Demo

December 31, 2020

This is a demonstration of the features of IJuliaTimeMachine. It also explains some of the design choices.

1 What IJulia Provides

IJulia already provides some historical information. `In` records the inputs to cells, and `Out` records their outputs. `IJulia.n` is the number of the current cell.

```
[1]: 1+1
```

```
[1]: 2
```

```
[2]: In[1]
```

```
[2]: "1+1"
```

```
[3]: Out[1]
```

```
[3]: 2
```

2 Setting up IJuliaTimeMachine

I recommend assigning `IJuliaTimeMachine` a shorter name, like `TM` as below.

```
[4]: import IJuliaTimeMachine
     TM = IJuliaTimeMachine
```

```
[4]: IJuliaTimeMachine
```

If you use it a lot, and don't want to keep typing the `TM` prefix, you can instead type `using IJuliaTimeMachine`. This will import `@past` and `vars`.

3 Recalling past states

```
[5]: x = 1+1
```

```
[5]: 2
```

[6]:
```
x = randn(3)
```

[6]:
```
3-element Array{Float64,1}:
 -0.07656817141946694
 -1.7219397904552953
 -1.6390299328866278
```

Say that I just accidentally change the value of the variable `x`, and I'd like to know what it's value was after cell 5. I can recover the state of variables from then with `@past`.

[7]:
```
TM.@past 5
x
```

[7]:
```
2
```

The value of `ans` was also recalled.

[8]:
```
ans
```

[8]:
```
2
```

The Time Machine stores a `deepcopy` of every variable. While this is inefficient, it allows us to recover a vector, even if some other cell changes one of its entries.

[9]:
```
vec = collect(1:4)
```

[9]:
```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

[10]:
```
vec[3] = 100
vec
```

[10]:
```
4-element Array{Int64,1}:
   1
   2
 100
   4
```

[11]:
```
TM.@past 9
vec
```

[11]:
```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

By default `@past` also recalls the output of the past cell, and so that output appears in the display.

```
[12]: TM.@past 9
```

```
[12]: 4-element Array{Int64,1}:
       1
       2
       3
       4
```

This can be very useful because IJulia's Out stores a pointer to an array, rather than the array. This means that the value recalled can be changed, like this.

```
[13]: y = [1;2;3]
```

```
[13]: 3-element Array{Int64,1}:
       1
       2
       3
```

```
[14]: Out[13]
```

```
[14]: 3-element Array{Int64,1}:
       1
       2
       3
```

```
[15]: y[3] = 0
      Out[13]
```

```
[15]: 3-element Array{Int64,1}:
       1
       2
       0
```

Note that the last element changed to a 0. This does not happen with the values stored by the Time Machine.

```
[16]: TM.@past 13
```

```
[16]: 3-element Array{Int64,1}:
       1
       2
       3
```

The Time Machine only stores variables it can effectively copy. Right now, it can not copy functions.

```
[17]: # note that cell is equal to the number of this cell, 17.
      cell = IJulia.n
      f(x) = x
      f(1)
```

```
[17]: 1
```

```
[18]: y = 2
      f(x) = 2x
      f(1)
```

```
[18]: 2
```

```
[19]: TM.@past cell
```

```
[19]: 1
```

```
[20]: f(1)
```

```
[20]: 2
```

But, if the only thing that changes in a function is a global variable, then you can essentially recover the function.

```
[21]: f(x) = y*x
```

```
[21]: f (generic function with 1 method)
```

```
[22]: cell = IJulia.n
      y = 3
      f(1)
```

```
[22]: 3
```

```
[23]: y = 4
      f(1)
```

```
[23]: 4
```

```
[24]: TM.@past cell
```

```
[24]: 3
```

```
[25]: f(1)
```

```
[25]: 3
```

You can stop the Time Machine from saving.

[26]:
```
TM.saving!(false)
```

[26]:
```
save_state (generic function with 1 method)
```

[27]:
```
z = "hello!"
```

[27]:
```
"hello!"
```

[28]:
```
TM.@past 27
```

```
State 27 was not saved.

Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] top-level scope at /Users/spielman/Dropbox/dev/IJuliaTimeMachine/src/
↪the_past.jl:98
 [3] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
↪1091
```

And, you can make it start saving again.

[29]:
```
TM.saving!(true)
```

[29]:
```
1-element Array{Function,1}:
 save_state (generic function with 1 method)
```

[30]:
```
z = "hi :)"
```

[30]:
```
"hi :)"
```

[31]:
```
z = "overwrite that"
```

[31]:
```
"overwrite that"
```

[32]:
```
TM.@past 30
z
```

[32]:
```
"hi :)"
```

If we really want to forget the past, or just save memory, we can clear some history. Just give a list of the cells to be cleared. If no list is specified, it clears all of them. This is the safest option, because to free up the memory used by a variable, one must clear *every* cell in which that variable exists.

[33]:
```
z = []
TM.clear_past([30])
```

[34]:
```
TM.@past 30
```

```
State 30 was not saved.

Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] top-level scope at /Users/spielman/Dropbox/dev/IJuliaTimeMachine/src/
 ↪the_past.jl:98
 [3] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

But, this didn't get rid of the string "hi :)", because that string was also saved in cell 32. You can see that by using the `vars` function, which returns a dictionary of all the variables saved with a cell. To get rid of that string, we would need to clear cells 30 and 32.

[35]:
```
TM.vars(32)
```

[35]:
```
Dict{Any,Any} with 5 entries:
  :vec  => [1, 2, 3, 4]
  :y    => 3
  :cell => 22
  :z    => "hi :)"
  :x    => 2
```

The big table of all the stored variables is `TM.VX.store`

[36]:
```
TM.VX.store
```

[36]:
```
Dict{Any,Any} with 14 entries:
  0xf02d7561089106d2 => [1, 2, 3, 4]
  0xc70ed234e0a3ad59 => 17
  0xf15becc4be4c0cd4 => [-0.0765682, -1.72194, -1.63903]
  0xf4e13e0713120297 => "hi :)"
  0x3688cf5d48899fa6 => 3
  0x5c97e8eedebac7cc => [1, 2, 100, 4]
  0x5a94851fb48a6e05 => 2
  0x40322bb383bc0a9c => [1, 2, 3]
  0x867b4bb4c42e5661 => 4
  0xf9dae1a92b5e5ba7 => Any[]
  0x6fd433390b9283dd => 22
  0xb98a8ec9a0959679 => "hello!"
  0x700bc5643854405f => "overwrite that"
  0x6680a11d7499012b => [1, 2, 0]
```

[37]:
```
z = []
TM.clear_past(30:36)
```

[38]:
```
TM.VX.store
```

[38]:
```
Dict{Any,Any} with 12 entries:
  0xf02d7561089106d2 => [1, 2, 3, 4]
  0xc70ed234e0a3ad59 => 17
  0xf15becc4be4c0cd4 => [-0.0765682, -1.72194, -1.63903]
  0x3688cf5d48899fa6 => 3
  0x5c97e8eedebac7cc => [1, 2, 100, 4]
  0x5a94851fb48a6e05 => 2
  0x40322bb383bc0a9c => [1, 2, 3]
  0x867b4bb4c42e5661 => 4
  0xf9dae1a92b5e5ba7 => Any[]
  0x6fd433390b9283dd => 22
  0xb98a8ec9a0959679 => "hello!"
  0x6680a11d7499012b => [1, 2, 0]
```

If you want to see the cell output that TM stored,look at `TM.ans(cell)`.

[39]:
```
TM.ans(23)
```

[39]:
```
4
```

Variables are stored by hash so that each piece of data is only stored once. This reduces wasted memory. Unfortunately, we can not use Julia's default hash function to do this, as small changes to data do not necessarily change this hash. Here are two examples of this problem.

[40]:
```
a = randn(1000,1000)
h0 = hash(a)
a[1,1] = 0
h1 = hash(a)
h0 == h1
```

[40]:
```
true
```

[41]:
```
mutable struct S
    a
end
x = S(1)
h0 = hash(x)
x.a = 0
h1 = hash(x)
h0 == h1
```

[41]:
```
true
```

TM uses its own hash function to avoid this problem.

```
[42]: a = randn(1000,1000)
      h0 = TM.tm_hash(a)
      a[1,1] = 0
      h1 = TM.tm_hash(a)
      h0 == h1
```

```
[42]: false
```

```
[43]: mutable struct S
          a
      end
      x = S(1)
      h0 = TM.tm_hash(x)
      x.a = 0
      h1 = TM.tm_hash(x)
      h0 == h1
```

```
[43]: false
```

Unfortunately, tm_hash is can be slow because it must examine every element of a data structure. But, you probably don't want to store a big chunk of data in the TM anyway.

If you have some big piece of data that you do not want stored, you can tell the Time Machine not to do that. The function `dontsave` tells the TM not to store the variable it is given. But, if you copy the variable, the copy will be saved. Similarly, it will be saved if you bind the variable to a new piece of data.

```
[44]: BIG = randn(1000,1000)
      TM.dontsave(BIG)
```

```
[45]: TM.vars(IJulia.n - 1)
```

```
[45]: Dict{Any,Any} with 9 entries:
        :a    => [0.0 0.580431 … 0.0275737 -0.607591; 1.60128 0.293623 … 0.177615 -0.…
        :vec  => [1, 2, 3, 4]
        :h1   => 0x4330888b40615b85
        :y    => 3
        :h0   => 0xc0fb5ae483d856da
        :cell => 22
        :S    => S
        :z    => Any[]
        :x    => S(0)
```

```
[46]: Y = BIG
      Y === BIG
```

```
[46]: true
```

[47]:
```
# Y is not saved because it has the same binding as BIG
TM.vars(IJulia.n - 1)
```

[47]:
```
Dict{Any,Any} with 9 entries:
  :a    => [0.0 0.580431 … 0.0275737 -0.607591; 1.60128 0.293623 … 0.177615 -0.…
  :vec  => [1, 2, 3, 4]
  :h1   => 0x4330888b40615b85
  :y    => 3
  :h0   => 0xc0fb5ae483d856da
  :cell => 22
  :S    => S
  :z    => Any[]
  :x    => S(0)
```

[48]:
```
# now, Y has a different binding
Y = copy(BIG)
Y === BIG, Y == BIG
```

[48]:
```
(false, true)
```

[49]:
```
TM.vars(IJulia.n - 1)
```

[49]:
```
Dict{Any,Any} with 10 entries:
  :a    => [0.0 0.580431 … 0.0275737 -0.607591; 1.60128 0.293623 … 0.177615 -0.…
  :vec  => [1, 2, 3, 4]
  :h1   => 0x4330888b40615b85
  :y    => 3
  :h0   => 0xc0fb5ae483d856da
  :cell => 22
  :S    => S
  :Y    => [-1.23786 0.744897 … -1.28247 0.0257696; 0.585993 -1.45679 … -0.5606…
  :z    => Any[]
  :x    => S(0)
```

[50]:
```
# This does not change the binding of BIG
BIG[1,1] = 0
```

[50]:
```
0
```

[51]:
```
TM.vars(IJulia.n - 1)
```

[51]:
```
Dict{Any,Any} with 10 entries:
  :a    => [0.0 0.580431 … 0.0275737 -0.607591; 1.60128 0.293623 … 0.177615 -0.…
  :vec  => [1, 2, 3, 4]
  :h1   => 0x4330888b40615b85
  :y    => 3
  :h0   => 0xc0fb5ae483d856da
  :cell => 22
  :S    => S
  :Y    => [-1.23786 0.744897 … -1.28247 0.0257696; 0.585993 -1.45679 … -0.5606…
  :z    => Any[]
  :x    => S(0)
```

On this laptop, it takes about half a second to hash an 800MB array. If Jupyter is reacting slowly, hashing a big variable could be the reason. You can fix this by telling Jupyter to stop saving it. But, you won't loose it.

[52]:
```
@time BIG = randn(1000,1000,100)
@show Base.summarysize(BIG)
@time TM.tm_hash(BIG)
```

```
  0.623984 seconds (2 allocations: 762.940 MiB, 8.07% gc time)
Base.summarysize(BIG) = 800000048
  0.488059 seconds (67.17 k allocations: 3.211 MiB)
```

[52]:
```
0xc1ba4cc9ec75c856
```

[53]:
```
TM.dontsave(Y)
```

[54]:
```
# But, we changed the binding of BIG. So it has been saved.
TM.vars(IJulia.n - 1)
```

[54]:
```
Dict{Any,Any} with 10 entries:
  :a    => [0.0 0.580431 … 0.0275737 -0.607591; 1.60128 0.293623 … 0.177615 -0.…
  :vec  => [1, 2, 3, 4]
  :h1   => 0x4330888b40615b85
  :y    => 3
  :h0   => 0xc0fb5ae483d856da
  :cell => 22
  :S    => S
  :z    => Any[]
  :BIG  => [-0.924035 1.59391 … -0.304681 -0.992116; -0.208401 -0.829228 … -1.2…
  :x    => S(0)
```

`IJulia.n` stores the number of a cell. We will keep using this to index cells to facilitate editing of the notebook.

[55]:
```
IJulia.n
```

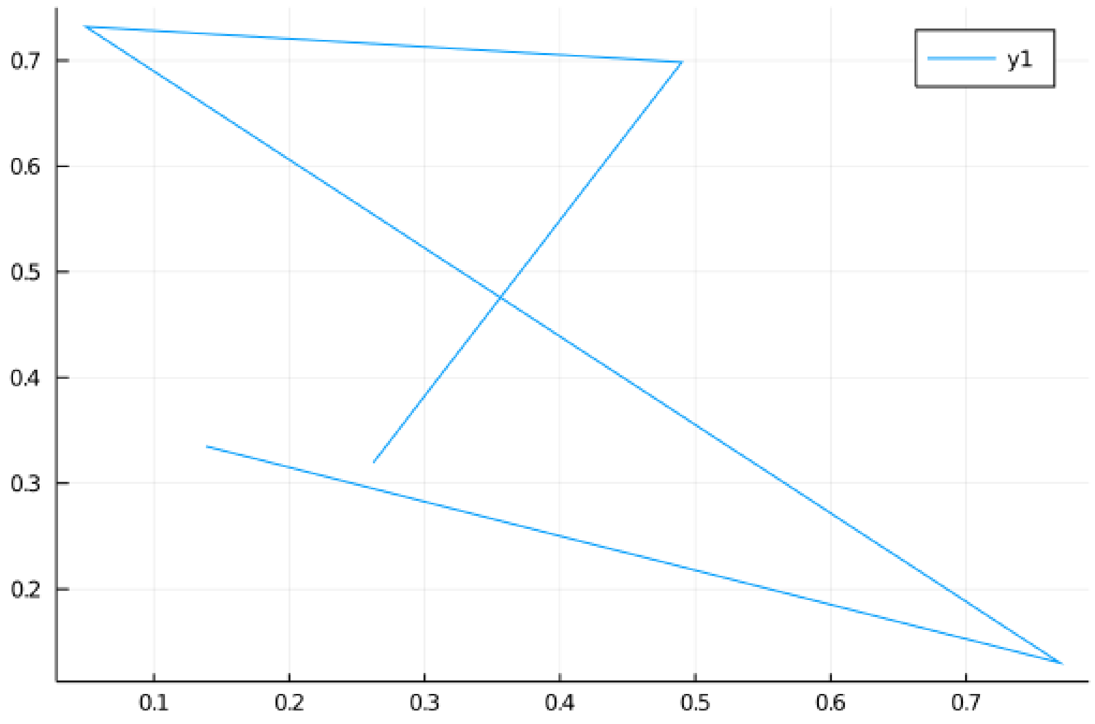
[55]:
```
55
```

It even works with plots.

[56]:
```
cell = IJulia.n
using Plots
```

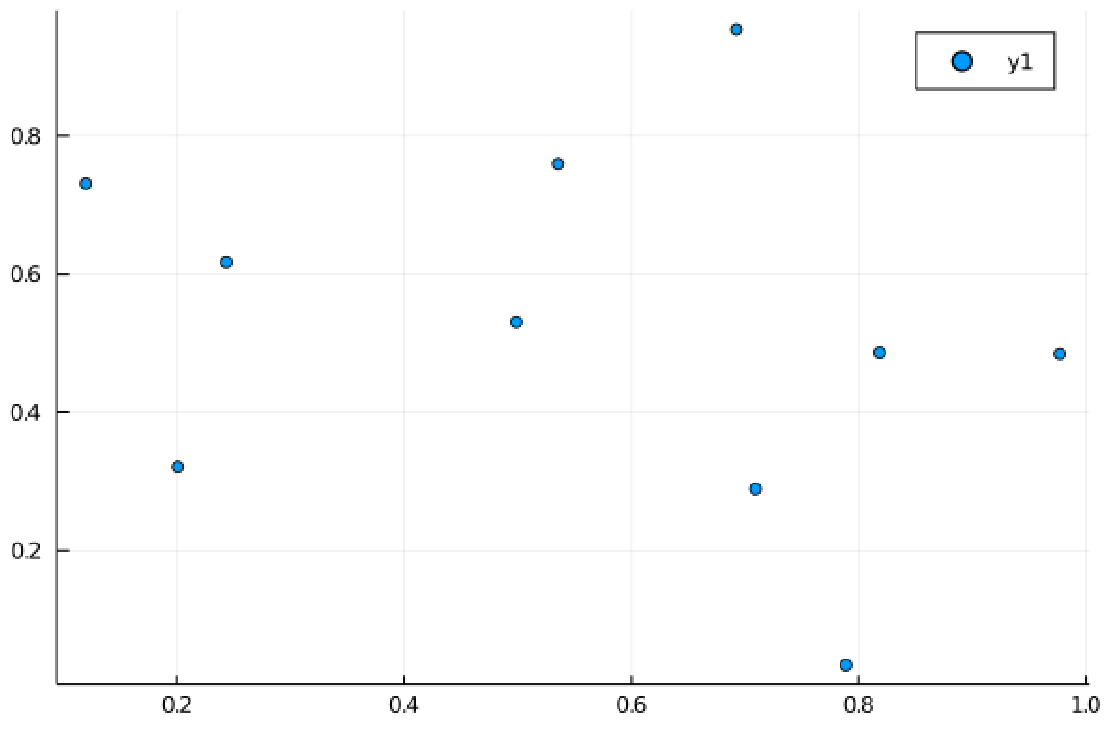
```
default(fmt = :png)
p = plot(rand(5), rand(5))
```

[56]:



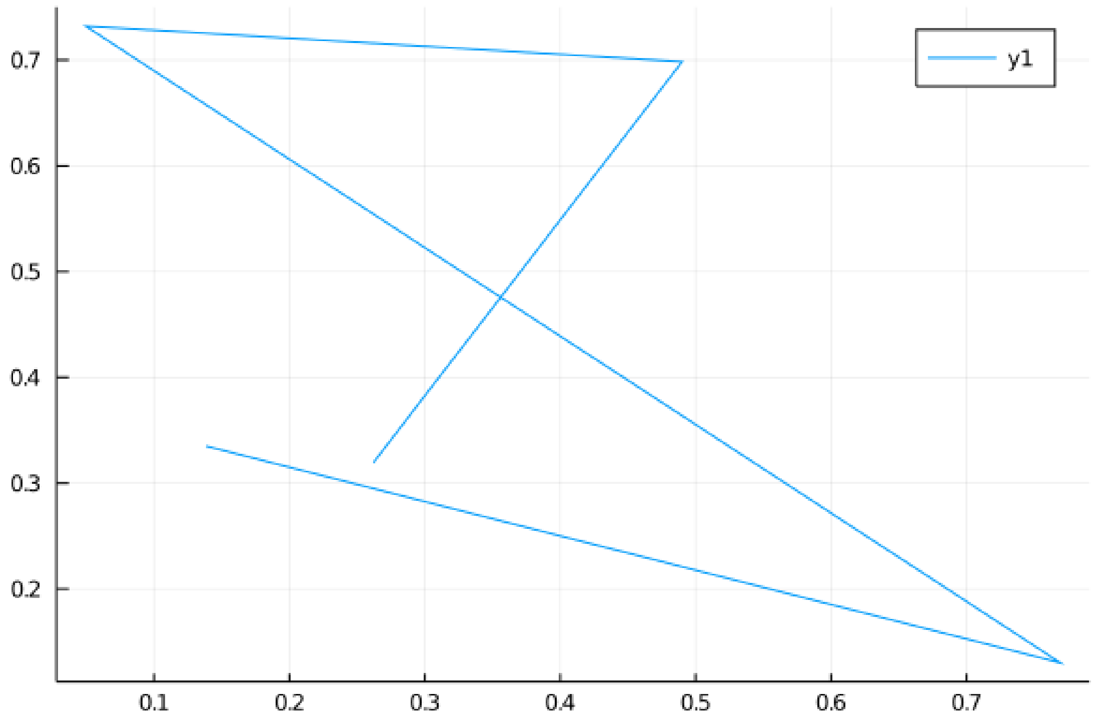
[57]:
```
p = scatter(rand(10), rand(10))
```

[57]:



[58]:
```
TM.vars(cell)[:p]
```

[58]:



4 Running big jobs in threads

The other feature of Time Machine is that it lets you run intensive jobs in threads, so that you can get other work done while they are running. If you have a multicore machine, you can also view this as a way to manage running a bunch of experiments from Jupyter. The key is to wrap the jobs in `TM.@thread begin`, followed by `end`. Jobs that are running inside a `@thread` block sandbox their variables. To access the values of the variables from the thread after it finishes, look at `vars`, `ans`, or use `@past`.

When the jobs finish, their result is stored in `Out`, and you can access their state from `past`. Unfortunately, if the jobs contain any print statements, they can show up in other cells.

To see which jobs are running, look at `TM.running`. `TM.finished` contains a list of those that have finished.

In the examples below, we will simulate the delay of a long-running job with `sleep`. As you will see, results will change after jobs finish.

[59]:
```
x = collect(1:3)
y = 1
```

[59]:
```
1
```

[60]:
```
t0 = time()
n = IJulia.n
TM.@thread begin
    y = y + 1
    push!(x,y)
    sleep(5)
    sum(x)
end
```

[60]:
```
Task (runnable) @0x00000001192add50
```

[61]:
```
println("After $(time() - t0) seconds.")
Out[n]
```

```
After 0.9098551273345947 seconds.
```

[61]:
```
Task (runnable) @0x00000001192add50
```

[62]:
```
sleep(10)
```

[63]:
```
println("After $(time() - t0) seconds.")
Out[n]
```

```
After 12.024251937866211 seconds.
```

```
[63]: 8
```

```
[64]: x, y
```

```
[64]: ([1, 2, 3], 1)
```

```
[65]: TM.@past n
      x, y
```

```
[65]: ([1, 2, 3, 2], 2)
```

There is a subtle reason that I put the `sleep(10)` statement on a separate line. The output of finished jobs is only inserted into `Out` at the start of the execution of the first cell that is run after the job finishes. So, it is possible to go one cell without the output being correct. If you want to test it, put the `sleep(10)` inside the next cell, and then run the cells in quick succession.

You can not put two `@thread` statements into one cell.

```
[66]: n = IJulia.n
      TM.@thread begin
          x = x + 1
          y[1] = 3
          sum(y)
      end
      TM.@thread begin
          x = x + 1
          y[1] = 4
          sum(y)
      end
```

```
@thread can be called at most once per cell.
```

But, you can have many running at once. That's the point!

```
[67]: function intense(n)
          sleep(n)
          println("I slept for $(n) seconds!")
          n^2
      end
```

```
[67]: intense (generic function with 1 method)
```

```
[68]: t0 = time()
      n1 = IJulia.n
      TM.@thread intense(10)
```

```
[68]: Task (runnable) @0x00000001192ad690
```

[69]:
```
println("After $(time()-t0) seconds")
n2 = IJulia.n
TM.@thread intense(11)
```

```
After 0.508124828338623 seconds
```

[69]:
```
Task (runnable) @0x000000011845f190
```

[70]:
```
println("After $(time()-t0) seconds")
n3 = IJulia.n
TM.@thread intense(2)
```

```
After 1.018509864807129 seconds
```

[70]:
```
Task (runnable) @0x00000001192ae890
```

[71]:
```
println("After $(time()-t0) seconds")
TM.running
```

```
After 1.5418009757995605 seconds
```

[71]:
```
Set{Any} with 3 elements:
  68
  69
  70
```

[72]:
```
sleep(3)
```

```
I slept for 2 seconds!
```

[73]:
```
println("After $(time()-t0) seconds")
TM.running
```

```
After 6.071583032608032 seconds
```

[73]:
```
Set{Any} with 2 elements:
  68
  69
```

[74]:
```
println("After $(time()-t0) seconds")
TM.finished
```

```
After 6.620029926300049 seconds
```

[74]:
```
2-element Array{Int64,1}:
 60
 70
```

[75]:
```
sleep(10)
println("After $(time()-t0) seconds")
TM.running
```

```
I slept for 10 seconds!
I slept for 11 seconds!
After 17.50132989883423 seconds
```

[75]:
```
Set{Any}()
```

[76]:
```
Out[n1], Out[n2], Out[n3]
```

[76]:
```
(100, 121, 4)
```

You can also get notifications of when jobs finish. These can be sent to Jupyter, in which case they will just print. You can also send them to the terminal in which Jupyter is running, or both.

[77]:
```
TM.notify_jupyter!(true)
TM.notify_terminal!(true)
@TM.thread begin
    println("this will run for 3 seconds")
    sleep(3)
end
```

```
this will run for 3 seconds
```

[77]:
```
Task (runnable) @0x000000011845c010
```

[78]:
```
n = IJulia.n
using LinearAlgebra
@TM.thread begin
    min_svd = Inf
    min_matrix = []
    for i in 1:100
        M = rand([-1;1], 1000, 1000)
        v = minimum(svdvals(M))
        if v < min_svd
            min_svd = v
            min_matrix = M
        end
    end
    min_svd
end
```

```
Cell 78
```

[78]:
```
Task (runnable) @0x00000001188be890
```

```
 finished.
Cell 77 finished.
```

[79]:
```
# do some other stuff, clobbering v and M
v = 0
M = randn(100,100)
;
```

[80]:
```
# that min svd value computed above
TM.ans(n)
```

[80]:
```
0.00012619151374331338
```

[81]:
```
# and, the matrix that realized it
M = TM.vars(n)[:min_matrix]
minimum(svdvals(M))
```

[81]:
```
0.00012619151374331338
```

[ ]:
```

```