Challenge 01: Timelines

The challenge

The world is an incredible, parallel, asynchronous place. At any given time, billions upon billions of events are firing all around us. On computing devices, these might last nanoseconds, they might be recurring events, they might be scheduled in the future, then might start and stop and restart according to external factors.

Time is of the essence as we aim to model these events!

Your task will be to parse event definitions and create and manipulate timelines that describes them as they happen.

Additional constraints

- You *must* write your solution using Python 3 code. Yes, even if you don't know it. See the syntax primer.
- You *cannot* use the Internet! Moderators will be spying on you to make sure you don't cheat!
- Every 20 minutes there will be a rotation within your group. Make sure your teammates can pick up where you left off!

Input format

Your software will need to parse files generated by a super real and very good profiling software. The profiler logs the times at which a new thread or process starts, when it sleeps, and when it dies.

Unfortunately the developers of the super real and very good profiler chose to create their own file format which you will need to read. "We'll just create our own standard!" the developers famously said. They defined their format in terms of "tasks" and "processors", where a task is either a thread or a process.

Tuples like this describe a task, where `t_n` is an integer time in seconds.

```
(t_0,...,t_n)
```

The profiler logs tasks on each processor, so on multi-core systems, each task will be prefixed with a processor ID, like so:

```
0:(0,2)
```

The above line indicates a task that ran on processor 0 from time = 0 until 2 seconds after system start.

When a task stops executing, the profiler marks this in the tuple by using a semi-colon instead of a comma. For example, a task that runs from time 0 until time 10, stopping at time 4, and resuming at time 9, looks like:

```
(0,4;9,10)
```

Sample inputs

Inputs will be plain text, new-line delimited files. The last line in the file will always be blank.

Single core

```
(0,2;6,10)
(2,5)
(20,45)
(5,6)
```

Multi-core

```
0:(12,34;65,200)
2:(0,13;14,16)
1:(67,100)
0:(34,40)
```

Tasks do not necessarily show up in order, nor sorted by processor.

Tasks on the same processor will *never* have overlapping execution times.

Output format

For each processor in the profile, output a graph of the activity to the console. Use `_` to indicate idle state, `/` to indicate resuming from idle, `|` to indicate activity, and `\` to indicate returning to idle. If the CPU is only active for 2 seconds, output `/\`. If the CPU is only active for 1 seconds, output `|`.

For example, given the input:

```
(0,1;2,5;6,9)
(14,14;17,22)
(12,12)
```

The output would look like:

```
/\/||\/||\_|__|__/||||\
```

The timeline should not display idle state past the last task, however it *should* display idle state from time 0 until the first recorded activity.

For a multi-core input file, you should print one graph per line, labelled with the processor ID. For example:

```
0:____/\__|__/||||||||\__|
1:_/||\__/|||||||\_____|__/\
2:___________________|
3:____/||||\__________|_|
```

Running the program

Your program should take two positional arguments. The first will be the name of the input file. The second will be an optional name of the output file. If no output file is supplied, then your program should print the results to the console (`stdout`).

Your script should be executable so that it can be run like this:

```
./timeline.py input.txt output.txt
```

or like this:

```
./timeline.py input.txt
```