# PyTuring Documentation

Max Miller
University of New Hampshire

May 10, 2018

## 1 Why PyTuring?

As we continue to build more and more complex computational methods, abstracted programming languages, and generally move away from working on the ground floor of computation, we can begin to lose sight of why we are able to use machines to compute. With this (and a final project for my math course) as motivation, I created PyTuring, a complete programming language for emulating Turing Machines. As an additional piece of motivation, I am also putting Turing's assertion "It is possible to invent a single machine which can be used to compute any computable sequence." (Church & Turing, 1937)

## 2 Whats With the Py?

Python is my language of choice to start most projects, as it is essentially like writing sudo-code with slightly more effectiveness. The ultimate goal is to compile the language with a home-brewed compiler built-in Rust, but if history shows then I will be building this compiler in Python as well.

## 3 Two Versions

There are two interpreters for PyTuring, both built in Python. The first is for a more traditional Turing Machine (specifically the ones he lays out in On Computable Numbers), with a Unicode alphabet as allowed tape inputs. This allows for more flexibility in symbol choices and requires less tape to perform more complex computations.

The second interpreter only allows for a finite alphabet of 0s and 1s, which serves the purpose of emulating a true binary, semi-conductor powered computer more effectively. This is where the bulk of my understanding and want to work on this project comes from.

## 4 Hello Turing

An example "Hello Turing" script in PyTuring:

```
S 0
T _
P 0
START
0 _ H 1;
1 H >> 1;
1 _ E 2;
2 E >> 2;
2 _ L 3;
3 L >> 4;
4 _ L 4;
4 L >> 5;
5 _ O 5;
5 O >> 6;
6 _ >> 6;
6 _ T 7;
7 T >> 7;
7 _ U 8;
8 U >> 8;
8 _ R 9;
9 R >> 10;
10 _ I 10;
10 I >> 11;
11 _ N 11;
11 N >> 12;
12 _ G 12;
12 _ >> 13;
END
```

Running this will get you:

```
Starting Tape:          _
End Tape:               hello_turing
```

As you can see, writing out a simple program in PyTuring can prove to be tedious, but it's purpose is not for efficiency. Let's break it down:

### 4.1 The First Three Lines:

```
S 0
```

The first line of any PyTuring program is the Start State, defined with an "S {START STATE}". The purpose of this line is mostly for debugging, as common practice is to start your machine in the zero state. Only integers are allowed as input.

```
T _
```

The second line defines the starting tape of your program, defined with a "T {TAPE}". This will be the initial input into your machine. For the purposes of this program, we have started with a single _, denoting a space. This was mostly for visual purposes, as a space is an equally valid character which holds the same value.

```
P 0
```

The third line defines the starting position of the input tape, indexed at zero, and defined by "P {TAPE POSITION}". Like the start state, this is mostly for debugging purposes, but in more complex programs it may be beneficial to work with a non-zero start position. Only integers are allowed as input.

## 4.2 The Body:

Once the first three lines have been defined, the body of the program must be denoted with:

```
START
...
END
```

This allows the compiler to differential the start of the program from the rest of the initial definitions.

## 4.3 Commands:

Each command is made up of four parts, separated by spaces. The structure goes as follows:

{*LOOK STATE*} {*TAPE VALUE*} {*ACTION*} {*GO TO STATE*}

```
0 _ H 1;
1 H >> 1;
```

If we look at our first two lines, we get a complete set of possible command configurations, as well as two examples of the structure. To break this down, we will compile a statement from the command structure which shows how to read a command. The first part LOOK STATE is an integer which tells the machine "If in state x". TAPE VALUE then continues this statement, adding on "If in state x, and the machine reads y on the tape", with Y being the value given in this section. ACTION is then what to do when the TAPE VALUE is read, making the statement now read "If in state x, and the machine reads y on the tape, do action z". If action z is a single Unicode character (that is not a space), then the machine will place that value on the tape. If the action z is “>>” or “<<”, then the machine will move right or left on the tape, respectively. A command can also call a submachine, which we will discuss later. Finally, when the machine gets to GO TO STATE, the machine will now move into the defined state, with the statement now reading ”If in state x, and the machine reads y on the tape, do action z, and move into state x $_n$”. The reason for using $x_n$ is because $x_n$ can be any state, including the state x, or a state with no associated commands.

## 4.4 Ending the Program:

A PyTuring program ends if and only if the program reaches a state for which there are no commands, or the program reads a tape value for which the current state has no commands. Because of this, a PyTuring program must be built with the end in mind, otherwise, the program will stop unexpectedly or run in an infinite loop until the processor decides to kill it, or you get smart enough to kill it yourself.

## 4.5 Submachines:

As Turing notes, a Universal Computing Machine must be able to emulate any other computing machine. To illustrate this concept, PyTuring implements submachines, also called contained-state methods. A submachine implements its own set of instructions and states but works off of the same tape as the master program. A submachine is defined with:

```
SUB MACHINE_NAME
    S 0
    ...
END
```

The “S 0” serves the same purpose as the master program, defining the start state of the submachine. Each submachine can be written with its own internal states and commands which will not affect the larger program. A submachine can be called in master program using the command:

```
0 call MACHINE_NAME 1;
```

Where the TAPE VALUE is “call” and the ACTION is “MACHINE_NAME”. Just like the master program, a submachine will exit if and only if the submachine reaches a state for which there are no commands, or the submachine reads a tape value for which the current state has no commands. Other than their educational benefits, submachines also serve the benefit of code organization.

## References

Church, A., & Turing, A. M. (1937, mar). On Computable Numbers with an Application to the Entscheidungsproblem. *The Journal of Symbolic Logic*, *2*(1), 42. Retrieved from `https://doi.org/10.2307%2F2268810` doi: 10.2307/2268810