the system
  - has no physical place
  - is “out-of context” (or backwards, is the more ample possible context)
- Turtles
  - are agents that move around in the world
  - have a limited (but tunable) world view
- Patches
  - are agents that cannot move around in the world
  - have a neighborhood
- Links
  - are agents that can link two turtles

Variables

- Variables are places to store values. A variable can be a global variable, a turtle variable, or a patch variable
  - If a variable is a global variable, there is only one value for the variable, and every agent can access it
  - Each turtle has its own value for every turtle variable (e.g., color)
  - Each patch has its own value for every patch variable (e.g., pcolor)
- You can also define your own variables. You can make a global variable
  - by adding a switch or a slider to your model
  - by using the globals keyword at the beginning of your code, like this

```
globals [ score ]
```

Variables

- Variables are places to store values. A variable can be a global variable, a turtle variable, or a patch variable
  - If a variable is a global variable, there is only one value for the variable, and every agent can access it
  - Each turtle has its own value for every turtle variable (e.g., color)
  - Each patch has its own value for every patch variable (e.g., pcolor)

- You can also define your own variables. You can make a new turtle, patch or link variable using the turtles-own, patches-own and links-own keywords

```
turtles-own [energy speed]
patches-own [friction]
links-own [strength]
```

Variables

- Variables are places to store values. A variable can be a global variable, a turtle variable, or a patch variable
  - If a variable is a global variable, there is only one value for the variable, and every agent can access it
  - Each turtle has its own value for every turtle variable (e.g., color)
  - Each patch has its own value for every patch variable (e.g., pcolor)

- Global variables can be read and set at any time by any agent. As well, a turtle can read and set patch variables of the patch it is standing on. For example, this code

```
ask turtles [ set pcolor red ]
```

- causes every turtle to make the patch it is standing on red
  - Because patch variables are shared by turtles in this way, you can't have a turtle variable and a patch variable with the same name

Variables

- Variables are places to store values. A variable can be a global variable, a turtle variable, or a patch variable
  - If a variable is a global variable, there is only one value for the variable, and every agent can access it
  - Each turtle has its own value for every turtle variable (e.g., color)
  - Each patch has its own value for every patch variable (e.g., pcolor)

- In other situations where you want an agent to read a different agent's variable, you can use of

```
show [color] of turtle 5
;; prints current color of turtle with who number 5
```

```
show [xcor + ycor] of turtle 5
;; prints the sum of the x and y coordinates of
;; turtle with who number 5
```

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]
  let temp [color] of turtle1
  ask turtle1 [ set color [color] of turtle2 ]
  ask turtle2 [ set color temp ]
end
```

Procedures

- In NetLogo, commands and reporters tell agents what to do
  - A command is an action for an agent to carry out
  - A reporter computes a result and report it
- Commands and reporters built into NetLogo are called primitives. Commands and reporters you define yourself are called procedures
  - The NetLogo Dictionary has a complete list of built-in commands and reporters
  - Each procedure has a name, preceded by the keyword to.
  - The keyword end marks the end of the commands in the procedure
  - Once you define a procedure, you can use it elsewhere in your program

```
to setup
  clear-all       ;; clear the world
  crt 10          ;; make 10 new turtles (crt: short for “create-turtles”)
end
```

Procedures

- setup and go are user-defined commands
- clear-all, crt ("create turtles"), ask, lt ("left turn"), and rt ("right turn") are all primitive commands
- random and turtles are primitive reporters

```
to go
  ask turtles
   [ fd 1            ;; all turtles move forward one step
     rt random 10    ;; ...and turn a random amount
     lt random 10 ]
end
```

- random takes a single number as an input and reports a random integer that is less than the input (in this case, between 0 and 9)
- turtles reports the agentset consisting of all the turtles

```
to setup
  clear-all       ;; clear the world
  crt 10          ;; make 10 new turtles (crt: short for “create-turtles”)
end
```

Procedures

- setup and go can be called by other procedures or by buttons
- Many NetLogo models have a once button that calls a procedure called setup, and a forever button that calls a procedure called go

- In NetLogo, you must specify which agents (turtles, patches, links, or the observer) are running each command
- If you don't specify, the code is run by the observer
- In the code, the observer uses ask to make the set of the chosen agents run the commands between the square brackets

```
to go
 ask turtles
  [ fd 1             ;; all turtles move forward one step
    rt random 10     ;; ...and turn a random amount
    lt random 10 ]
end
```

```
to setup
  clear-all       ;; clear the world
  crt 10          ;; make 10 new turtles (crt: short for “create-turtles”)
end
```

Procedures

- clear-all and crt can only be run by the observer
- fd, on the other hand, can only be run by turtles
- Some other commands and reporters, such as set, can be run by different agent types

```
to go
 ask turtles
  [ fd 1             ;; all turtles move forward one step
    rt random 10     ;; ...and turn a random amount
    lt random 10 ]
end
```

```
to setup
  clear-all       ;; clear the world
  crt 10          ;; make 10 new turtles (crt: short for “create-turtles”)
end
```

Procedures with inputs

- Your own procedures can take inputs, just like primitives do
- To create a procedure that accepts inputs, include a list of input names in square brackets after the procedure name

```
to draw-polygon [num-sides len]
  pen-down
  repeat num-sides
   [ fd len                    ;; forward of len step
     rt 360 / num-sides ]   ;; turns right by 360/num-sides degrees
end
```

```
ask turtles [ draw-polygon 8 who ]
```

- Elsewhere in the program you could ask turtles to each draw an octagon with a side length equal to its who number
- who is a built-in turtle variable, an integer ID number greater than or equal to zero

Reporter procedures

- Just like you can define your own commands, you can define your own reporters
- First, use to-report instead of to to begin your procedure. Then, in the body of the procedure, use report to report the value you want to report.

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report (- number) ]
end
```

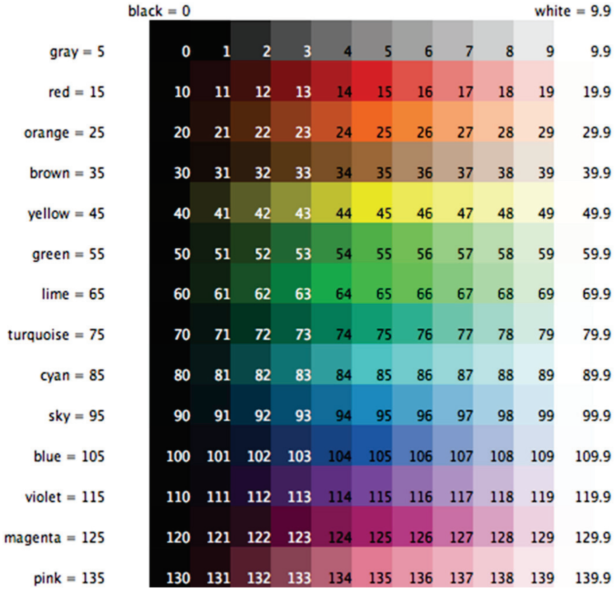
- if condition [commands]
  - If condition reports true, runs commands

```
if xcor > 0 [ set color blue ]
;; turtles in the right half of the world
;; turn blue
```

- ifelse reporter [commands1] [commands2]
  - Reporter must report a boolean (true or false) value
  - If reporter reports true, runs commands1
  - If reporter reports false, runs commands2

Colors (NetLogo representation)

- Numbers in the range 0 to 140, with the exception of 140 itself
  - If you use a number outside the 0 to 140 range, NetLogo will repeatedly add or subtract 140 from the number until it is in the 0 to 140 range.
  - Some of the colors have names (you can use these names in your code)
  - Every named color except black and white has a number ending in 5
  - On either side of each named color are darker and lighter shades of the color.
  - 0 is pure black. 9.9 is pure white
  - 10, 20, and so on are all so dark they appear black
  - 19.9, 29.9 and so on are all so light they appear white

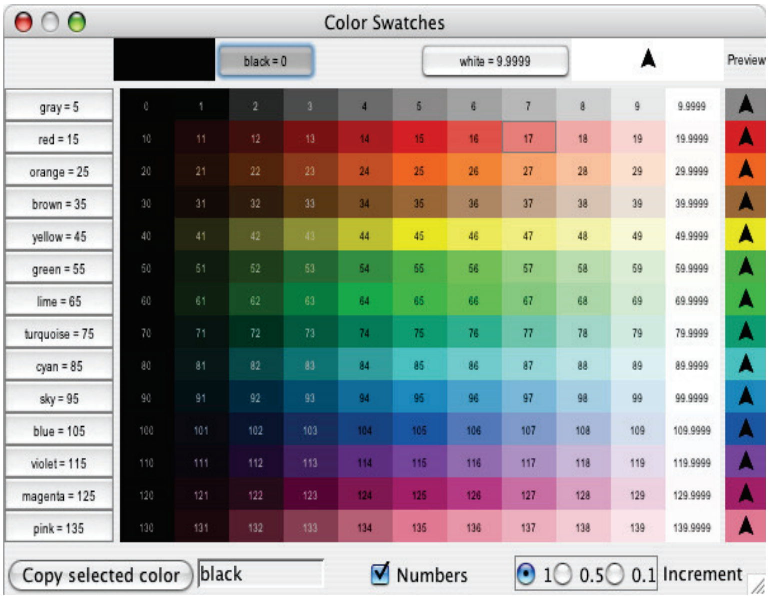


	black = 0										white = 9.9
gray = 5	0	1	2	3	4	5	6	7	8	9	9.9
red = 15	10	11	12	13	14	15	16	17	18	19	19.9
orange = 25	20	21	22	23	24	25	26	27	28	29	29.9
brown = 35	30	31	32	33	34	35	36	37	38	39	39.9
yellow = 45	40	41	42	43	44	45	46	47	48	49	49.9
green = 55	50	51	52	53	54	55	56	57	58	59	59.9
lime = 65	60	61	62	63	64	65	66	67	68	69	69.9
turquoise = 75	70	71	72	73	74	75	76	77	78	79	79.9
cyan = 85	80	81	82	83	84	85	86	87	88	89	89.9
sky = 95	90	91	92	93	94	95	96	97	98	99	99.9
blue = 105	100	101	102	103	104	105	106	107	108	109	109.9
violet = 115	110	111	112	113	114	115	116	117	118	119	119.9
magenta = 125	120	121	122	123	124	125	126	127	128	129	129.9
pink = 135	130	131	132	133	134	135	136	137	138	139	139.9

Colors (RGB representation)

- The second color representation in NetLogo is an RGB (red/green/blue) list.
- RGB lists are made up of three integers between 0 and 255
  - if a number is outside that range 255 is repeatedly subtracted until it is in the range
  - You can set any color variables in NetLogo (color for turtles and links and pcolor for patches) to an RGB list

```
set pcolor [255 0 0]
```



		black = 0					white = 9.9999					Preview
gray = 5	0	1	2	3	4	5	6	7	8	9	9.9999	▲
red = 15	10	11	12	13	14	15	16	17	18	19	19.9999	▲
orange = 25	20	21	22	23	24	25	26	27	28	29	29.9999	▲
brown = 35	30	31	32	33	34	35	36	37	38	39	39.9999	▲
yellow = 45	40	41	42	43	44	45	46	47	48	49	49.9999	▲
green = 55	50	51	52	53	54	55	56	57	58	59	59.9999	▲
lime = 65	60	61	62	63	64	65	66	67	68	69	69.9999	▲
turquoise = 75	70	71	72	73	74	75	76	77	78	79	79.9999	▲
cyan = 85	80	81	82	83	84	85	86	87	88	89	89.9999	▲
sky = 95	90	91	92	93	94	95	96	97	98	99	99.9999	▲
blue = 105	100	101	102	103	104	105	106	107	108	109	109.9999	▲
violet = 115	110	111	112	113	114	115	116	117	118	119	119.9999	▲
magenta = 125	120	121	122	123	124	125	126	127	128	129	129.9999	▲
pink = 135	130	131	132	133	134	135	136	137	138	139	139.9999	▲

Copy selected color | black | ☑ Numbers | ● 1 ○ 0.5 ○ 0.1 Increment

ASK

- NetLogo uses the ask command to give commands to turtles, patches, and links
- All code to be run by turtles must be located in a turtle "context". You can establish a turtle context in any of three ways
  - In a button, by choosing "Turtles" from the popup menu. Any code you put in the button will be run by all turtles
  - In the Command Center, by choosing "Turtles" from the popup menu. Any commands you enter will be run by all the turtle.
  - By using ask turtles
- The same goes for patches, links, and the observer, except that you cannot ask the observer. Any code that is not inside any ask is by default observer code

ASK

- NetLogo uses the ask command to give commands to turtles, patches, and links

```
to setup
  clear-all
  crt 100              ;; create 100 turtles
  ask turtles
    [ set color red              ;; turn them red
      rt random-float 360    ;; give them random headings
      fd 50 ]                        ;; spread them around
  ask patches
    [ if pxcor > 0                   ;; patches on the right side
        [ set pcolor green ] ]  ;; of the view turn green
end
```

ASK

- **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**

```
ask turtles
  [ fd 1
    set color red ]
```

- **first one turtle moves and turns red, then another turtle moves and turns red, and so on**

- **But if you write it this way:**

```
ask turtles [ fd 1 ]
ask turtles [ set color red ]
```

- **first all of the turtles move. After they have all moved, they all turn red**

ASK

```
to setup
  clear-all
  crt 3                      ;; make 3 turtles
  ask turtle 0                        ;; tell the first one...
    [ fd 1 ]                          ;; ...to go forward
  ask turtle 1                   ;; tell the second one...
    [ set color green ]          ;; ...to become green
  ask turtle 2                        ;; tell the third one...
    [ rt 90 ]                         ;; ...to turn right
  ask patch 2 -2               ;; ask the patch at (2,-2)
    [ set pcolor blue ]        ;; ...to become blue
  ask turtle 0                        ;; ask the first turtle
    [ ask patch-at 1 0                ;; ...to ask patch to the east
      [ set pcolor red ] ]            ;; ...to become red
  ask turtle 0                                   ;; tell the first turtle...
    [ create-link-with turtle 1 ]                ;; ...make a link with the second
  ask link 0 1                        ;; tell the link between turtle 0 and 1
    [ set color blue ]                ;; ...to become blue
end
```

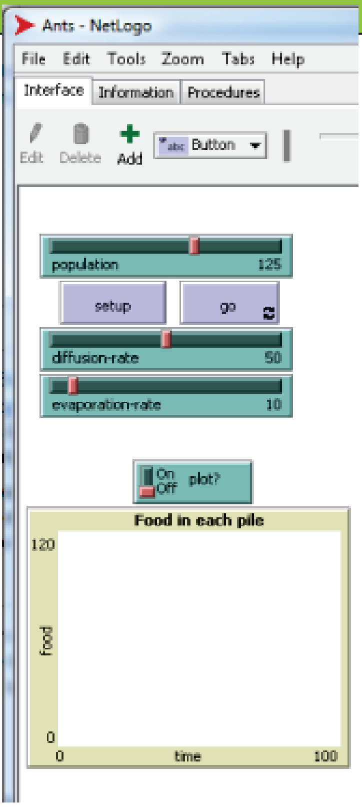
- **The turtle primitive reporter reports the turtle with the required who number**
- **The patch primitive reporter takes values for pxcor and pycor and reports the patch with those coordinates**
- **The patch-at primitive reporter takes offsets: distances, in the x and y directions, from the first agent**

Ask-Concurrent

- ask-concurrent produces simulated concurrency via a mechanism of turn-taking
  - The first agent takes a turn, then the second agent takes a turn, and so on until every agent in the asked agentset has had a turn. Then we go back to the first agent. This continues until all of the agents have finished running all of the commands
  - An agent's "turn" ends when it performs an action that affects the state of the world, such as moving, or creating a turtle, or changing the value of a global, turtle, patch, or link variable. (Setting a local variable doesn't count)
- The forward (fd) and back (bk) commands are treated specially
  - When used inside ask-concurrent, these commands can take multiple turns to execute. During its turn, the turtle can only move by one step. Thus, for example, fd 20 is equivalent to repeat 20 [fd 1], where the turtle's turn ends after each run of fd.
- Note the different actions in:

```
ask turtles [ fd 5 ]
ask-concurrent turtles [ fd 5 ]
```

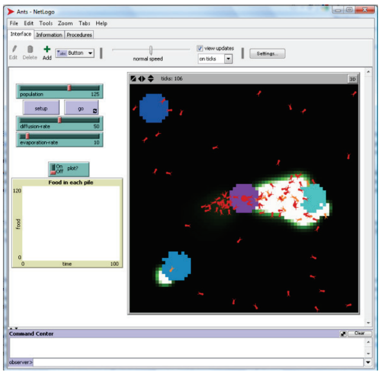
Graphical interface



population 125

diffusion-rate 50

evaporation-rate 10



Buttons

- Buttons in the interface tab provide an easy way to control the model. Typically a model will have
  - a "setup" button, to set up the initial state of the world
  - a "go" button to make the model run continuously
  - some models will have additional buttons that perform other actions

- A button contains some NetLogo code
  - That code is run when you press the button
  - A button may be either a "once button", or a "forever button“
    - Once buttons run their code once, then stop and pop back up
    - Forever buttons keep running their code over and over again, until either the code (i) hits the stop command or (ii) you press the button again to stop it
  - If you stop the button, the code doesn't get interrupted. The button waits until the code has finished, then pops up

Buttons

- Normally, a button is labeled with the code that it runs
  - But you can also edit a button and enter a "display name" for the button, which is a text that appears on the button instead of the code

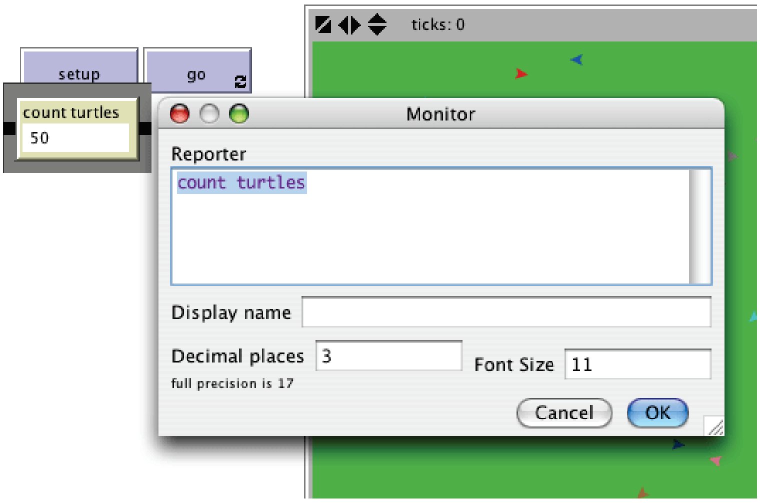
- When you put code in a button, you must also specify which agents you want to run that code
  - You can choose to have the observer run the code, or all turtles, or all patches, or all links

- When you edit a button, you have the option to assign an "action key“

- Buttons take turns
  - More than one button can be pressed at a time. If this happens, the buttons "take turns", which means that only one button runs at a time
  - Each button runs its code all the way through once while the other buttons wait, then the next button gets its turn

Monitors

- To create a monitor, you can use the monitor icon on the Toolbar and click on an open spot in the Interface
  - A dialog box will appear
  - In the dialog box you can type what you want to monitor (e.g., count turtles)
  - press the OK button to close the dialog box



count turtles
50

Reporter: count turtles

Display name:

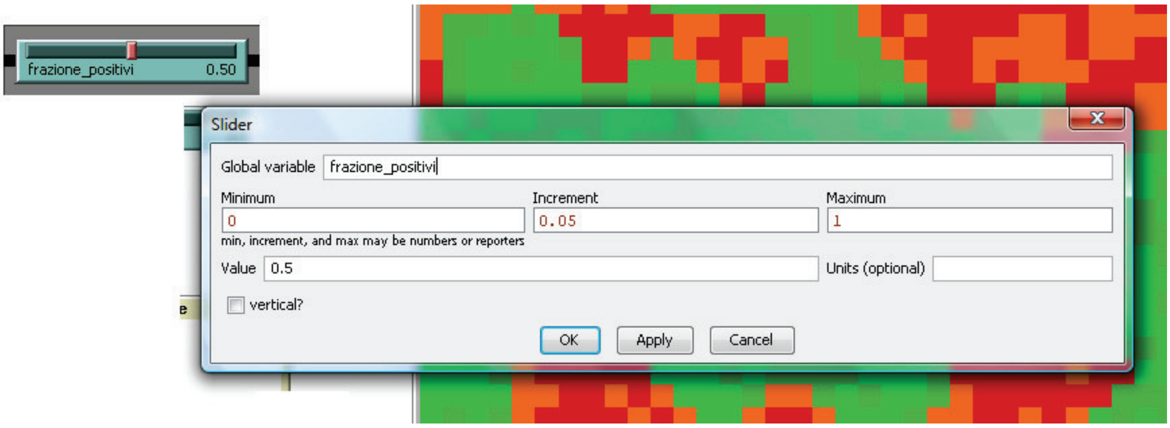
Decimal places: 3

full precision is 17

Font Size: 11

- **turtles** reports an "agentset", the set of all turtles
- **count** tells us how many agents are in that set

Sliders



frazione_positivi 0.50

Global variable: frazione_positivi

Minimum	Increment	Maximum
0	0.05	1

min, increment, and max may be numbers or reporters

Value: 0.5

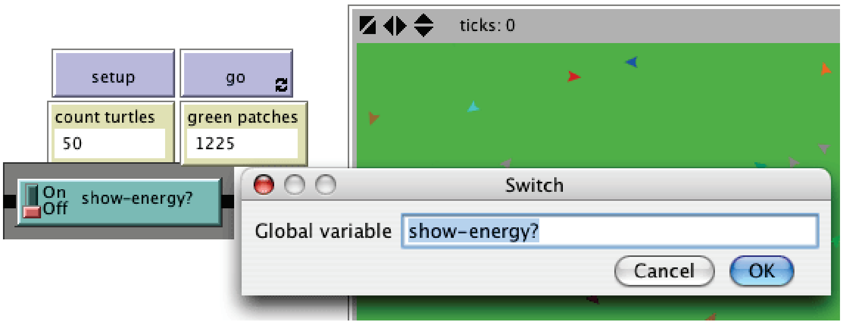
Units (optional):

□ vertical?

- To create a slider, you can use the monitor icon on the Toolbar and click on an open spot in the Interface
  - A dialog box will appear
  - In the dialog box you can type the variable you want create and manipulate (e.g., frazione_positivi)
  - press the OK button to close the dialog box
- The slider **creates a new global variable**

Switches and labels

- To create a switch, click on the switch icon on the Toolbar (in the Interface tab) and click on an open spot in the Interface
  - A dialog box will appear
  - In the Global variable section of the dialog box type: show-energy?
  - Don't forget to include the question mark in the name



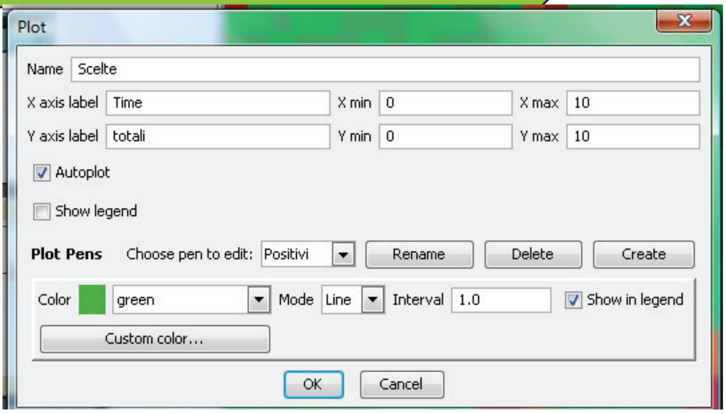
count turtles 50

green patches 1225

Global variable: show-energy?

Plotting

- To make plotting work, we'll need
  - to create a plot in the Interface tab
  - set some settings in it
  - then we have to add one more procedure to the Procedures tab, which will update the plot for us



Name: Scelte

X axis label: Time　X min: 0　X max: 10

Y axis label: totali　Y min: 0　Y max: 10

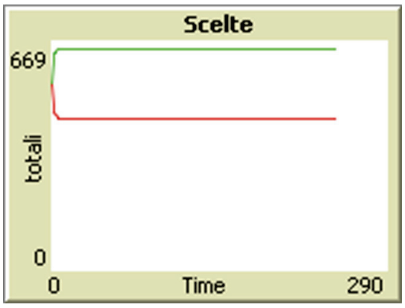
☑ Autoplot

□ Show legend

Choose pen to edit: Positivi

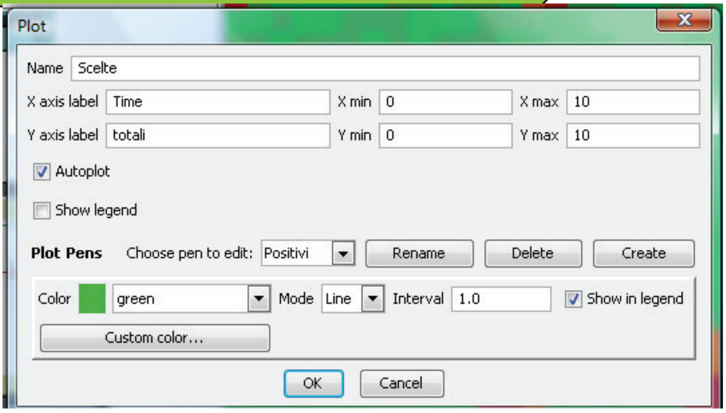
Color: 🟩 green　Mode: Line　Interval: 1.0　☑ Show in legend

```
to do-plots
  set-current-plot "Scelte"
  set-current-plot-pen "Positivi"
  plot numero_pos
  set-current-plot-pen "Negativi"
  plot numero_neg
end
```



Plotting

- To make plotting work, we'll need
  - Create a plot, using the plot icon on the Toolbar and click on an open spot in the Interface
  - Set its Name to “Scelte"
  - Set the X axis label to "time"
  - Set the Y axis label to "totali"



Plot

Name: Scelte

X axis label: Time X min: 0 X max: 10

Y axis label: totali Y min: 0 Y max: 10

☑ Autoplot

□ Show legend

Plot Pens Choose pen to edit: Positivi

Color 🟩 green Mode: Line Interval: 1.0 ☑ Show in legend

```
to do-plots
  set-current-plot "Scelte"
  set-current-plot-pen "Positivi"
  plot numero_pos
  set-current-plot-pen "Negativi"
  plot numero_neg
end
```

- To create two pens
  - (for each pen) Press the 'Create' button in the Plot dialog, to create a new pen
  - (for each pen) Enter the name of this pen and press OK in the "Enter Pen Name" dialog
  - (for each pen) Select the color for the pens
  - Select OK in the Plot dialog box

Agentsets

- An agentset is a set of agents
  - An agentset can contain either turtles, patches or links, but not more than one type at once

- An agentset is not in any particular order
  - In fact, it's always in a random order
  - And every time you use it, the agentset is in a different random order
  - So, no one agent always gets to go first

- The turtles primitive reports the agentset of all turtles
- The patches primitive reports the agentset of all patches
- The links primitive reports the agentset of all links

Agentsets

- It is possible to construct agentsets that contain only some turtles, some patches or some links.
  - All the red turtles
  - The patches with pxcor evenly divisible by five
  - The turtles in the first quadrant that are on a green patch

- One way is to use some reporters, as
  - turtles-here to make an agentset containing only the turtles on my patch
  - turtles-at to make an agentset containing only the turtles on some other patch at some x and y offsets
  - turtles-on
    - to make an agentset containing the turtles standing on a given patch or set of patches
    - to make an agentset containing the turtles standing on the same patch as a given turtle or set of turtles

Agentsets

```
other turtles                          ;; all other turtles
other turtles-here                     ;; all other turtles on this patch
turtles with [color = red]             ;; all red turtles
turtles-here with [color = red]        ;; all red turtles on my patch
patches with [pxcor > 0]               ;; patches on right side of view
turtles in-radius 3                    ;; all turtles less than 3 patches away

;; the four patches to the east, north, west, and south
patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
Neighbors4                             ;; shorthand for those four patches

;; turtles in the first quadrant that are on a green patch
turtles with [(xcor > 0) and (ycor > 0)  and (pcolor = green)]

turtles-on neighbors4          ;; turtles standing on my neighboring four patches
[my-links] of turtle 0         ;; all the links connected to turtle 0
```

Agentsets

- Once you have created an agentset, here are some 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
  - Use one-of to pick a random agent from the set

```
ask one-of turtles [ set color green ]
```

  - Use the max-one-of or min-one-of reporters to find out which agent is the most or least along some scale (for example, to remove it)

```
ask max-one-of turtles [sum assets] [ die ]
```

  - Use of to make a list of values, one for each agent in the agentset

```
show mean [sum assets] of turtles
```

Self - myself

- "self" is simple; it means "me“

- "myself" means "the turtle or patch who asked me to do what I'm doing right now"
  - when an agent has been asked to run some code, using myself in that code reports the agent (turtle or patch) that did the asking
  - myself is most often used in conjunction with of to read or set variables in the asking agent

```
ask turtles
  [ ask patches in-radius 3
      [ set pcolor [color] of myself ] ]
;; each turtle makes a colored "splotch" around itself
```

Breeds

- NetLogo allows you to define different "breeds" of turtles and breeds of links. Once you have defined breeds, you can go on and make the different breeds behave differently
  - For example, you could have breeds called sheep and wolves, and have the wolves try to eat the sheep

- You have to define turtle breeds using the breed keyword, at the top of the Procedures tab, before any procedures
  - You can refer to a member of the breed using the singular form, just like the turtle reporter

```
breed [wolves wolf]
breed [sheep a-sheep]
```

Breeds

- When you define a breed such as sheep, an agentset for that breed is automatically created, so that all of the agentset capabilities described above are immediately available with the sheep agentset

- The following new primitives are also automatically available once you define the breed sheep
  - create-sheep, hatch-sheep, sprout-sheep, sheep-here, sheep-at, sheep-on, and is-a-sheep?

- Also, you can use sheep-own to define new turtle variables that only turtles of the given breed have

Lists

- 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, or a string, an agent or agentset, or even another list

- Constant lists

```
set mylist [2 4 6 8 ]
set mylist [[2 4] [3 5]]
set mylist [ ]
```

- item

```
set mylist [2 4 6 8 ]
show item 2 mylist
=> 6
```

```
set mylist [2 4 6 8 ]
show item 0 mylist
=> 2
```

Lists on the fly

- If you want to make a list in which the values are not constants, but are determined by reporters, use the list reporter. The list reporter accepts two other reporters, runs them, and reports the results as a list
  - If I wanted a list to contain two random values, I might write:

```
set lista-casuale list (random 10) (random 20)
```

- If you wont to construct a list of a specific length by repeatedly running a given reporter
  - n-values size [reporter]

```
show n-values 5 [? * ?]
=> [0 1 4 9 16]
```

```
show n-values 5 [1]
=> [1 1 1 1 1]
show n-values 5 [?]
=> [0 1 2 3 4]
show n-values 3 [turtle ?]
=> [(turtle 0) (turtle 1) (turtle 2)]
```

Lists on the fly

- Note the use of ? in reporters to refer to the number of the item currently being computed, starting from zero

- If you wont to construct a list of a specific length by repeatedly running a given reporter
  - n-values size [reporter]

```
show n-values 5 [? * ?]
=> [0 1 4 9 16]
```

```
show n-values 5 [1]
=> [1 1 1 1 1]
show n-values 5 [?]
=> [0 1 2 3 4]
show n-values 3 [turtle ?]
=> [(turtle 0) (turtle 1) (turtle 2)]
```

Changing list items

- Technically, lists can't be modified, but
  - you can construct new lists based on old lists
  - you can use replace-item *index list value*
    - Note that replace-item is used in conjunction with set to change a list

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

Changing list items

- **To add an item to the end of a list, use the lput reporter**
  - **fput adds an item to the beginning of a list**

```
set mylist lput 42 mylist
; mylist is now [2 7 10 Bob [3 0 -2] 42]
```

- **The but-last (bl for short) reporter reports all the list items but the last**

```
set mylist but-last mylist
; mylist is now [2 7 10 Bob [3 0 -2]]
```

- **Suppose you want to get rid of item 0, the 2 at the beginning of the list (use of but-first)**

```
set mylist but-first mylist
; mylist is now [7 10 Bob [3 0 -2]]
```

Changing list items

```
; mylist is now [7 10 Bob [3 0 -2]]
```

- **Suppose you wanted to change the third item that's nested inside item 3 from -2 to 9**
  - **note that the name that can be used to call the nested list [3 0 -2] is item 3 mylist**
  - **then the replace-item reporter can be nested to change the list-within-a-list. The parentheses are added for clarity**

```
set mylist (replace-item 3 mylist (replace-item 2 (item 3 mylist) 9))
; mylist is now [7 10 Bob [3 0 9]]
```

Iterating over lists

- If you want to do some operation on each item in a list in turn, the foreach command and the map reporter may be helpful

- foreach is used to run a *command* or *commands* on each item in a list. It takes an input list and a block of commands
  - foreach list [commands]
  - (foreach list1 ... [commands])

```
foreach [1.1  2.2  2.6] [ show (word ? " -> " round ?) ]
=> 1.1 -> 1
=> 2.2 -> 2
=> 2.6 -> 3
```

Iterating over lists

- If you want to do some operation on each item in a list in turn, the foreach command and the map reporter may be helpful

- With multiple lists, foreach runs *commands* for each group of items from each list
  - the commands are run once for the first items, once for the second items, and so on
  - all the lists must be the same length
  - in commands, use ?1 through ?n to refer to the current item of each list

```
(foreach [1 2 3] [2 4 6] [ show
word "the sum is: " (?1 + ?2) ])
=> "the sum is: 3"
=> "the sum is: 6"
=> "the sum is: 9"
```

```
(foreach list (turtle 1) (turtle 2) [3 4]
  [ ask ?1 [ fd ?2 ] ])
;; turtle 1 moves forward 3 patches
;; turtle 2 moves forward 4 patches
```

Iterating over lists

- **If you want to do some operation on each item in a list in turn, the foreach command and the map reporter may be helpful**

- **With a single *list*, the given reporter is run for each item in the list, and a list of the results is collected and reported**
  - **map [reporter] list**
  - **(map [reporter] list1 ...)**

```
show map [round ?] [1.1 2.2 2.7]
=> [1 2 3]
show map [? * ?] [1 2 3]
=> [1 4 9]
```

Iterating over lists

- **If you want to do some operation on each item in a list in turn, the foreach command and the map reporter may be helpful**

- **With multiple lists, the given reporter is run for each group of items from each list**
  - **so, it is run once for the first items, once for the second items, and so on. All the lists must be the same length**
  - **in reporter, use ?1 through ?n to refer to the current item of each list**

```
show (map [?1 + ?2] [1 2 3] [2 4 6])
=> [3 6 9]
show (map [?1 + ?2 = ?3] [1 2 3] [2 4 6] [3 5 9])
=> [true false true]
```

Iterating over lists

- In some situations, you may need to use some other technique such as a loop using repeat or while, or a recursive procedure
  - repeat number [commands]
  - while [reporter] [commands]
    - note that in this case the reporter may have different values for different agents, so some agents may run commands a different number of times than other agents

```
pd repeat 36 [ fd 1 rt 10 ]
;; the turtle draws a circle
```

```
while [any? other turtles-here] [ fd 1 ]
;; turtle moves until it finds a patch that has
;; no other turtles on it
```

Iterating over lists

- The sort-by primitive uses a similar syntax to map and foreach, except that since the reporter needs to compare two objects
  - the two special variables ?1 and ?2 are used in place of ?

```
show sort-by [?1 < ?2] [4 1 3 2]
;; prints [1 2 3 4]
```

- Varying number of inputs
  - some commands and reporters involving lists and strings may take a varying number of inputs
  - In these cases, in order to pass them a number of inputs other than their default, the primitive and its inputs must be surrounded by parentheses

```
show list 1 2
=> [1 2]
show (list 1 2 3 4)
=> [1 2 3 4]
show (list)
=> []
```

Lists of agents

- Agentsets are always in random order, a different random order every time
  - if you need your agents to do something in a fixed order, you need to make a list of the agents instead

- There are two primitives that help you do this, sort and sort-by
  - both can take an agentset as input. The result is always a new list, containing the same agents as the agentset did, but in a particular order
  - if you use sort on an agentset of turtles, the result is a list of turtles sorted in ascending order by who number
  - if you use sort on an agentset of patches, the result is a list of patches sorted left-to-right, top-to-bottom
  - if you need descending order instead, you can combine reverse with sort, for example reverse sort turtles
  - if you want your agents to be ordered by some other criterion than the standard ones sort uses, you'll need to use sort-by

Asking a list of agents

- Once you have a list of agents, you might want to ask them each to do something. To do this, use the foreach and ask commands in combination:

```
foreach sort turtles [
  ask ? [
    ...
  ]
]
```