Compact Python Universal Turing Machine Simulator

By David Hartkop

Abstract

This Python script lets you program and play with your very own Turing Machines. It works well for very simple examples, and allows students to see their results immediately. The simulator, however, is vastly expandable: Using Python 3, there is actually *no specific limit* to the number of machine states, rules per state, or number of symbols in the alphabet. Your simulation is limited only by the physical system memory on your computer.

How is the universal Turing Machine represented? Jump to an example on Pg.7, or keep reading for a more thorough explanation:

This simulator represents a classic single-tape Turing Machine architecture with the following parts:

1. Tape memory
2. Control memory
3. Control system

Part 1. Tape memory

The tape memory of a Turing Machine is analogous to the data memory in a modern computer. It is a volume of memory arranged in discreet positions, each with an address. Data representing 'symbols' can be read *from* each memory position, or may be written *to* each. In Python, the tape memory is represented as a simple list of elements:

```
TapeMem = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The example shows a 'tape' of 10 elements, and an alphabet of 10 symbols. Turing's theoretical universal machine assumed a memory with an infinite number of elements, and an infinite number of symbols. Provided that your computer has enough memory, a list of this type can have nearly infinite element count. This simulator uses only integers as symbols, but since Python 3, there is no upward limit to the value of an integer either.

The ’tape memory’ is accessed in a linear fashion by moving the ‘tape head’ to the LEFT or to the RIGHT by one position. For example, let’s say the head is located at tape position TapeMem[3]:

```
TapeMem = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                    ^
                   HEAD
```

If the Turing Machine is directed to move the tape head LEFT, then the tape memory will then be read at TapeMem[2]:

```
TapeMem = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                 ^
                HEAD
```

When setting up the simulator, it is important to set up your TapeMem’s content and the start position. Specify the value and number of all elements in the list TapeMem, and then enter a value for the playheadPosition variable. These values depend entirely on the Turing machine program that is being run. This is an example setup:

```
TapeMem = [8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8]

playheadPosition = 1
```

Part 2. Control memory

The control memory of a Turing Machine is analogous to the program memory of a modern computer. The control memory for this simulator is organized as an array of 'states', each of which contain some number of 'rules', each rule containing four 'instruction elements'. For example, if your Turing Machine has three state, you could write the control memory like this:

```
(state 0)

(state 1)

(state 2)
```

Each state can contain an arbitrary number of rules. If you could open up the states to see the rules inside, they might look like this:

```
(state 0)  -----opened---->  ((rule 0), (rule 1), (rule 2))
(state 1)  -----opened---->  ((rule 0), (rule 1))
(state 2)  -----opened---->  ((rule 0), (rule 1), (rule 2), (rule 3))
```

If we were to look inside each rule, each would show four numbers, neatly separated by commas. A fully revealed control memory might look like this:

```
((7, 5, 0, 2), (1, 5, 1, 1), (7, 5, 0, 2))
((5, 5, 0, 2), (2, 5, 1, 0))
((3, 5, 1, 2), (0, 5, 0, 2), (6, 5, 1, 5), (4, 5, 0, 2))
```

Let's just look at the *first* rule in the *first* state. In python, this rule would be addressed like this:

```
ControlMem[0][0]    -----addresses this---->   (7, 5, 0, 2)
```

The four numbers inside of each rule are actually the four 'instruction elements' that are used by the Turing Machine's control system when the machine is operating. The position of each of the four numbers is important. For instance, this is a translation of the instruction elements in the above rule:

```
The control system 'understands' (7,5,0,2) to mean:
"If the tape head reads the number 7,
then over-write it with number 5 at current tape position,
then move the tape head LEFT (0 means left, 1 means right),
and finally, change the Turing Machine to STATE 2."
```

Another example translation of Turing Machine instruction elements is this:

```
The rule (4,0,1,3) means, "If tape head reads 4, change it to 0, move
tape head right, go to state 3."
```

The above examples are simplified to show the concept of how the control memory is organized. In Python, this data structure can be written as a set of carefully nested lists or tuples, like this:

```
ControlMem = (\
((0, 1, 1, 0), (8, 8, 0, 1), (2, 3, 1, 0), (4, 4, 0, 2)), \
((1, 2, 0, 1), (8, 8, 1, 0), (3, 4, 0, 1)), \
((),)\
)
```

In this example control memory, there are three states, enclosed like this: (() () ()), and split to be on 3 different lines. The first state has four rules, (each rule is a set of 4 numbers). The second state has three rules. The third state has no rules at all, and will be treated as a 'halt condition'. This brings the simulation to a stop.

When trying different Turing Machines, or designing your own, you can set up the control memory to have as many state as you like, and each state can have as many rules as you like. Each rule, however will contain exactly four instruction elements, which will always be translated by the control system in the same way as shown above.

Regardless of the size of your Turing Machine, the instruction elements of any rule can be accessed the same way. For example, if we wish to display *State #1*, *Rule #2*, we would run this Python code:

```
Type this:            print(ControlMem[1][2])
python prints this:   (3, 4, 0, 1)
```

Note: The Python simulator uses integers as the symbols instruction elements. If you wish to use other symbols, you may use strings, but each instruction element must be then enclosed in quotes. I would recommend only doing this to the first two instruction elements, and leaving the latter two alone:

(“X”, “N”, 0, 1)   <---- *don't mess with the last two elements' data types!*

This is because the python script is set up to recognize a 0 or 1 for head direction, and directly uses the last instruction number in the indexing process.

Part 3. Control system

The control system of this Turing Machine simulator consists of a while loop that runs until a 'halt condition' is reached. The most basic halt condition built into this simulation is when the Turing machine enters a state that has zero instructions. The system recognizes that nothing further will happen *(because there are no further instructions)* and halts. There is also a built-in cycle counter in the Python code that lets you specify the maximum number of cycles the simulation is allowed to run. This is handy when running Turing machines that may never halt .

You can specify the maximum cycles allowed by entering a number of cycles into the variable "cycles". For instance, to allow 100 cycles, set the variable at the top of the script as follows:

```
cycles = 100
```

To allow for infinite cycles, just comment-out (#) the cycle incrementing at the bottom of the script as follows:

```
#cyclecount = cyclecount+1
```

The control system runs through four distinct operations, already alluded to when we discussed the four instruction elements in each instruction above. The operations are as follows:

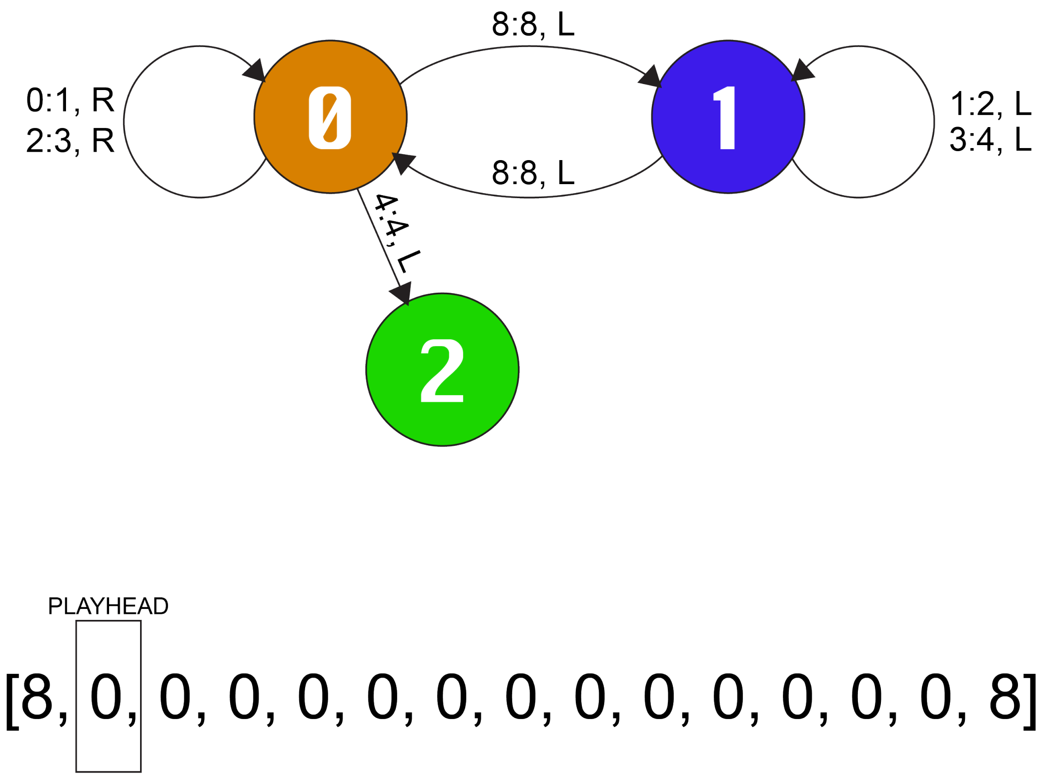
```
1. READ playhead
2. WRITE new value to playhead
3. MOVE the playhead (L or R)
4. Change the STATE
```

Each of the operations is informed by the instruction element found in the relevant rule in the current state of the machine. While this writeup does not cover the control system line-by line, it should be noted that each of the control operations is demarcated by a boldfaced comment, for example:

```
#STEP 01:READ playhead into currentPlayheadRead
#STEP 02:Write new value to playhead according to control memory
#STEP 03: move the playhead (L or R)
#STEP 04: change the state according to the control memory
```

Putting it all together

Traditionally, a Turing machine program is represented graphically in the form of a Turing Diagram. Here is an example Turing diagram for the example machine built into the simulation:

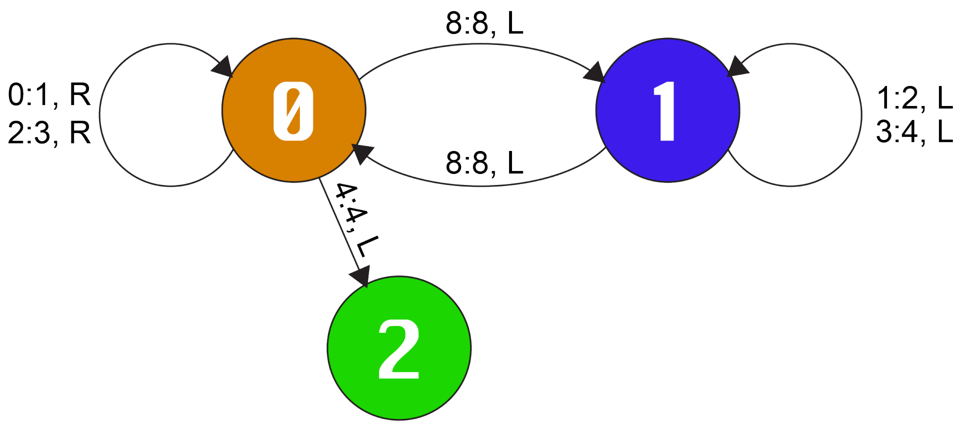


This diagram shows a Turing Machine with three states (the colored bubbles numbered 0, 1, and 2). There are arrows attached to the various states. Each arrow can be though of as a conditional statement that leads away from a given state. For instance, look above at the arrow connecting state 0 to state 1; it bears the label, “8:8, L”.   This conditional statement means, “If the tape head reads an 8, replace it with 8, then move the tape head left, then follow the arrow to enter into machine state 1.”

In addition to the diagram, a Turing Machine requires some preset data on the 'tape' and a specific starting location; this is shown below the diagram in [brackets].

In order to use the Python simulator, you must first translate a Turing diagram into the proper structure of states, rules, and tape memory. As a useful example, the following page shows the given example above, along with its translation into Python for this simulator:

Example Classic Turing Machine diagram:



PLAYHEAD

[8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8]

Example Turing Machine diagram translated for the Python simulator:

```
ControlMem = (\
((0, 1, 1, 0), (8, 8, 0, 1), (2, 3, 1, 0), (4, 4, 0, 2)), \
((1, 2, 0, 1), (8, 8, 1, 0), (3, 4, 0, 1)), \
((),)\
)
TapeMem = [8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8]
playheadPosition = 1
```

I hope that this writeup helps and along with the Turing machine simulator provide a fun and engaging way to further explore the fundamentals of computer science with the most famous of all universal computing architectures, the Turing Machine.

Sincerely,

David Hartkop :-)

```
Some example Turing Machines that run in the simulator:
----------------------------------------------------------------------

#Name: Blank Turing Machine Template
#Description: Blank placeholder with a tape of 12 zeros and a control memory
of 3 states. The first two states have 3 rules, and the third is a HALT state
containing no rules.

TapeMem = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

playheadPosition = 0

ControlMem = (\
((0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)), \
((0, 0, 0, 0), (0, 0, 0, 0), (0, 0, 0, 0)), \
((),)\
)
----------------------------------------------------------------------

#Name: Simple Zig-Zag
#Description: Writes zeros and 1s back and forth between a pair of 2s.

TapeMem = [2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]

playheadPosition = 1

ControlMem = (\
((1, 0, 1, 0), (2, 2, 0, 1)), \
((0, 1, 0, 1), (2, 2, 1, 0)), \
)
----------------------------------------------------------------------

#Name: Number Counting Zig-Zag
#Description: Writes increasing numbers back and forth between a pair of 8s.
When it reaches 4, the program halts.

TapeMem = [8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8]

playheadPosition = 1

ControlMem = (\
((0, 1, 1, 0), (8, 8, 0, 1), (2, 3, 1, 0), (4, 4, 0, 2)), \
((1, 2, 0, 1), (8, 8, 1, 0), (3, 4, 0, 1)), \
((),)\
)
----------------------------------------------------------------------

#Name: Binary Count-Up
#Description: Counts up in binary starting with initial zero. The new binary
number should be read each time the playhead lands on the number 3.

TapeMem = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 3]

playheadPosition = 10

ControlMem = (\
((1, 0, 0, 0), (2, 1, 1, 1), (0, 1, 1, 0), (3, 3, 0, 0)), \
((1, 1, 1, 1), (0, 0, 1, 1), (2, 2, 1, 0), (3, 3, 0, 0)), \
)
```

```
#Name: Unary to binary converter
#Description:  tallies a row of 1's and writes count in binary. For example:
```

The four number 1's on this row:

[5, 5, 5, 5, 1, 1, 1, 1, 5, 4] 0

Are converted to the number 4 in binary (101) in this row:

[5, 1, 0, 1, 3, 3, 3, 3, 5, 4] 41

TapeMem = [5, 5, 5, 5, **1**, 1, 1, 1, 5, 4]

```
playheadPosition = 4

ControlMem = (\

((2, 2, 1, 0), (3, 3, 1, 0), (1, 3, 0, 1), (5, 5, 0, 2)), \

((5, 4, 1, 0), (2, 4, 1, 0), (3, 3, 0, 1), (4, 2, 0, 1)), \

((3, 3, 0, 2), (2, 0, 0, 2), (4, 1, 0, 2), (5, 5, 0, 3)), \

((),)\

)
----------------------------------------------------------------------
#Name: Binary numbers adder:
#Description: Takes in two binary numbers seperated by 2 and writes their
sum. For example:
```

The two numbers to add are underlined in the start tape condition:

[2, 2, 2, 2, 1, 0, 0, 1, 2, 1, 0, 1, 2, 2, 2, 2]   9 + 5 = ?

And they are added together in the halt condition:

[2, 2, 2, 2, 1, 1, 1, 0, 2, 0, 0, 0, 2, 2, 2, 2]   = 14

TapeMem = [2, 2, 2, 2, 1, 0, 0, 1, 2, 1, 0, **1**, 2, 2, 2, 2]

```
playheadPosition = 11

ControlMem = (\

((0, 1, 0, 0), (1, 0, 0, 1)), \

((0, 0, 0, 1), (1, 1, 0, 1), (2, 2, 0, 2)), \

((1, 0, 0, 2), (2, 1, 1, 3), (0, 1, 1, 3)), \

((0, 0, 1, 3), (1, 1, 1, 3), (2, 2, 1, 4)), \

((0, 0, 1, 4), (2, 2, 1, 6), (1, 1, 1, 5)), \

((0, 0, 1, 5), (1, 1, 1, 5), (2, 2, 0, 0)), \

((),)\

)
```