logopy Documentation

Release 0.1

Carl Waldbieser

Nov 29, 2022

Contents:

CHAPTER 1

logopy

logopy is an interpreter for the Logo programming language written in Python3. Logo is often associated with Turtle Graphics, and logopy integrates with Python's own *turtle* module.

I developed logopy after my son (then age 11) got a book from our local library about computer game programming. The first few chapters were all about Logo, while the later half of the book focused on Python.

Unfortunately, the book recommended Windows based interpreters, and we don't run Windows OS at home. Poking around on my Linux workstation, I found UCBLogo in the software repositories, so I installed it and fired it up. My son had fun making shapes while I went on to do something else.

Later on, after seeing some of the programs he had typed in from the book, I had a bit of nostalgia for when I was in grade school and I had entered commands to direct the turtle on the Apple IIe's in my school's computer room. I started writing some of my own commands to wow my son.

UCBLogo performed pretty well, but a couple of times the program crashed, and there were some features I wished it had. Eventually, I thought, "Someone must have written a Logo interpreter in Python! I mean, it already has the turtle graphics built in!". A bit of searching found Ian Bicking's pylogo. I was intially excited, but I had a bit of trouble getting it up and running. Trying to run it under Python3 failed, and the code looked like it hadn't been updated in quite some time.

That is when I had the ridiculous notion that writing my own interpreter would probably be much easier than trying to update the existing code. I told myself, "Oh, I'll try to throw something together in an hour. If it goes longer than that, I'll just forget this Logo business . . . " I kept telling myself that during countless snatches of free time I found myself working on this project.

Ultimately, I think the reason I kept working on it is because of something Seymour Papert, one of the inventers of Logo had written. He argues that people learn things by doing, and by making connections[1] . For me, programming is a process where I make all sorts of connections about what is going on in a system. The errors and mistakes actually make me slow down and think about what is going on. Why did the tokens parse this way? Why did the intepreter choose to evaluate this way and not that way? Let's see if I can get this to draw a square. Good, now a triangle? Easy! An ellipse? Hmm, that is a bit harder. I'm going to have to really think about this one. Am I going to have to pull out an old geomtry textbook? And so on.

[1] In *The Children's Machine Rethinking School in the Age of the Computer*, Papert devotes an entire chapter to suggest that the deliberate part of learning is mostly about making connections between mental artifacts that already exist in our minds.

Eventually, I've gotten the project to a state where is not so frustrating to use, and it can actually be fun to see my logo programs move the turtle around. I hope that anyone else who also enjoys Logo programming thinks so, too.

CHAPTER 2

Install and Run logopy

Two ways to install and run.

In both examples, I am running the interpreter in interactive GUI mode and have set the folder where I can load my logo scripts from with the *load* command.

2.1 Using *pip*

Install using *pip*:

```
$ pip install logopy
```

Then, run using:

```
$ logopycli.py -s ./example_scripts/ gui
```

2.2 Using *pipenv*

Information about pipenv .

Clone the repository from GitHub and install dependencies:

```
$ git clone https://github.com/cwaldbieser/logopy.git
$ cd logopy/
$ pipenv install
```

Run logopy:

```
$ cd logopy/
$ PYTHONPATH=. pipenv run ./bin/logopycli.py -s ./example_scripts/ gui
```

You must explicitly set the PYTHONPATH to include the *logopy* package folder when running the program using *pipenv* in this way since the library is not installed into your python's `site-packages` folder.

2.3 Logo Compatibility

I tried to maintain compatibility with UCBLogo when I could. There are some times I decided to make some changes that had to do with the environment, but most of the commands I've implemented at this time should behave the same as documented in the UCBLogo Manual .

CHAPTER 3

Turtle Back Ends

By default, when *logopy* is run, it will not activate any turtle graphics unless one of the turtle sub-commands is used (*gui*, *svg*) or if the program specified by the `-f` option invokes any turtle commands. In the latter case, the default Python-TKinter turtle backend is launched.

Specific sub-commands can be used to activate individual turtle back ends.

3.1 Python-TKinter Turtle Graphics

This back end is a very simple Tkinter application that incorporates the Python turtle module. Commands may be typed in the entry input at the bottom of the application. Output appears in the text widget above it. The turtle will draw in the large area at the top of the application.

The *HALT* command is not a Logo command, but is understood by this environment. It will cause the current command to abort as soon as possible. This is useful if the turtle is acting on a very time consuming series of commands.

3.2 SVG Turtle Backend

The Scalable Vector Graphics (SVG) turtle back end attempts to map turtle geometry to SVG graphics. This backend is not interactive. It executes a Logo program and writes the turtle output to an SVG file. The SVG file can be used directly in web pages or other applications that can use SVG. It can also be animated in web pages using Vivus .

3.3 Detailed Turtle Back End Information

3.3.1 SVG Turtle Back End

The SVG turtle back end is used to produce Scalable Vector Graphics from turtle commands. A line drawn by the turtle will be represented by the appropriate markup in SVG.

The SVG turtle is meant to be used in a non-interactive batch mode. The `-f` command line argument should be used to select a Logo program that contains turtle commands. For interactive exploration while developing images, use the GUI turtle back end.

The SVG turtle can be directed to write its results to a single SVG file using the `-o` command line option. The SVG image can then be used in applications, such as web browsers, which support SVG.

Because SVG enjoys such robust web browser support, the turtle can also be instructed to create a minimal web page that animates the SVG image so that it appears to be drawn on the page. The `--html` command line option can be used to specify a folder that the generated content will be placed into. The animation uses the vivus.js JavaScript library to animate the content.

Note: Vivus.js uses a different license than logopy. While logopy uses the GPLv3 license, vivus.js uses the MIT license. As a user of the software, this probably won't make much difference to you. If you plan on distributing the software, you may want to review your obligations under the different licenses.

There are additional command line options that allow you to influence the behavior of the animations:

- `--html-title` allows you to set the web page title.
- `--html-width` allows you to set the width of the image to be displayed in pixels.
- `--html-scale` allows you to specify the width of the image as a percentage of the total screen width.
- `--animation-duration` allows you to specify the animation duration in frames (see vivus documentation).
- `--animation-type` allows you to specify the animation type (see the vivus documentation).
- `--animation-start` allows you to specify if the animation should start automatically (**automatic**) or when the image scrolls into the viewport (**inviewport**).

Examples of SVG images created with the SVG turtle can be seen in the Logopy Gallery .

CHAPTER 4

LogoPy Compatibility Commands

LogoPy has some commands that are valid in all backends, but unlikely to be available in other Logo implementations. They are different from extension commands because they provide features that are necessary for writing portable Logo programs.

Note: Even in Logo implementations that do not provide these commands as primitives, it is not hard to provide implementations of these commands in the Logo language, itself.

4.1 CARTESIAN.HEADING

```
to CARTESIAN.HEADING :heading
```

The *CARTESIAN.HEADING* command takes a heading for the current LogoPy backend and maps it to a heading in the standard Cartesian coordinate system (North = 90 degrees, South = 270 degrees, East = 0 degrees, West = 180 degrees).

4.2 TURTLE.HEADING

```
to TURTLE.HEADING :heading
```

The *TURTLE.HEADING* command takes a Cartesian heading and outputs the corresponding heading in the current LogoPy backend.

CHAPTER 5

Extension Commands

Extension commands are primitives that are available in logopy, but unlikely to be available in other logo implementations. Further, some extensions may only be compatible with certain turtle backends.

Extensions provide interesting features at the expense of portability. All logopy extension commands begin with "EXT." to make it clear to the programmer that this is an extension that is probably not available on a different Logo interpreter.

5.1 EXT.ELLIPSE

```
to EXT.ELLIPSE :major :minor [:angle 360] [:clockwise true]
```

The *EXT.ELLIPSE* command draws an ellipse with its major and minor axes set to lengths *major* and *minor*. The major axis will bisect the ellipse at an angle equal to the current turtle heading. The minor axis bisects the ellipse at an angle perpendicular to the major axis. The major axis may be shorter than the minor axis.

The center of the ellipse will be located to the right of the turtle if the ellipse is drawn clockwise. If the ellipse is drawn anti-clockwise, the center will be to the left of the turtle. In both cases, the center of the ellipse will be half the length of the minor axis from the turtle.

An optional parameter may be specified to indicate the angle that should be swept out from the center of the ellipse when drawing its perimeter. In this manner, an elliptical arc can be drawn instead of a complete ellipse. Drawing starts at the turtle's starting position, and continues throughout *angle* degrees around the perimeter of the ellipse.

The optional parameter *clockwise* determines whether the turtle will trace out the ellipse in a clockwise or anti-clockwise direction.

The turtle moves as it draws the ellipse. If a complete ellipse is drawn, the turtle will end up at its original starting position and heading. Otherwise, the turtle will move to the end of the elliptical arc and have a heading perpendicular to the curve at that point.

5.2 EXT.UNFILLED

```
to EXT.UNFILLED :instrlist
```

The *EXT.UNFILLED* command behaves specially when used with the SVG turtle backend. It can be used inside a *FILLED* instruction to indicate that an enclosed shape should not be filled.

When used with the TK GUI back end, the Logo instructions contained in *instrlist* are processed normally by the interpreter.

When used with the SVG turtle backend, a number of special rules apply. First, the extension can only be used within an enclosing *FILLED* command. When used within a *FILLED* command, the path(s) traversed are masked. The mask will prevent those areas from being filled.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search