Turing machines

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

A Turing Machine (TM) is an idealised model of a computer. Whereas real computers have finite memory, a TM has infinite memory, represented as a *tape* of consecutive *cells*, each of which contains either the *symbol* 0 or the symbol 1, that extends infinitely both left and right. To represent a finite section of the tape of a TM, a particular kind of **object**, namely, a Python **list**, is appropriate. We just need to define a Python list whose elements are the **integers** 0 and 1, that Python also represents as objects. We can then let a **variable**, say `tape`, denote what the list **literal** `[0, 0, 1, 1, 1, 0]` itself denotes, namely, a sequence of bits stored in computer memory meant to be interpreted as the Python list that consists of the integers 0, 0, 1, 1, 1 and 0. This is a particular kind of **statement**: an **assignment**. It involves two **expressions**: the list literal and the variable, the list literal itself involving many occurrences of two expressions: `0` and `1`. Expressions are to be **evaluated**; the result of the evaluation is a mathematical entity, the expression's **value**. Statements are to be **executed**. In the following cell, on the line that precedes the statement `tape = [0, 0, 1, 1, 1, 0]` and on the line of that statement, following it, are two **comments**. When running the cell, `tape = [0, 0, 1, 1, 1, 0]` is executed. Then `tape` is evaluated and a "standard" expression is output whose value is the same as `tape`'s value; that expression happens to be the list literal that `tape` has been assigned to:

[1]:
```
# 6 consecutive cells of a TM
tape = [0, 0, 1, 1, 1, 0] # Some cells contain 0, others contain 1
tape
```

[1]:
```
[0, 0, 1, 1, 1, 0]
```

To more conveniently create a list with a repeating pattern (e.g., a list consisting of nothing but 0's, or a list consisting of nothing but 1's, or a list where 0's and 1's alternate), one can use the `*` binary **operator**. When both operands are numbers, `*` is multiplication. When running the following cell, the expression `3*5` evaluates to the number 15, and `15`, a more "standard" expression that also evaluates to 15, is output:

[2]:
```
3 * 5
```

[2]:
```
15
```

When one operand is a list L and the other operand is a positive integer n, `*` duplicates L n many times. Again, "standard" expressions are output whose values are the same as those of the expressions that make up all lines in the following cell, the comment being ignored:

[3]:
```
[0] * 0 # The empty list
1 * [0]
```

```
[1] * 7
5 * [0, 1]
```

[3]: `[]`

[3]: `[0]`

[3]: `[1, 1, 1, 1, 1, 1, 1]`

[3]: `[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]`

One can refer to individual members of a list via their **index**, in relation to an increasing enumeration of its elements from left to right, starting with 0, or in relation to a decreasing enumeration of its elements from right to left, starting with -1. To illustrate this claim, we use the `print()` **function**. This function takes an arbitrary number of **arguments**, and outputs "standard" expressions whose values are the same as those of the arguments; by default, in the output, two consecutive expressions are separated by a space. In both invocations of `print()` in the cell below, `print()` receives 6 arguments. These invocations are statements, which are executed when running the cell, and the arguments to `print()` are displayed as "standard" expressions with the same values as a side effect:

[4]:
```
print(tape[0], tape[1], tape[2], tape[3], tape[4], tape[5])
print(tape[-1], tape[-2], tape[-3], tape[-4], tape[-5], tape[-6])
```

```
0 0 1 1 1 0
0 1 1 1 0 0
```

So the positive index of the last element of a nonempty list is equal to the length of the list minus 1, and the negative index of the first element of a nonempty list is equal to minus the length of the list. The length of a list is what the `len()` function returns when it receives (an expression that evaluates to that) list as argument:

[5]:
```
len(tape)
```

[5]: `6`

Using an index which is at least equal to the length of the list generates an `IndexError` **exception**:

[6]:
```
tape[len(tape)]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-6-57e58a86054f> in <module>
----> 1 tape[len(tape)]

IndexError: list index out of range
```

Using an index which is smaller than minus the length of the list also generates an `IndexError` exception. The code in the following cell makes use of two operators: unary `-`, which negates its

unique operand, and binary `-`, which subtracts its second operand from its first operand, the former operator taking precedence over the latter:

[7]:
```
tape[-len(tape) - 1]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-98a523e8b54d> in <module>
----> 1 tape[-len(tape) - 1]

IndexError: list index out of range
```

Let us define a function to nicely display what `tape` represents on demand. To start with, we just design the function. We let the **body** of the **function definition** consist of nothing but comments that describe what the function is meant to do. Running the contents of the following cell shows that Python does not accept this function definition as such:

[8]:
```
def display_tape():
    # Draw a horizontal line to represent
    #   the top boundary of the tape fragment.
    # Draw the tape fragment's cell contents,
    #   also drawing vertical line segments as cell boundaries.
    # Draw a horizontal line to represent
    #   the bottom boundary of the tape fragment.
```

```
  File "<ipython-input-8-35410398860a>", line 7
    #   the bottom boundary of the tape fragment.
                                                 ^
IndentationError: expected an indented block
```

The body of a function definition should consist of at least one statement. We can make `pass` the only statement. The function definition is now acceptable (though it is only provisional, as we have not **implemented** the function so that it behaves as **specified** by the comments in the function body):

[9]:
```
def display_tape():
    # Draw a horizontal line to represent
    #   the top boundary of the tape fragment.
    # Draw the tape fragment's cell contents,
    #   also drawing vertical line segments as cell boundaries.
    # Draw a horizontal line to represent
    #   the bottom boundary of the tape fragment.
    pass
```

When **called**, a function **returns** a value. An expression having that value, the function call itself being such an expression, can be assigned to a variable. A `return` statement allows one to explicitly

let a function return a value:

[10]:
```
def f():
    return 2

f() # Returns 2, as shown when running the cell
x = f()
print(x)
```

[10]:
```
2
```

```
2
```

A function that does not eventually execute a `return` statement still returns some value, namely, the special value that the "standard" expression `None` evaluates to:

[11]:
```
display_tape() # Returns None, which is not shown when running the cell
x = display_tape()
print(x)
```

```
None
```

A function can also explicitly return (the value that is the result of evaluating) `None`:

[12]:
```
def f():
    return None

f() # Returns None, which is not shown when running the cell
x = f()
print(x)
```

```
None
```

Getting back to the comments in `display_tape()`'s body, we see that we have to twice draw a line. This can be achieved by printing out a (representation of a) **string** of hyphens. More generally, a string is a sequence of **characters**. String literals can be delimited with single quotes; a backslash allows one to escape a single quote and make it part of the string:

[13]:
```
string = 'A string with \' (single quote), not ", as delimiter'
string
print(string)
```

[13]:
```
'A string with \' (single quote), not ", as delimiter'
```

```
A string with ' (single quote), not ", as delimiter
```

Alternatively, string literals can be delimited with double quotes; then double quotes, not single quotes, need to be escaped to be part of the string:

[14]:
```
string = "A string with \" (double quote), not ', as delimiter"
string
print(string)
```

[14]:
```
'A string with " (double quote), not \', as delimiter'
```

```
A string with " (double quote), not ', as delimiter
```

String literals can span many lines: just use either three single quotes or three double quotes as delimiters. One could escape new line characters and define those string literals with single or double quotes as delimiters, but they would not read as well:

[15]:
```
string = '''A string containing both ' and "
with \''' (triple quote) as delimiter;
it actually contains four single quotes,
one of which is escaped'''
string
print(string)
```

[15]:
```
'A string containing both \' and "\nwith \'\'\' (triple quote) as delimiter;\nit
actually contains four single quotes,\none of which is escaped'
```

```
A string containing both ' and "
with ''' (triple quote) as delimiter;
it actually contains four single quotes,
one of which is escaped
```

[16]:
```
string = """
A string containing both ' and "
delimited with triple quotes
(observe: 4 spaces at the end of the previous line)
and containing five new line characters
"""
string
print(string)
```

[16]:
```
'\nA string containing both \' and "\ndelimited with triple quotes
\n(observe: 4 spaces at the end of the previous line)\nand containing five new
line characters\n'
```

```
A string containing both ' and "
delimited with triple quotes
(observe: 4 spaces at the end of the previous line)
and containing five new line characters
```

The * operator can also be used with a string and a positive integer as operands:

[17]:
```
# The empty string
'acb' * 0
1 * 'abc'
'abc' * 2
3 * 'abc'
```

[17]: ''

[17]: 'abc'

[17]: 'abcabc'

[17]: 'abcabcabc'

Since `display_tape()` has to twice draw the same line, it is preferable not to duplicate code and define an auxiliary function, say `draw_horizontal_line()`, to draw that line, and let `display_tape()` call `draw_horizontal_line()` twice. As `display_tape()`, the function `draw_horizontal_line()` takes no argument. In the function body, `tape` is used as a **global** variable: `tape` has been **declared** outside the function body, but `tape`'s value can still be retrieved in the function body. We define the function and then call it:

[18]:
```
def draw_horizontal_line():
    # multiplication takes precedence over addition
    print('-' * (2 * len(tape) + 1))

draw_horizontal_line()
```

```
-------------
```

We can now partially implement `display_tape()`, removing the `pass` statement, and replacing the first and last comments in its body with calls to `draw_horizontal_line()`:

[19]:
```
def display_tape():
    draw_horizontal_line()
    # Output cell contents, delimited with |'s.
    draw_horizontal_line()

display_tape()
```

```
-------------
-------------
```

To complete the implementation of `display_tape()`, we need to write code that outputs a string consisting of the characters `'|'`, `'0'` and `'1'`. From `tape`, a (variable that evaluates to a) list consisting of the integers (that are the values of) `0` and `1`, we could obtain a corresponding list consisting of the characters (that are the values of) `'0'` and `'1'`, letting `str()` convert numbers to strings, and making use of a **list comprehension**. The following expression reads as: the list of all elements of the form `str(symbol)` where `symbol` ranges over `tape`, from beginning to end:

[20]:
```
[str(symbol) for symbol in tape]
```

[20]: `['0', '0', '1', '1', '1', '0']`

One could then use a particular function, the `join()` **method** of the `str` **class**, to create a string from the (one character) strings in the former list. If `join()` is applied to the empty string, then all those characters are "glued" together:

[21]:
```
''.join([str(symbol) for symbol in tape])
```

[21]:
```
'001110'
```

We could "glue" the characters with any other string:

[22]:
```
'+A+'.join([str(symbol) for symbol in tape])
```

[22]:
```
'0+A+0+A+1+A+1+A+1+A+0'
```

This is what we want:

[23]:
```
'|'.join([str(symbol) for symbol in tape])
```

[23]:
```
'0|0|1|1|1|0'
```

The previous expression is correct, but not as good as it could and should be. Indeed, the expression is evaluated by processing all elements that make up (the list that is the value of) `tape` to create the list (that is the value of) `[str(symbol) for symbol in tape]`, and then processing all elements that make up that second list to create the desired string. A better option is to use a **generator expression**:

[24]:
```
(str(symbol) for symbol in tape)
```

[24]:
```
<generator object <genexpr> at 0x10b80f190>
```

A generator expression is a potential sequence of elements. One way to actualise the sequence and retrieve each of its members, one by one, is to call the `next()` function; when all elements have been retrieved, a new call to `next()` generates a `StopIteration` exception:

[25]:
```
E = (str(symbol) for symbol in tape)
next(E)
next(E)
next(E)
next(E)
next(E)
next(E)
next(E)
```

[25]:
```
'0'
```

[25]:
```
'0'
```

[25]:
```
'1'
```

[25]:
```
'1'
```

[25]:
```
'1'
```

[25]:
```
'0'
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-25-7f0fd1280c99> in <module>
      6 next(E)
      7 next(E)
----> 8 next(E)

StopIteration:
```

Any (expression that evaluates to a) generator expression can be passed as an argument to `list()` which behind the scene, calls `next()` until `StopIteration` is generated, letting it gracefully complete the construction of the list:

[26]:
```
# The full syntax would be list((str(symbol) for symbol in tape)),
# but Python lets us  simplify it and omit one pair of parentheses.
list(str(symbol) for symbol in tape)
```

[26]:
```
['0', '0', '1', '1', '1', '0']
```

`join()` also accepts a generator expression rather than a list as argument. In the code below, `join()` processes the `'0'`'s and `'1'`'s it receives from the generator expression `(str(symbol) for symbol in tape)`, as that generator expression processes the `0`'s and `1`'s that make up the list `tape`. The desired string is created "on the fly"; no intermediate list is created:

[27]:
```
'|'.join(str(symbol) for symbol in tape)
```

[27]:
```
'0|0|1|1|1|0'
```

We also want to display a vertical bar at both ends. This can be done by concatenating three strings into one. For that purpose, one can use the `+` binary operator. When both operands are numbers, `+` is addition, but when both operands are strings, `+` is concatenation:

[28]:
```
'ABC' + 'DEF'
```

[28]:
```
'ABCDEF'
```

`+` is left associative. Therefore, in the following statement, the first occurrence of `+` creates a new string S from its operands, and the second occurrence of `+` creates a new string from S and its second operand:

[29]:
```
'|' + '|'.join(str(symbol) for symbol in tape) + '|'
```

[29]:
```
'|0|0|1|1|1|0|'
```

Since our aim is only to display a sequence of characters, we do not need to create a new string from three strings: we can instead let `print()` take those three strings as arguments and print them out changing the separator from the default, a space, to an empty string, using the optional **keyword only parameter** `sep` to `print()`. Compare both calls to `print()` that follow, both of which takes 3 **positional arguments**, the second of which moreover takes a **keyword argument**:

[30]:
```
print('|', '|'.join(str(symbol) for symbol in tape), '|')
print('|', '|'.join(str(symbol) for symbol in tape), '|', sep='')
```

```
| 0|0|1|1|1|0 |
|0|0|1|1|1|0|
```

We now have the full implementation of `display_tape()`:

[31]:
```
def display_tape():
    draw_horizontal_line()
    print('|', '|'.join(str(e) for e in tape), '|', sep='')
    draw_horizontal_line()

display_tape()
```

```
-------------
|0|0|1|1|1|0|
-------------
```

Besides a tape, a TM has a *head*, that can be positioned below one of the cells that make up the tape, thereby revealing its contents to the TM. A TM does not have a global view of the tape, it only has a very local view, that of a single cell, but the contents of that cell changes over time as the TM moves its head left or right. A TM also has a *program* to perform some computation. We assume that before computation starts, the tape has been "initialised" in such a way that only a finite number of cells contain the symbol 1, some cell contains 1, and the head is positioned below the cell that contains the leftmost 1. The section of the tape that spans between the cells that store the leftmost and rightmost 1's is meant to encode some data (numbers, text, images, videos…).

At any stage of the computation, including before it starts and when it ends, if it ever comes to an end, the TM is in one of a finite number of *states*. The program of a TM consists of a finite set of *instructions*, each instruction being a quintuple of the form (*state*, *symbol*, *new_state*, *new_symbol*, *direction*), with the following intended meaning: if the current state of the TM is *state*, and if the head of the TM is currently positioned under a cell that stores *symbol*, then the contents of that cell becomes *new_symbol* (which can be the same as *symbol*), the state of the TM becomes *new_state* (which can be the same as *state*), and the head of the TM moves one cell to the right or one cell to the left depending on whether *direction* is R or L. A TM is *deterministic*. This means that at any stage, at most one instruction can be executed: its program does not have two distinct instructions that both start with the same first two elements, with the same state and symbol. Computation runs for as long as some instruction can be executed. Either there is always such an instruction, in which case computation never terminates, or at some stage no instruction can be executed, in which case computation ends.

For illustration purposes, assume that `tape` is set to `[0, 0, 1, 1, 1, 1, 0]` and represents a segment of the tape that contains all 1's on the tape. Suppose that the head of the TM is positioned below the cell that stores the leftmost 1, corresponding to the member of `tape` of index 2, as it is meant to be before computation starts. Suppose that there are two possible states, green and blue. Also suppose that the initial state, that is, the state of the TM before computation starts, is green. If the program of the TM has no instruction whose first two members are green and 1, then computation stops. One the other hand, if the program has one such instruction, then that instruction is one of the following 8 instructions, and the TM executes it:

- (green, 1, green, 1, R): the state of the TM remains green, `tape` is left unchanged, and the head of the TM moves right (so positions itself below the cell that corresponds to `1` at index 3 in `tape`).
- (green, 1, green, 1, L): the state of the TM remains green, `tape` is left unchanged, and the head of the TM moves left (so positions itself below the cell that corresponds to `0` at index 1 in `tape`).
- (green, 1, green, 0, R): the state of the TM remains green, the `1` in `tape` at index 2 is changed to `0`, and the head of the TM moves right (so positions itself below the cell that corresponds to `1` at index 3 in `tape`).
- (green, 1, green, 0, L): the state of the TM remains green, the `1` in `tape` at index 2 is changed to `0`, and the head of the TM moves left (so positions itself below the cell that corresponds to `0` at index 1 in `tape`).
- (green, 1, blue, 1, R): the state of the TM changes to blue, `tape` is left unchanged, and the head of the TM moves right (so positions itself below the cell that corresponds to `1` at index 3 in `tape`).
- (green, 1, blue, 1, L): the state of the TM changes to blue, `tape` is left unchanged, and the head of the TM moves left (so positions itself below the cell that corresponds to `0` at index 1 in `tape`).
- (green, 1, blue, 0, R): the state of the TM changes to blue, the `1` in `tape` at index 2 is changed to `0`, and the head of the TM moves right (so positions itself below the cell that corresponds to `1` at index 3 in `tape`).
- (green, 1, blue, 0, L): the state of the TM changes to blue, the `1` in `tape` at index 2 is changed to `0`, and the head of the TM moves left (so positions itself below the cell that corresponds to `0` at index 1 in `tape`).

Intuitively, a state captures some memory of the past. If infinitely many states were available, it would be possible to remember everything that happened since computation started. As only finitely many states are available, states can usually keep track of only part of what happened; for instance, one usually cannot remember how many 1's have been overwritten with 0's since computation started, but finitely many states are enough to remember whether that number is even or odd.

We provide sample TM programs to compute various functions from $\mathbf{N}^*$ to $\mathbf{N}$, or from $\mathbf{N}^* \times \mathbf{N}^*$ to $\mathbf{N}$ ($\mathbf{N}$ denotes the set of natural numbers, and $\mathbf{N}^*$ the set of strictly positive natural numbers):

- the successor function, that maps n to $n + 1$
- the parity function, that maps n to 0 if n is even, and to 1 otherwise
- division by 2, that maps n to $\lfloor \frac{n}{2} \rfloor$
- addition
- multiplication

For the first three programs, data is for a single nonzero natural number, encoded in unary: 1 is encoded as 1, 2 as 11, 3 as 111, 4 as 1111… For the last two programs, data is for two nonzero natural numbers, both encoded in unary, and separated by a single 0.

For all programs, when computation stops, data is "erased" and the tape just stores the natural number r that is the result of the computation, represented in unary.

- If $r > 0$, the head of the TM is positioned under the cell that stores the leftmost 1.
- If $r = 0$, the tape contains nothing but 0's and the head is positioned anywhere on the tape.

These TM programs are saved in the files:

- `successor.txt`
- `parity.txt`
- `division_by_2.txt`
- `addition.txt`
- `multiplication.txt`

The program `turing_machine_simulator.py` creates a widget to experiment with those programs and others. It has a Help menu. Rather than representing the head of the TM in one way or another, it identifies the cell that the head is currently positioned under by displaying its contents in boldface red.

Let us now examine how to process the contents of a file containing the program of a TM, say `division_by_2.txt`. The `open()` function returns a handle to a file, which we can make the value of a variable on which we can then operate to read and extract the contents of the file, before eventually closing it. `open()` expects to be given as argument a string that represents the location of the file. Using the name of the file for the string makes the location relative to the **working directory**, which is fine here since `division_by_2.txt` is indeed stored in the working directory. By default, `open()` works in reading mode; this is appropriate since we do not want to overwrite or modify `division_by_2.txt`:

[32]:
```
TM_program_file = open('division_by_2.txt')
TM_program_file
# Operate on TM_program_file to read and process
# the contents of division_by_2.txt.
TM_program_file.close()
```

[32]:
```
<_io.TextIOWrapper name='division_by_2.txt' mode='r' encoding='UTF-8'>
```

The previous syntax is simple, but it is preferable to opt for an alternative and use a **context manager** and an associated `with ... as` expression, that gracefully closes the file after it has been processed, or earlier in case problems happen during processing:

[33]:
```
with open('division_by_2.txt') as TM_program_file:
    TM_program_file
    # Operate on TM_program_file to read and process
    # the contents of division_by_2.txt
```

[33]:
```
<_io.TextIOWrapper name='division_by_2.txt' mode='r' encoding='UTF-8'>
```

The contents of the file can be extracted all at once, as a list of strings, one string per line in the file:

[34]:
```
with open('division_by_2.txt') as TM_program_file:
    TM_program_file.readlines()
```

[34]:
```
['# Initial state: del1\n',
 '\n',
 'del1 1 del2 0 R\n',
```

```
 'del2 1 mov1R 0 R\n',
 'mov1R 1 mov1R 1 R\n',
 'mov1R 0 mov2R 0 R\n',
 'mov2R 1 mov2R 1 R\n',
 'mov2R 0 mov1L 1 L\n',
 'mov1L 1 mov1L 1 L\n',
 'mov1L 0 mov2L 0 L\n',
 'mov2L 1 mov2L 1 L\n',
 'mov2L 0 del1 0 R\n',
 'del1 0 end 0 R\n',
 'del2 0 end 0 R\n']
```

When dealing with very large files, storing the whole file contents as a list of strings can be too ineffective. Instead, lines can be read one by one on demand with the `readline()` method:

[35]:
```
with open('division_by_2.txt') as TM_program_file:
    TM_program_file.readline()
    TM_program_file.readline()
    TM_program_file.readline()
    TM_program_file.readline()
```

[35]:
```
'# Initial state: del1\n'
```

[35]:
```
'\n'
```

[35]:
```
'del1 1 del2 0 R\n'
```

[35]:
```
'del2 1 mov1R 0 R\n'
```

In fact, `open()` returns an **iterator**, that is, an object of the same type as a generator expression, which `next()` can be applied to; essentially, `readline()` is just alternative syntax for `next()`:

[36]:
```
with open('division_by_2.txt') as TM_program_file:
    next(TM_program_file)
    next(TM_program_file)
    next(TM_program_file)
    next(TM_program_file)
```

[36]:
```
'# Initial state: del1\n'
```

[36]:
```
'\n'
```

[36]:
```
'del1 1 del2 0 R\n'
```

[36]:
```
'del2 1 mov1R 0 R\n'
```

Iterators are usually best processed with a `for` statement, a kind of **loop**. Behind the scene, `for` calls `next()` until the latter generates a `StopIteration` exception, which lets it gracefully exit the loop. Note that since every line of the file being processed yields a string that ends in a new line

character, and `print()` "goes to the next line" by default, that is, outputs a new line character, the output of the following code fragment shows one blank line between two consecutive lines:

[37]:
```
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        print(line)
```

```
# Initial state: del1



del1 1 del2 0 R

del2 1 mov1R 0 R

mov1R 1 mov1R 1 R

mov1R 0 mov2R 0 R

mov2R 1 mov2R 1 R

mov2R 0 mov1L 1 L

mov1L 1 mov1L 1 L

mov1L 0 mov2L 0 L

mov2L 1 mov2L 1 L

mov2L 0 del1 0 R

del1 0 end 0 R

del2 0 end 0 R
```

These blank lines can be eliminated from the output by changing the value of the keyword only parameter `end` of `print()` from the default new line character to an empty string:

[38]:
```
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        print(line, end='')
```

```
# Initial state: del1

del1 1 del2 0 R
del2 1 mov1R 0 R
mov1R 1 mov1R 1 R
mov1R 0 mov2R 0 R
```

```
mov2R 1 mov2R 1 R
mov2R 0 mov1L 1 L
mov1L 1 mov1L 1 L
mov1L 0 mov2L 0 L
mov2L 1 mov2L 1 L
mov2L 0 del1 0 R
del1 0 end 0 R
del2 0 end 0 R
```

We are only interested in lines that represent instructions, neither in lines that represent a comment nor by blank lines. For the former, the `startswith()` method of the `str` class is useful. It returns one of both **Boolean values** True and False:

[39]:
```
'A string'.startswith('')
'A string'.startswith('A')
'A string'.startswith('A ')
'A string'.startswith('A s')
'A string'.startswith('a')
```

[39]: True

[39]: True

[39]: True

[39]: True

[39]: False

For the latter, the `isspace()` method of the `str` class is useful:

[40]:
```
''.isspace()
' a'.isspace()
' '.isspace()
'   '.isspace()
# \t is for tab
'   \t   \n'.isspace()
'''



'''.isspace()
```

[40]: False

[40]: False

[40]: True

[40]: True

[40]: True

[40]: True

Using `startswith()` and `isspace()` as part of a **Boolean expression** that makes up the **condition** of an `if` statement, a kind of **test**, whose body is executed if and only if the condition evaluates to (the value of the "special" expression) `True` rather than (the value of the "special" expression) `False`, we can output only lines that represent instructions. The Boolean expression makes use of two **logical operators**, namely, **not** and **and**, for *negation* and *conjunction*, respectively:

[41]:
```
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        if not line.startswith('#') and not line.isspace():
            print(line, end='')
```

```
del1 1 del2 0 R
del2 1 mov1R 0 R
mov1R 1 mov1R 1 R
mov1R 0 mov2R 0 R
mov2R 1 mov2R 1 R
mov2R 0 mov1L 1 L
mov1L 1 mov1L 1 L
mov1L 0 mov2L 0 L
mov2L 1 mov2L 1 L
mov2L 0 del1 0 R
del1 0 end 0 R
del2 0 end 0 R
```

Alternatively, we can test the negation of the condition of the `if` statement in the previous code fragment (applying one of *de Morgan's laws* to get a *disjunction* from a conjunction as well as applying *double negation elimination*) and use a `continue` statement not to process any further any line that is not of interest. The Boolean expression then makes use of the logical operator **or**, for (*inclusive* as opposed to *exclusive*) disjunction.

[42]:
```
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        if line.startswith('#') or line.isspace():
            continue
        print(line, end='')
```

```
del1 1 del2 0 R
del2 1 mov1R 0 R
mov1R 1 mov1R 1 R
mov1R 0 mov2R 0 R
mov2R 1 mov2R 1 R
mov2R 0 mov1L 1 L
mov1L 1 mov1L 1 L
mov1L 0 mov2L 0 L
mov2L 1 mov2L 1 L
```

```
mov2L 0 del1 0 R
del1 0 end 0 R
del2 0 end 0 R
```

After an instruction has been retrieved, it is necessary to isolate its 5 components. The `split()` method of the `str` class is useful. We can pass to `split()` a nonempty string as argument:

[43]:
```
'aXaaXaaaXaaaXaaXaX'.split('a')
'aXaaXaaaXaaaXaaXaX'.split('aa')
'aXaaXaaaXaaaXaaXaX'.split('aaa')
'aXaaXaaaXaaaXaaXaX'.split('aaaa')
```

[43]: `['', 'X', '', 'X', '', '', 'X', '', '', 'X', '', 'X', 'X']`

[43]: `['aX', 'X', 'aX', 'aX', 'XaX']`

[43]: `['aXaaX', 'X', 'XaaXaX']`

[43]: `['aXaaXaaaXaaaXaaXaX']`

If no argument is passed to `split()`, then any longest sequence of space characters will play the role of separator, and any leading or trailing sequence of space characters will be ignored:

[44]:
```
'  \n\t  X X\tX\nX   \t  \n   X'.split()
```

[44]: `['X', 'X', 'X', 'X', 'X']`

So we can now get from each instruction a list of 5 strings:

[45]:
```
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        if line.startswith('#') or line.isspace():
            continue
        print(line.split())
```

```
['del1', '1', 'del2', '0', 'R']
['del2', '1', 'mov1R', '0', 'R']
['mov1R', '1', 'mov1R', '1', 'R']
['mov1R', '0', 'mov2R', '0', 'R']
['mov2R', '1', 'mov2R', '1', 'R']
['mov2R', '0', 'mov1L', '1', 'L']
['mov1L', '1', 'mov1L', '1', 'L']
['mov1L', '0', 'mov2L', '0', 'L']
['mov2L', '1', 'mov2L', '1', 'L']
['mov2L', '0', 'del1', '0', 'R']
['del1', '0', 'end', '0', 'R']
['del2', '0', 'end', '0', 'R']
```

A list of 5 strings is not the best representation of an instruction. Recall that since a TM is deterministic, no two distinct instructions share the same first two elements. A good way to put it is that the program of a TM is a function that maps a pair of the form (*state*, *symbol*) to a triple

of the form (*new_state*, *new_symbol*, *direction*): (*state*, *symbol*) represents a possible configuration (what is possibly the current state and the symbol stored in the cell that the head of the TM is currently positioned under), while (*new_state*, *new_symbol*, *direction*) represents what to do in case the computation is at a stage when that possible configuration happens to be the current one. To represent functions, mappings, Python offers **dictionaries**, that associate values to **keys**:

[46]:
```
# An empty dictionary
{}
# A dictionary with 4 keys, mapping names of digits to digits.
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
# A dictionary with 4 keys, mapping English words to German translations
{'tap': 'Hahn', 'table': 'Tisch', 'rooster': 'Hahn', 'train': 'Zug'}
# A dictionary with 3 keys, mapping German words to English translations
{'Hahn': ['tap', 'rooster'], 'Tisch': ['table'], 'Zug': ['train']}
```

[46]:
```
{}
```

[46]:
```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

[46]:
```
{'tap': 'Hahn', 'table': 'Tisch', 'rooster': 'Hahn', 'train': 'Zug'}
```

[46]:
```
{'Hahn': ['tap', 'rooster'], 'Tisch': ['table'], 'Zug': ['train']}
```

Observe how we mapped an English word to a German word, but a German word to a list of English words: this is because the English words under consideration all have a single German translation, whereas some of the German words under consideration have more than one English translation.

Getting back to our TM instructions, we could think of creating a dictionary `TM_program` that would have for each instruction of the form (*state*, *symbol*, *new_state*, *new_symbol*, *direction*), the list [*state*, *symbol*] as a key and the list [*new_state*, *new_symbol*, *direction*] as value for that key. That does not work:

[47]:
```
{['del1', '1']: ['del2', '0', 'R']}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-384215f30c9d> in <module>
----> 1 {['del1', '1']: ['del2', '0', 'R']}

TypeError: unhashable type: 'list'
```

The keys of a dictionary should be **immutable** objects, that is, objects that cannot change. It is possible to change a dictionary by adding or removing a key together with its associated value, but an existing key cannot be modified. A list is **mutable**, as it can be changed in many ways. It is for instance possible to change one of the members of a list:

[48]:
```
L = [10, 11, 12]
L[1] = 21
```

```
L
```

[48]: [10, 21, 12]

It is possible to add elements to a list:

[49]:
```
L = [10, 11, 12]
L.append(13)
L
```

[49]: [10, 11, 12, 13]

It is possible to remove elements from a list:

[50]:
```
L = [10, 11, 12]
L.remove(11)
L
```

[50]: [10, 12]

Tuples offer alternatives to lists to define sequences of values. Tuple literals are surrounded by parentheses rather than by square brackets. In many contexts, the parentheses are optional as commas are all what is needed to define a tuple:

[51]:
```
# The empty tuple
()
(10,)
10,
(10, 11)
10, 11
(10, 11, 12)
10, 11, 12
# Not a tuple, but 10 surrounded by parentheses
(10)
```

[51]: ()

[51]: (10,)

[51]: (10,)

[51]: (10, 11)

[51]: (10, 11)

[51]: (10, 11, 12)

[51]: (10, 11, 12)

[51]: 10

It is not possible to change the value of one of the members of a tuple:

[52]:
```
T = 10, 11, 12
T[1] = 21
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-52-0c5661d0cf37> in <module>
      1 T = 10, 11, 12
----> 2 T[1] = 21

TypeError: 'tuple' object does not support item assignment
```

It is not possible to add or remove elements to or from a tuple: there is no append nor remove method for tuples. Tuples can be dictionary keys. Dictionary values can be lists or tuples, only the keys should be immutable:

[53]:
```
{('del1', '1'): ['del2', '0', 'R']}
{('del1', '1'): ('del2', '0', 'R')}
```

[53]:
```
{('del1', '1'): ['del2', '0', 'R']}
```

[53]:
```
{('del1', '1'): ('del2', '0', 'R')}
```

Let us opt for representing the program of a TM as a dictionary where both keys and values are tuples. One can start with an empty dictionary and add a new key and its associated value every time a new instruction is processed:

[54]:
```
TM_program = {}
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        if line.startswith('#') or line.isspace():
            continue
        instruction = line.split()
        # Simplified syntax, without parentheses for keys and values.
        # The full syntax would be be:
        # TM_program[(instruction[0], instruction[1])] =\
        #         (instruction[2], instruction[3], instruction[4])
        # \ used for line continuation
        TM_program[instruction[0], instruction[1]] =\
                instruction[2], instruction[3], instruction[4]
TM_program
```

[54]:
```
{('del1', '1'): ('del2', '0', 'R'),
 ('del2', '1'): ('mov1R', '0', 'R'),
 ('mov1R', '1'): ('mov1R', '1', 'R'),
 ('mov1R', '0'): ('mov2R', '0', 'R'),
 ('mov2R', '1'): ('mov2R', '1', 'R'),
```

```
 ('mov2R', '0'): ('mov1L', '1', 'L'),
 ('mov1L', '1'): ('mov1L', '1', 'L'),
 ('mov1L', '0'): ('mov2L', '0', 'L'),
 ('mov2L', '1'): ('mov2L', '1', 'L'),
 ('mov2L', '0'): ('del1', '0', 'R'),
 ('del1', '0'): ('end', '0', 'R'),
 ('del2', '0'): ('end', '0', 'R')}
```

The previous code fragment is not as readable as it can be. Python lets us assign each element of a list or a tuple to each of the corresponding elements of a list or a tuple of the same length:

[55]:
```
[a1, b1, c1] = [10, 11, 12]
# Simplified syntax for what could also be written as
# (a2, b2, c2) = (21, 22, 23)
# or
# a2, b2, c2 = (21, 22, 23)
# or
# (a2, b2, c2) = 21, 22, 23
a2, b2, c2 = 21, 22, 23
# Simplified syntax for what could also be written as
# [a3, b3, c3] = (31, 32, 33)
[a3, b3, c3] = 31, 32, 33
# Simplified syntax for what could also be written as
# (a4, b4, c4) = [41, 42, 43]
a4, b4, c4 = [41, 42, 43]

[a1, b1, c1, a2, b2, c2, a3, b3, c3, a4, b4, c4]
# Simplified syntax for what could also be written as
# (a1, b1, c1, a2, b2, c2, a3, b3, c3, a4, b4, c4)
a1, b1, c1, a2, b2, c2, a3, b3, c3, a4, b4, c4
```

[55]:
```
[10, 11, 12, 21, 22, 23, 31, 32, 33, 41, 42, 43]
```

[55]:
```
(10, 11, 12, 21, 22, 23, 31, 32, 33, 41, 42, 43)
```

Though functionally equivalent to what we wrote above, the following code fragment is more readable:

[56]:
```
TM_program = {}
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        if line.startswith('#') or line.isspace():
            continue
        state, symbol, new_state, new_symbol, direction = line.split()
        # Simplified syntax, without parentheses for both keys and
        # values. The full syntax would be be:
        # TM_program[(state, symbol)] = (new_state, new_symbol, direction)
        TM_program[state, symbol] = new_state, new_symbol, direction
```

```
TM_program
```

[56]:
```
{('del1', '1'): ('del2', '0', 'R'),
 ('del2', '1'): ('mov1R', '0', 'R'),
 ('mov1R', '1'): ('mov1R', '1', 'R'),
 ('mov1R', '0'): ('mov2R', '0', 'R'),
 ('mov2R', '1'): ('mov2R', '1', 'R'),
 ('mov2R', '0'): ('mov1L', '1', 'L'),
 ('mov1L', '1'): ('mov1L', '1', 'L'),
 ('mov1L', '0'): ('mov2L', '0', 'L'),
 ('mov2L', '1'): ('mov2L', '1', 'L'),
 ('mov2L', '0'): ('del1', '0', 'R'),
 ('del1', '0'): ('end', '0', 'R'),
 ('del2', '0'): ('end', '0', 'R')}
```

States and directions are naturally represented as strings. On the other hand, it would be simpler and more natural to represent symbols as the integers `0` and `1`, not as the strings `'0'` and `'1'`, all the more so that `tape` consists of `0`'s and `1`'s, not `'0'`'s and `'1'`'s. One can get the latter from the former with `int()`:

[57]:
```
int('0')
int('1')
int('17')
int('-23')
```

[57]:
```
0
```

[57]:
```
1
```

[57]:
```
17
```

[57]:
```
-23
```

So we can improve our code further as follows:

[58]:
```
TM_program = {}
with open('division_by_2.txt') as TM_program_file:
    for line in TM_program_file:
        if line.startswith('#') or line.isspace():
            continue
        state, symbol, new_state, new_symbol, direction = line.split()
        TM_program[state, int(symbol)] =\
                new_state, int(new_symbol), direction
TM_program
```

[58]:
```
{('del1', 1): ('del2', 0, 'R'),
 ('del2', 1): ('mov1R', 0, 'R'),
 ('mov1R', 1): ('mov1R', 1, 'R'),
 ('mov1R', 0): ('mov2R', 0, 'R'),
```

```
 ('mov2R', 1): ('mov2R', 1, 'R'),
 ('mov2R', 0): ('mov1L', 1, 'L'),
 ('mov1L', 1): ('mov1L', 1, 'L'),
 ('mov1L', 0): ('mov2L', 0, 'L'),
 ('mov2L', 1): ('mov2L', 1, 'L'),
 ('mov2L', 0): ('del1', 0, 'R'),
 ('del1', 0): ('end', 0, 'R'),
 ('del2', 0): ('end', 0, 'R')}
```

Now that the program of the TM has been captured in a form that seems appropriate for further use, we can write code to simulate computation. Recall that `tape` represents only a finite section of the tape. As computation progresses, larger and larger sections of the tape are potentially explored, possibly requiring to extend `tape` accordingly, left or right. The widget does this. Here, in order to simplify our task, we assume that `tape` is defined in such a way that it is large enough and contains enough `0`'s at both ends for the head of the TM not to go beyond the section determined by `tape`, for the input under consideration, so there are enough `0`'s left and right of the `1`'s in `tape` that encode the input for all instructions to be executed within `tape`'s boundaries. With this in mind, and knowing how the division by 2 program works, let us set `tape` as follows, so set for an input of 7, making use of the `+` operator to concatenate lists:

[59]:
```
tape = [0] * 3 + [1] * 7 + [0] * 6
```

When computation ends, there should be only 3 consecutive `1`'s in `tape` since $\lfloor \frac{7}{2} \rfloor = 3$. At any stage of the computation, we need to know the current state and the head's current position. Before computation starts, the current state is the initial state, set to `'del1'`:

[60]:
```
current_state = 'del1'
```

The head is supposed to be positioned below the cell of the tape that contains the leftmost 1. The `index()` method of the `str` class makes it easy to determine that position within `tape`:

[61]:
```
current_position = tape.index(1)
current_position
```

[61]:
```
3
```

We know how to find out the contents of the cell which the head is currently positioned under, that is, the current symbol:

[62]:
```
current_symbol = tape[current_position]
current_symbol
```

[62]:
```
1
```

At this stage, is there a (unique) instruction to execute? Yes if and only if `TM_program` has `(current_state, current_symbol)` as one of its keys:

[63]:
```
(current_state, current_symbol) in TM_program.keys()
```

[63]:
```
True
```

Instead of asking whether a given object is one of the keys of a dictionary, one can more simply ask whether the object is in the dictionary (as a key):

[64]:
```
(current_state, current_symbol) in TM_program
```

[64]:
```
True
```

One can then retrieve the rest of the instruction, its "to do" part:

[65]:
```
new_state, new_symbol, direction = TM_program[current_state, current_symbol]
new_state
new_symbol
direction
```

[65]:
```
'del2'
```

[65]:
```
0
```

[65]:
```
'R'
```

Executing that instruction means changing `tape[current_symbol]` to `new_symbol` (more precisely, letting `tape[current_symbol]` evaluate to the value that `new_symbol` evaluates to, by letting `tape[current_symbol]` denote the bits in memory that `new_symbol` denotes), changing the current state from `current_state` to `new_state`, changing `current_position` from the value it currently has to that value to plus or minus 1 depending on whether `direction` is `'R'` or `'L'`, respectively, and changing `current_symbol` to the value stored in `tape` at the index which is the new value of `current_position`.

To test whether `direction` is `'R'` or `'L'`, it is useful to make use of **equality**, one of the 6 **comparison operators**:

[66]:
```
# Equality
2 == 2, 2 == 3
# Inequality
'a' != 'b', 'a' != 'a'
# Less than (w.r.t. lexicographic order)
'abcd' < 'abe', 'z' < 'abc'
# Less than or equal to (w.r.t. lexicographic order)
[1, 2, 3] <= [1, 2, 3], [2] <= [1, 3]
# Greater than or equal to
False >= False, False > True
# Greater than
2 > True, 0 > False
```

[66]:
```
(True, False)
```

[66]:
```
(True, False)
```

[66]:
```
(True, False)
```

[66]: (True, False)

[66]: (True, False)

[66]: (True, False)

To assign the appropriate value to `current_position`, it is useful to make use of an `if ... else` expression. Putting it all together:

[67]:
```
tape[current_position] = new_symbol
current_state = new_state
current_position = current_position + 1 if direction == 'R'\
                                        else current_position - 1
current_symbol = tape[current_position]

tape
current_state
current_position
current_symbol
```

[67]: [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]

[67]: 'del2'

[67]: 4

[67]: 1

This should be done again and again for as long as there is some instruction to execute, as determined by whether or not `(current_state, current_symbol)` is one of the keys of `TM_program`. To better visualise computation, we can, at every stage of the computation, output a graphical representation of `tape` and below, output the value of `current_state` with its leftmost character right below the symbol in the cell that the head is positioned under. Let us define a function for that purpose:

[68]:
```
def display_current_configuration():
    display_tape()
    print('  ' * current_position, current_state)
```

Let us start from scratch and check that `display_current_configuration()` works as intended:

[69]:
```
tape = [0] * 3 + [1] * 7 + [0] * 6
current_state = 'del1'
current_position = tape.index(1)
current_symbol = tape[current_position]
display_current_configuration()
```

```
---------------------------------
|0|0|0|1|1|1|1|1|1|1|0|0|0|0|0|0|
```

```
---------------------------------
       del1
```

Let us now simulate the first 3 stages of the computation:

[70]:
```
new_state, new_symbol, direction =\
        TM_program[current_state, current_symbol]
tape[current_position] = new_symbol
current_state = new_state
current_position = current_position + 1 if direction == 'R'\
                                        else current_position - 1
current_symbol = tape[current_position]
display_current_configuration()
```

```
---------------------------------
|0|0|0|0|1|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
         del2
```

[71]:
```
new_state, new_symbol, direction =\
        TM_program[current_state, current_symbol]
tape[current_position] = new_symbol
current_state = new_state
current_position = current_position + 1 if direction == 'R'\
                                        else current_position - 1
current_symbol = tape[current_position]
display_current_configuration()
```

```
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
           mov1R
```

[72]:
```
new_state, new_symbol, direction =\
        TM_program[current_state, current_symbol]
tape[current_position] = new_symbol
current_state = new_state
current_position = current_position + 1 if direction == 'R'\
                                        else current_position - 1
current_symbol = tape[current_position]
display_current_configuration()
```

```
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
             mov1R
```

A `while` statement, another kind of loop, lets us execute those 6 statements again and again, for as long as `(current_state, current_symbol)` is one of the keys of `TM_program`:

[73]:
```
tape = [0] * 3 + [1] * 7 + [0] * 6
current_state = 'del1'
current_position = tape.index(1)
current_symbol = tape[current_position]
display_current_configuration()
while (current_state, current_symbol) in TM_program:
    new_state, new_symbol, direction =\
            TM_program[current_state, current_symbol]
    tape[current_position] = new_symbol
    current_state = new_state
    current_position = current_position + 1 if direction == 'R'\
                                            else current_position - 1
    current_symbol = tape[current_position]
    display_current_configuration()
```

```
---------------------------------
|0|0|0|1|1|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
       del1
---------------------------------
|0|0|0|0|1|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
         del2
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
           mov1R
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
             mov1R
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
               mov1R
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
                 mov1R
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
                   mov1R
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
                     mov1R
---------------------------------
```

```
|0|0|0|0|0|1|1|1|1|1|0|0|0|0|0|0|
---------------------------------
                      mov2R
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
                    mov1L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
                   mov2L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
                 mov2L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
               mov2L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
             mov2L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
           mov2L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
         mov2L
---------------------------------
|0|0|0|0|0|1|1|1|1|1|0|1|0|0|0|0|
---------------------------------
           del1
---------------------------------
|0|0|0|0|0|0|1|1|1|1|0|1|0|0|0|0|
---------------------------------
             del2
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|0|0|0|0|
---------------------------------
               mov1R
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|0|0|0|0|
---------------------------------
                 mov1R
---------------------------------
```

```
|0|0|0|0|0|0|0|1|1|1|0|1|0|0|0|0|
---------------------------------
                  mov1R
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|0|0|0|0|
---------------------------------
                   mov1R
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|0|0|0|0|
---------------------------------
                     mov2R
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|0|0|0|0|
---------------------------------
                       mov2R
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|1|0|0|0|
---------------------------------
                     mov1L
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|1|0|0|0|
---------------------------------
                   mov1L
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|1|0|0|0|
---------------------------------
                  mov2L
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|1|0|0|0|
---------------------------------
                mov2L
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|1|0|0|0|
---------------------------------
               mov2L
---------------------------------
|0|0|0|0|0|0|0|1|1|1|0|1|1|0|0|0|
---------------------------------
                del1
---------------------------------
|0|0|0|0|0|0|0|0|1|1|0|1|1|0|0|0|
---------------------------------
                 del2
---------------------------------
```

```
|0|0|0|0|0|0|0|0|0|1|0|1|1|0|0|0|
--------------------------------
                  mov1R
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|0|0|0|
--------------------------------
                   mov1R
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|0|0|0|
--------------------------------
                    mov2R
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|0|0|0|
--------------------------------
                     mov2R
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|0|0|0|
--------------------------------
                      mov2R
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|1|0|0|
--------------------------------
                     mov1L
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|1|0|0|
--------------------------------
                   mov1L
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|1|0|0|
--------------------------------
                    mov1L
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|1|0|0|
--------------------------------
                  mov2L
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|1|0|0|
--------------------------------
                mov2L
--------------------------------
|0|0|0|0|0|0|0|0|0|1|0|1|1|1|0|0|
--------------------------------
                  del1
--------------------------------
|0|0|0|0|0|0|0|0|0|0|0|1|1|1|0|0|
--------------------------------
                   del2
--------------------------------
```

```
|0|0|0|0|0|0|0|0|0|0|0|1|1|1|0|0|
--------------------------------
                          end
```

Observe the following:

[74]:
```
direction = 'R'
direction == 'R'
# direction == 'R' evaluates to True, which is then converted to 1
# for the arithmetic expression to make sense
2 * (direction == 'R') - 1
```

[74]:
```
True
```

[74]:
```
1
```

[75]:
```
direction = 'L'
direction == 'R'
# direction == 'R' evaluates to False, which is then converted to 0
# for the arithmetic expression to make sense
2 * (direction == 'R') - 1
```

[75]:
```
False
```

[75]:
```
-1
```

This allows one to change the simulation loop as follows:

[76]:
```
tape = [0] * 3 + [1] * 2 + [0] * 3
current_state = 'del1'
current_position = tape.index(1)
current_symbol = tape[current_position]
display_current_configuration()
while (current_state, current_symbol) in TM_program:
    new_state, new_symbol, direction =\
            TM_program[current_state, current_symbol]
    tape[current_position] = new_symbol
    current_state = new_state
    # Alternative notation for
    # current_position = current_position + 2 * (direction == 'R') - 1
    current_position += 2 * (direction == 'R') - 1
    current_symbol = tape[current_position]
    display_current_configuration()
```

```
-----------------
|0|0|0|1|1|0|0|0|
-----------------
       del1
-----------------
|0|0|0|0|1|0|0|0|
```

```
-----------------
         del2
-----------------
|0|0|0|0|0|0|0|0|
-----------------
           mov1R
-----------------
|0|0|0|0|0|0|0|0|
-----------------
             mov2R
-----------------
|0|0|0|0|0|0|1|0|
-----------------
           mov1L
-----------------
|0|0|0|0|0|0|1|0|
-----------------
         mov2L
-----------------
|0|0|0|0|0|0|1|0|
-----------------
           del1
-----------------
|0|0|0|0|0|0|1|0|
-----------------
             end
```