This is a demonstration of the features of IJuliaTimeMachine.

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.

In [1]:
```
1+1
```
Out[1]:
```
2
```

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

In [3]:
```
Out[1]
```
Out[3]:
```
2
```

Setting up IJuliaTimeMachine

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

In [4]:
```
using IJuliaTimeMachine
TM = IJuliaTimeMachine
```
Out[4]:
```
IJuliaTimeMachine
```

Recalling past states

In [5]:
```
x = 1+1
```
Out[5]:
```
2
```

In [6]:
```
x = randn(3)
```
Out[6]:
```
3-element Array{Float64,1}:
  0.7883376083383903
 -0.3227728304284359
  1.1642447725348362
```

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`.

In [7]:
```
TM.@past 5
x
```
Out[7]:
```
2
```

The value of `ans` was also recalled.

In [8]:
```
ans
```
Out[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.

In [9]:
```
vec = collect(1:4)
```
Out[9]:
```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

In [10]:
```
vec[3] = 100
vec
```
Out[10]:
```
4-element Array{Int64,1}:
   1
   2
 100
   4
```

In [11]:
```
TM.@past 9
```
Out[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.

In [12]:
```
vec
```
Out[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.

In [13]:
```
y = [1;2;3]
```
Out[13]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

In [14]:
```
Out[13]
```
Out[14]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

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

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

In [17]:
```
cell = IJulia.n
f(x) = x
f(1)
```
Out[17]:
```
1
```

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

In [19]:
```
TM.@past cell
```
Out[19]:
```
1
```

In [20]:
```
f(1)
```
Out[20]:
```
2
```

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

In [21]:
```
f(x) = y*x
```
Out[21]:
```
f (generic function with 1 method)
```

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

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

In [24]:
```
TM.@past cell
```
Out[24]:
```
3
```

In [25]:
```
f(1)
```
Out[25]:
```
3
```

You can stop the Time Machine from saving.

In [26]:
```
TM.stop_saving()
```
Out[26]:
```
save_state (generic function with 1 method)
```

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

In [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:121
 [3] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091
```

And, you can make it start saving again.

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

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

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

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

In [33]:
```
TM.clear_past([30])
```

In [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:121
 [3] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091
```

If you want to see exactly what was stored, or exactly what variables are being overwritten, look at `TM.past`.

In [35]:
```
TM.past[23].vars
```
Out[35]:
```
Dict{Any,Any} with 4 entries:
  :x    => 2
  :y    => 4
  :cell => 22
  :vec  => [1, 2, 3, 4]
```

In [36]:
```
TM.past[23].ans
```
Out[36]:
```
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 work on sandboxed variables. They start by copying all variables the exist when they are called. But, they do not change the values of any variables. To access the values of the variables they change, 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, I will simulate the delay of a long-running job with `sleep`. As you will see, results will change after jobs finish.

In [37]:
```
x = collect(1:3)
y = 1
```
Out[37]:
```
1
```

In [38]:
```
t0 = time()
n = IJulia.n
TM.@thread begin
    y = y + 1
    push!(x,y)
    sleep(5)
    sum(y)
end
```
Out[38]:
```
Task (runnable) @0x000000011f426ad0
```

In [39]:
```
println("After $(time() - t0) seconds.")
Out[n]
```
```
After 0.7151029109954834 seconds.
```
Out[39]:
```
Task (runnable) @0x000000011f426ad0
```

In [40]:
```
sleep(10)
```

In [41]:
```
println("After $(time() - t0) seconds.")
Out[n]
```
```
After 10.80537486076355 seconds.
```
Out[41]:
```
2
```

In [42]:
```
x, y
```
Out[42]:
```
([1, 2, 3], 1)
```

In [43]:
```
TM.@past n
x, y
```
Out[43]:
```
([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.

In [44]:
```
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!

In [45]:
```
function intense(n)
    sleep(n)
    println("I slept for $(n) seconds!")
    n^2
end
```
Out[45]:
```
intense (generic function with 1 method)
```

In [46]:
```
t0 = time()
n1 = IJulia.n
TM.@thread intense(10)
```
Out[46]:
```
Task (runnable) @0x000000011de95f90
```

In [47]:
```
println("After $(time()-t0) seconds")
n2 = IJulia.n
TM.@thread intense(11)
```
```
After 0.03081798553466797 seconds
```
Out[47]:
```
Task (runnable) @0x000000011de961d0
```

In [48]:
```
println("After $(time()-t0) seconds")
n3 = IJulia.n
TM.@thread intense(2)
```
```
After 0.1841750144958496 seconds
```
Out[48]:
```
Task (runnable) @0x000000011de96410
```

In [49]:
```
println("After $(time()-t0) seconds")
TM.running
```
```
After 0.2656099796295166 seconds
```
Out[49]:
```
Set{Any} with 4 elements:
  47
  48
  44
  46
```

In [50]:
```
sleep(3)
```
```
I slept for 2 seconds!
```

In [51]:
```
println("After $(time()-t0) seconds")
TM.running
```
```
After 4.430108070373535 seconds
```
Out[51]:
```
Set{Any} with 3 elements:
  47
  44
  46
```

In [52]:
```
println("After $(time()-t0) seconds")
TM.finished
```
```
After 4.483709096908569 seconds
```
Out[52]:
```
2-element Array{Int64,1}:
 38
 48
```

In [53]:
```
sleep(10)
println("After $(time()-t0) seconds")
TM.running
```
```
I slept for 10 seconds!
I slept for 11 seconds!
After 14.552716970443726 seconds
```
Out[53]:
```
Set{Any} with 1 element:
  44
```

In [54]:
```
Out[n1], Out[n2], Out[n3]
```
Out[54]:
```
(100, 121, 4)
```

I don't know why cell 44 is listed as running. That must be a bug!

In [57]:
```
TM.running
```
Out[57]:
```
Set{Any} with 1 element:
  44
```

In [56]:
```
Out[44]
```
```
KeyError: key 44 not found

Stacktrace:
 [1] getindex(::Dict{Int64,Any}, ::Int64) at ./dict.jl:467
 [2] top-level scope at In[56]:1
 [3] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091
```

In [ ]:
```

```