92

Useful magic commands in Jupyter Notebook/Lab

30 mins read

Jupyter Notebook/Lab is the go-to tool used by data scientists and developers worldwide to perform data analysis nowadays. It provides a very easy-to-use interface and lots of other functionalities like markdown, latex, inline plots, etc. Apart from these, it even provides a list of useful magic commands which let us perform a bunch of tasks from the Jupyter notebook itself which developers need to do in the command prompt/shell. As a part of this tutorial, I'll cover some of the very commonly used magic commands. You can find the Jupyter file for the examples of this post on my Github:

https://gist.github.com/iamirmasoud/f9287458be73d4903131e754ce423821

There are two types of magic commands available with Jupyter Notebook/Lab:

- **Line Magic Commands:** It applies the command to one line of the Jupyter cell as its name suggests.
- **Cell Magic Commands:** It applies the command to the whole cell of the notebook and needs to be kept at the beginning of the cell.

I'll now explain the usage of magic commands one by one with simple examples.

Line Magic Commands

In this section, I will explain the commonly used line magic command which can make the life of the developer easy by providing some of the useful functionalities in the notebook itself.

`%lsmagic`

The `%lsmagic` commands list all the available magic commands with a notebook.

```
%lsmagic
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%ja

Automagic is ON, % prefix IS NOT needed for line magics.
```

If I run this command in the Jupyter lab then it'll return an expandable tree-like structure for a list of commands as shown below.

```
    HTML: "Other"
    SVG: "Other"
    bash: "Other"
    capture: "ExecutionMagics"
    debug: "ExecutionMagics"
    file: "Other"
    html: "DisplayMagics"
    javascript: "DisplayMagics"
    js: "DisplayMagics"
    latex: "DisplayMagics"
    markdown: "DisplayMagics"
    perl: "Other"
    prun: "ExecutionMagics"
    pypy: "Other"
    python: "Other"
    python2: "Other"
    python3: "Other"
    ruby: "Other"
    script: "ScriptMagics"
    sh: "Other"
    svg: "DisplayMagics"
    sx: "OSMagics"
    system: "OSMagics"
    time: "ExecutionMagics"
    timeit: "ExecutionMagics"
    writefile: "OSMagics"
▸ line: {} 97 keys
```


%magic

The `%magic` commands print information about the magic commands system in the Jseaupyter notebook. It kind of gives an overview of the magic commands system available in the notebook.

```
%magic
```

%quickref

The `%quickref` line command gives us a cheat sheet covering an overview of each magic command available.

```
%quickref
```

```
The following magic functions are currently available:

%alias:
    Define an alias for a system command.
%alias_magic:
    ::
%autoawait:

%autocall:
    Make functions callable without having to type parentheses.
%automagic:
      Make magic functions callable without having to type the
initial %.
%autosave:
```

%alias_magic

The `%alias_magic` line command as its name suggests creates an alias for any existing magic command. We can then call the command by alias and it will perform the same functionality as the original command. Below I have renamed the `%pwd` command to the `%currdir` command which displays the current working directory. We need to give a new name for the command followed by a command name to create an alias.

```
%alias_magic currdir pwd
```

```
Created `%currdir` as an alias for `%pwd`.
```

```
%currdir
```

```
'/home/sunny'
```

%autocall

The `%autocall` line command lets us call functions in a notebook without typing parenthesis. We can type the function name followed by a list of argument values separated by a comma. Below I have created a simple function that adds two numbers. I have then turned on `autocall` by calling the magic command. After turning on `autocall`, we are able to execute the function without parenthesis. I have then turned off `autocall` and calling the function without parenthesis fails.

```
def addition(a,b):
    return a+b
```

```
%autocall
```

```
Automatic calling is: Smart
```

```
addition 5, 5
```

```
------> addition(5, 5)
------> addition(5, 5)
```

```
%autocall
```

```
Automatic calling is: OFF
```

```
addition 5, 5
```

```
  File "<ipython-input-12-4d245131862c>", line 1
    addition 5, 5
             ^
SyntaxError: invalid syntax
```

%automagic

The `%automagic` line command let us call magic command in Jupyter notebook without typing `%` sign at the beginning. I can turn `automagic` on and off by executing the `%automagic` line command. Below I have explained the usage of the same.

```
%automagic
```

```
Automagic is OFF, % prefix IS needed for line magics.
```

```
pwd
```

```
---------------------------------------------------------------
----------
NameError                                 Traceback (most recent
call last)
<ipython-input-14-86938b1e80ee> in <module>
----> 1 pwd

NameError: name 'pwd' is not defined
```

```
%automagic
```

```
Automagic is ON, % prefix IS NOT needed for line magics.
```

```
'/home/sunny'
```

%pwd

The `%pwd` line command as its name suggests returns the present working directory.

```
%pwd
```

```
'/home/sunny'
```

%cd

The `%cd` line command lets us change our working directory as explained below.

```
%cd Desktop/
```

```
/home/sunny/Desktop
```

```
%pwd
```

```
'/home/sunny/Desktop'
```

%system

The `%system` command lets us execute Unix shell commands in the Jupyter notebook. I can execute any single line Unix shell command from the notebook. I have explained below the usage of the command with two simple examples.

```
%system echo 'Hello World'
```

```
['Hello World']
```

```
%system ls -lrt | grep python
```

```
['-rw-r--r--  1 sunny sunny   3393723 Mar  4  2020 How to build dashboard using Python (Dash & Plotly) and deploy online (pythonanywhere.com).html~']
```

%sx

The `%sx` command works exactly like the `%system` command.

```
%sx echo 'Hello World'
```

```
['Hello World']
```

```
%sx ls -lrt | grep python
```

```
['-rw-r--r--  1 sunny sunny   3393723 Mar  4  2020 How to build dashboard using Python (Dash & Plotly) and deploy online (pythonanywhere.com).html~']
```

%time

The `%time` line command measures the execution time of the line which follows it using the `time` python module. It returns both, the CPU and wall time of execution. Below I have explained with a simple example of how to use the command. It even returns the execution value of the command which we have kept in a variable. It's available as a cell command as well.

```
%time out = [i*i for i in range(1000000)]
```

```
CPU times: user 44.9 ms, sys: 11.3 ms, total: 56.3 ms
Wall time: 55.2 ms
```

%timeit

The `%timeit` line command measures the execution time of the function using the `timeit` python module. It provides a few other functionalities as well. It executes the command given as input for 7 rounds where each round executes code 10 times totaling 70 times by default. It takes the best of each iteration in each round and gives time measurement with standard deviation. Below are some useful arguments for the command.

- `-n <loops>` – It accepts integer values specifying the number of iterations per round.
- `-r <runs>` – It accepts integer values specifying the number of rounds to test the timer.
- `-t` – This option forces `%timeit` to use `time.time` to measure time which returns wall time.
- `-c` – This option forces `%timeit` to use `time.clock` to measure time which returns CPU time.
- `-q` – This option instructs `%timeit` to not print results to the output.
- `-o` – This option returns `TimeitResult` object.

Below I have explained with simple example usage of the command.

```
%timeit out = [i*i for i in range(1000000)]
```

```
53.1 ms ± 1.27 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit -n 5 -r 5 out = [i*i for i in range(1000000)]
```

```
52.2 ms ± 1.87 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

%who

The `%who` line command returns all variables of a particular type. We can give a variable type followed the command and it'll return a list of all variables with that type. We can even give more than one type if we want to see variables of different types which are currently active in the Jupyter notebook and not collected by the garbage collector.

Below I have explained with a few simple examples of how we can use `%who`.

```
a = 100
b = 5.5
c = [11,12,13]
d = {'key1':'val1', 'key2':'val2'}
e = "Hello World"

%who str
%who dict
%who float
%who list
```

```
e
d
b
c
```

```
def addition(a,b):
    return a + b

def division(a,b):
    return a / b if b!=0 else 0

%who function
```

```
addition         division
```

```
%who function list
```

%who_ls

The `%who_ls` commands work exactly like `%who` but it returns a list of variable names as a list of strings which is sorted as well.

```
%who_ls function
```

```
['addition', 'division']
```

%whos

The `%whos` command also works like `%who` but it gives a little more information about variables that match the given type.

```
a = 100
b = 5.5
c = [11,12,13]
d = {'key1':'val1', 'key2':'val2'}
e = "Hello World"

%whos str
%whos dict
%whos float
%whos list
```

```
Variable   Type    Data/Info
---------------------------
e          str     Hello World
Variable   Type    Data/Info
---------------------------
d          dict    n=2
Variable   Type     Data/Info
-----------------------------
b          float    5.5
Variable   Type    Data/Info
---------------------------
c          list    n=3
out        list    n=1000000
```

```
%whos function
```

```
Variable   Type        Data/Info
--------------------------------
addition   function    <function addition at 0x7fdedc575620>
division   function    <function division at 0x7fdedc63ba60>
```

%load

The `%load` command accepts the filename followed by it and loads the code present in that file in the current cell. It also comments execution of itself once the cell is executed. It can even accept URLs where the code is kept and loads it from there.

We can use the below-mentioned options along with the command if we want to load only a particular part of the file and not the whole file.

- **-r** – It accepts integer range which only loads code that falls into that range in the file.
- **-s** – It accepts class or function name and loads code of that class or function into the cell rather than the whole file.

Below I have explained the usage of the command with simple examples.

```
# %load profiling_example.py

from memory_profiler import profile

@profile(precision=4)
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

if __name__ == "__main__":
    main_func()
```

```
# %load -r 5-10 profiling_example.py
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
```

```
# %load -s main_func profiling_example.py
@profile(precision=4)
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)
```

%load_ext

The `%load_ext` commands load any external module library which can then be used as a magic command in a notebook. Please make a note that only a few libraries that have implemented support for Jupyter Notebook can be loaded. The `snakeviz`, `line_profiler`, and `memory_profiler` are examples of it.

Below I have loaded `snakeviz` as an extension in a notebook. We can then use `%snakeviz` to profile a line of code and visualize it.

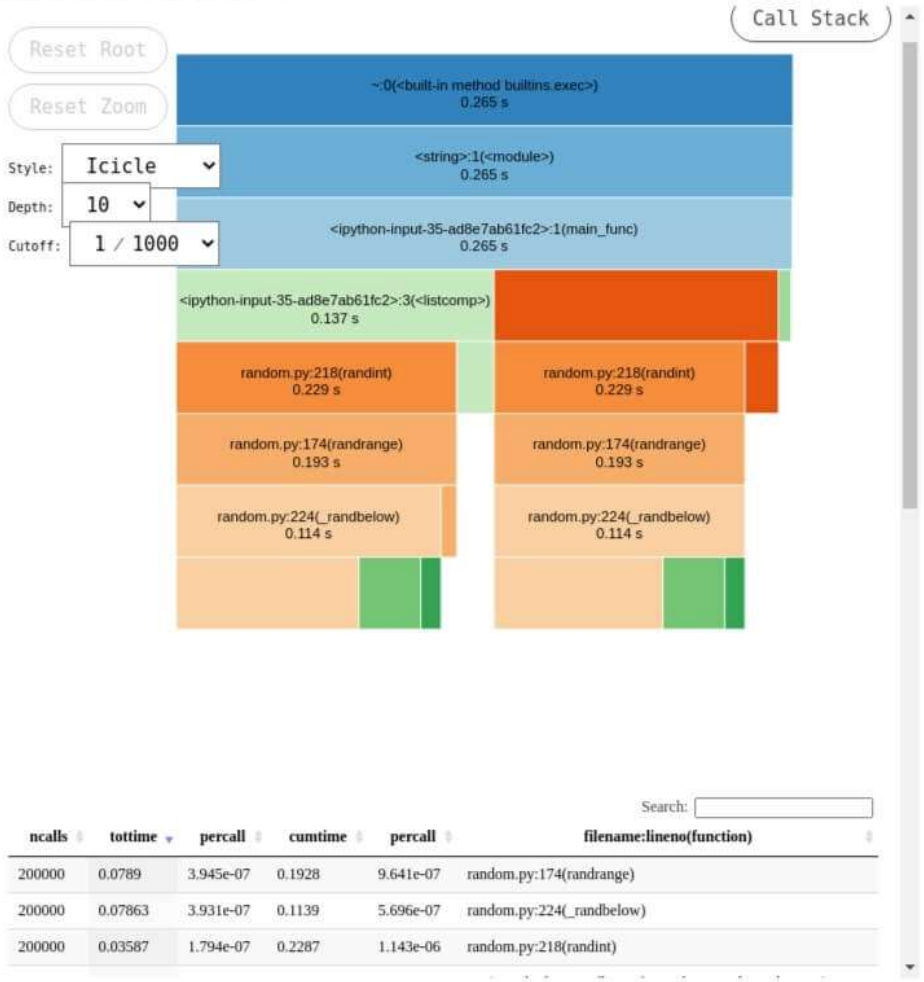
If you are interested in learning about how to use `snakeviz`, `line_profiler`, and `memory_profiler` with Jupyter notebook then please feel free to check out tutorials on the same.

- Snakeviz – Visualize Profiling Results in Python
- line_profiler – Line by Line Profiling of Python Code
- How to profiler memory usage in python using memory_profiler?

```
%load_ext snakeviz
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
```

```
%snakeviz main_func()
```

```
1098377

*** Profile stats marshalled to file '/tmp/tmp5l2nnobm'.
Embedding SnakeViz in this document...
```



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
200000	0.0789	3.945e-07	0.1928	9.641e-07	random.py:174(randrange)
200000	0.07863	3.931e-07	0.1139	5.696e-07	random.py:224(_randbelow)
200000	0.03587	1.794e-07	0.2287	1.143e-06	random.py:218(randint)

%unload_ext

The `%unload_ext` line command unloads any external loaded extension.

```
%unload_ext snakeviz
```

```
The snakeviz extension doesn't define how to unload it.
```

%reload_ext

The `%reload_ext` line command reloads externally loaded extension. We can reload it if it misbehaves.

```
%reload_ext snakeviz
```

%tb

The `%tb` command stack trace of the last failure which had happened in the notebook.

```
%tb
```

```
NameError                                 Traceback (most recent
call last)
~/anaconda3/lib/python3.7/site-
packages/IPython/core/interactiveshell.py  in  find_user_code(self,
target, raw, py_only, skip_encoding_cookie, search_ns)
              3736                                                try:
# User namespace
-> 3737             codeobj = eval(target, self.user_ns)
   3738         except Exception:

<string> in <module>
NameError: name 'profiling_example' is not defined
During  handling  of  the  above  exception,  another  exception
occurred:
ValueError                                Traceback (most recent
call last)
<ipython-input-31-f6e2b7c9668b> in <module>
---->            1            get_ipython().run_line_magic('load',
'profiling_example.py')

~/anaconda3/lib/python3.7/site-
packages/IPython/core/interactiveshell.py  in  run_line_magic(self,
magic_name, line, _stack_depth)
      2325                                    kwargs['local_ns']  =
self.get_local_scope(stack_depth)
   2326             with self.builtin_trap:
-> 2327                 result = fn(*args, **kwargs)
   2328             return result
   2329
<decorator-gen-46> in load(self, arg_s)
~/anaconda3/lib/python3.7/site-packages/IPython/core/magic.py   in
<lambda>(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):
~/anaconda3/lib/python3.7/site-
packages/IPython/core/magics/code.py in load(self, arg_s)
    331         search_ns = 'n' in opts
    332
-->  333               contents  =  self.shell.find_user_code(args,
search_ns=search_ns)
    334
    335         if 's' in opts:
~/anaconda3/lib/python3.7/site-
packages/IPython/core/interactiveshell.py  in  find_user_code(self,
target, raw, py_only, skip_encoding_cookie, search_ns)
   3738         except Exception:
    3739                 raise ValueError(("'%s' was not found in
history, as a file, url, "
-> 3740                                   "nor in the user
namespace.") % target)
   3741
   3742         if isinstance(codeobj, str):

ValueError: 'profiling_example.py' was not found in history, as a
file, url, nor in the user namespace.
```

%env

The %env line command can be used to get, set, and list environment variables. If we call the command without any argument then it'll list all environment variables. We can give the environment variable name followed by the command and it'll return the value of that environment variable. We can also set the value of the environment variable using it which I have explained with an example below.

```
%env DESKTOP_SESSION
```

```
'ubuntu'
```

```
%env HOME
```

```
'/home/sunny'
```

```
%env HOME=/home/sunny/Desktop
```

```
env: HOME=/home/sunny/Desktop
```

```
%env HOME
```

```
'/home/sunny/Desktop'
```

%set_env

The `%set_env` command lets us set the value of environment variables.

```
%set_env HOME=/home/sunny
```

```
env: HOME=/home/sunny
```

```
%env HOME
```

```
'/home/sunny'
```

```
home = "/home/sunny/Desktop/"
%set_env HOME=$home
```

```
%env HOME
```

```
'/home/sunny/Desktop/'
```

%conda

The `%conda` line command allows us to execute the Conda package manager command in the Jupyter notebook. Below I am listing down a list of available Conda environments on the system.

```
%conda env list
```

```
# conda environments:
#
base                  *  /home/sunny/anaconda3
py27                     /home/sunny/anaconda3/envs/py27
py37                     /home/sunny/anaconda3/envs/py37
scalene_env              /home/sunny/anaconda3/envs/scalene_env

Note: you may need to restart the kernel to use updated packages.
```

%pip

The `%pip` line command lets us install the python module using the pip package manager in the Jupyter notebook.

```
%pip install sklearn
```

```
Requirement already satisfied: sklearn in
./anaconda3/lib/python3.7/site-packages (0.0)
Requirement already satisfied: scikit-learn in
./anaconda3/lib/python3.7/site-packages (from sklearn) (0.21.2)
Requirement already satisfied: numpy>=1.11.0 in
./anaconda3/lib/python3.7/site-packages (from scikit-learn-
>sklearn) (1.17.1)
Requirement already satisfied: scipy>=0.17.0 in
./anaconda3/lib/python3.7/site-packages (from scikit-learn-
>sklearn) (1.4.1)
Requirement already satisfied: joblib>=0.11 in
./anaconda3/lib/python3.7/site-packages (from scikit-learn-
>sklearn) (0.13.2)
Note: you may need to restart the kernel to use updated packages.
```

%dhist

The `%dhist` command lists down all directory which was visited in the notebook. It shows the history of directories visited.

```
Directory history (kept in _dh)
0: /home/sunny
1: /home/sunny/Desktop
2: /home/sunny
```

%history

The `%history` line command list down the history of commands which were executed in a notebook. We can use the `-n` option to show commands which fall in a particular range in history. We can even store a history of commands executed to an output file using the `-f` option followed by the file name.

```
%history -n 6-9
```

```
   6: %alias_magic currdir pwd
   7: %currdir
   8:
def addition(a,b):
    return a+b
   9: %autocall
```

```
%history -n 6-9 -p
```

```
   6: >>> %alias_magic currdir pwd
   7: >>> %currdir
   8:
>>> def addition(a,b):
...     return a+b
...
   9: >>> %autocall
```

```
%history -n 6-9 -p -f magic.out
```

```
File 'magic.out' exists. Overwrite? y
Overwriting file.
```

```
!cat magic.out
```

```
   6: >>> %alias_magic currdir pwd
   7: >>> %currdir
   8:
>>> def addition(a,b):
```

%doctest_mode

The `%doctest_mode` line command informs the IPython kernel to behave as much as a normal python shell which will influence how it asks for values and prints output.

```
%doctest_mode
```

```
Exception reporting mode: Plain
Doctest mode is: ON
```

```
a=10
a/0
```

```
Traceback (most recent call last):

  File "<ipython-input-53-f9c81372a7d7>", line 3, in <module>
    a/0

ZeroDivisionError: division by zero
```

```
%doctest_mode
```

```
Exception reporting mode: Context
Doctest mode is: OFF
```

%prun

The `%prun` command lets us profile python code in Jupyter notebook using the `profile` module. It lists down the time taken by various functions. Here is a tutorial on profile module.

It has a list of the below options which can be useful for different tasks.

- `-l` – This option accepts an integer argument followed by it which will restrict the number of lines of profiling output printed to standard output.
- `-s` – This option accepts the string argument followed by it and will sort the profiling argument based on that string. The string is one of the column names of the profiling output. Below is a list of possible values.
  - calls
  - tottime
  - cumulative
  - file

- name
- line

- `-T` – This option saves profiling results to a file. We need to give the file name after this option to save the output to it.
- `-q` – This option prevents printing of profiling results to standard output.

Below I have explained the usage of `%prun` with simple examples. It's also available as a cell command.

```
import random
%prun arr1 = [random.randint(1,10) for i in range(100000)]
```

```
559668 function calls in 0.140 seconds

   Ordered by: internal time

         ncalls      tottime      percall      cumtime      percall
filename:lineno(function)
      100000         0.042         0.000         0.102         0.000
random.py:174(randrange)
      100000         0.042         0.000         0.060         0.000
random.py:224(_randbelow)
      100000         0.019         0.000         0.122         0.000
random.py:218(randint)
               1        0.018        0.018        0.140        0.140
<string>:1(<listcomp>)
     159664       0.014       0.000       0.014       0.000  {method
'getrandbits' of '_random.Random' objects}
     100000       0.005       0.000       0.005       0.000  {method
'bit_length' of 'int' objects}
               1        0.000        0.000        0.140        0.140
<string>:1(<module>)
          1     0.000     0.000     0.140     0.140 {built-in method
builtins.exec}
          1     0.000     0.000     0.000     0.000 {method 'disable'
of '_lsprof.Profiler' objects}
```

```
%prun    -l    10    -s    tottime    -T    prof_res.out    arr1    =
[random.randint(1,10) for i in range(100000)]
```

```
*** Profile printout saved to text file 'prof_res.out'.
```

```
!cat prof_res.out
```

```
         560268 function calls in 0.159 seconds

   Ordered by: internal time
```

```
      100000         0.048         0.000         0.116         0.000
random.py:174(randrange)
      100000         0.047         0.000         0.068         0.000
random.py:224(_randbelow)
      100000         0.023         0.000         0.139         0.000
random.py:218(randint)
           1         0.020         0.020         0.159         0.159
<string>:1(<listcomp>)
      160264       0.016       0.000       0.016       0.000  {method
'getrandbits' of '_random.Random' objects}
      100000       0.006       0.000       0.006       0.000  {method
'bit_length' of 'int' objects}
           1     0.000     0.000     0.159     0.159 {built-in method
builtins.exec}
           1         0.000         0.000         0.159         0.159
<string>:1(<module>)
           1     0.000     0.000     0.000     0.000 {method 'disable'
of '_lsprof.Profiler' objects}
```

%matplotlib

The `%matplotlib` line command sets up which backend to use to plot matplotlib plots. We can execute a command with the –list option and it'll return a list of available backend strings. If I call the command without any argument then it'll be set `TkAgg` as backend.

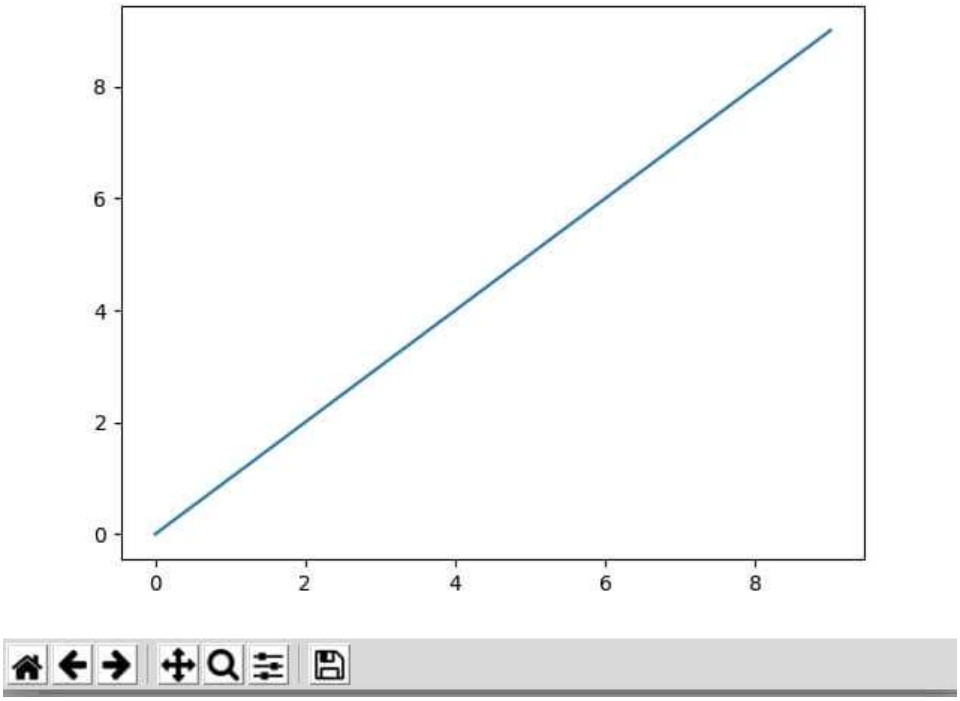
```
%matplotlib --list
```

```
Available matplotlib backends: ['tk', 'gtk', 'gtk3', 'wx', 'qt4',
'qt5', 'qt', 'osx', 'nbagg', 'notebook', 'agg', 'svg', 'pdf',
'ps', 'inline', 'ipympl', 'widget']
```

```
%matplotlib
```

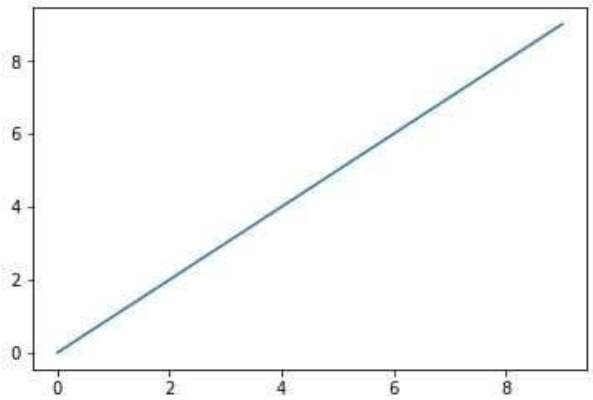
```
Using matplotlib backend: TkAgg
```

```
import matplotlib.pyplot as plt

plt.plot(range(10), range(10))
plt.show()
```



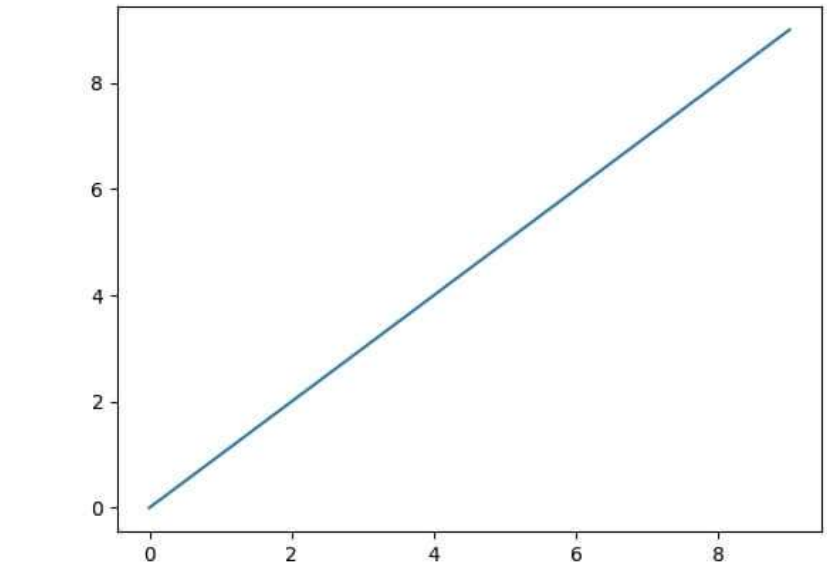

```
%matplotlib inline
```

```
import matplotlib.pyplot as plt

plt.plot(range(10), range(10))
plt.show()
```




```
%matplotlib widget
#%matplotlib notebook
```

```
import matplotlib.pyplot as plt
plt.plot(range(10), range(10))
plt.show()
```



%pdef

The `%pdef` command prints the signature of any callable object. We can inspect the signature of functions using this line command which can be useful if a signature is quite long.

```
def division(a, b):
    return a / b if b!=0 else 0
%pdef division
```

```
division(a, b)
```

%pdoc

The `%pdoc` line command prints docstring of callable objects. We can print a docstring of the function which has a general description of arguments and inner working of the function.

```
def division(a, b):
    '''
    This function divides first argument by second.
    It return 0 if second argument is 0 to avoid divide by zero errc
    '''
    return a / b if b!=0 else 0
%pdoc division
```

```
Class docstring:
    This function divides first argument by second.
    It return 0 if second argument is 0 to avoid divide by zero errc
Call docstring:
    Call self as a function.
```

%precision

The `%precision` line command sets the precision of printing floating-point numbers. We can specify how many numbers to print after the decimal point. It'll round the

number.

```
%precision 3
```

```
'%.3f'
```

```
a = 1.23678
a
```

```
1.237
```

```
%precision 0
```

```
'%.0f'
```

```
a
```

```
1
```

%psearch

The `%psearch` line command lets us search namespace to find a list of objects which match the wildcard argument given to it. We can search for variable names that have some string present in them using this command. I have explained the usage of the command below.

```
val1 = 10
val2 = 20
val3 = 50
top_val = 10000

%psearch val*
```

```
val1
val2
val3
```

```
%psearch *val*
```

```
eval
top_val
val1
val2
val3
```

The `%psource` command takes any object as input and prints the source code of it. Below I am using it to print the source code of the division function I had created earlier.

```
%psource division
def division(a, b):
    '''
    This function divides first argument by second.
    It return 0 if second argument is 0 to avoid divide by zero errc
    '''
    return a / b if b!=0 else 0
```

%pycat

The `%pycat` line command shows us a syntax-highlighted file which is given as input to it.

```
%pycat profiling_example.py
```

```
from memory_profiler import profile

@profile(precision=4)
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

if __name__ == "__main__":
    main_func()
```

%pylab

The `%pylab` command loads NumPy and matplotlib to work into the namespace. After executing this command, we can directly call the numpy and matplotlib functions without needing to import these libraries. I have explained the usage below.

```
%pylab
```

```
Using matplotlib backend: TkAgg
Populating the interactive namespace from numpy and matplotlib
```

```
arr = array([1,2,3,4])
type(arr)
```

```
numpy.ndarray
```

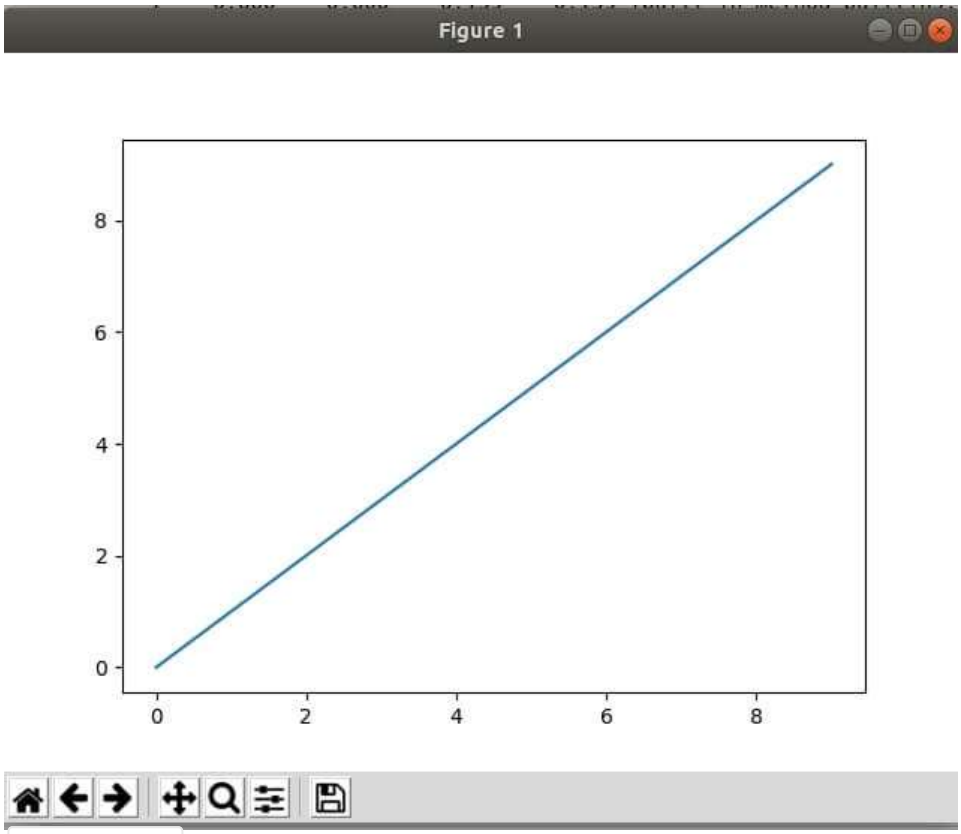
```
type(rng)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
numpy.ndarray
```

```
plot(range(10), range(10))
```

```
[<matplotlib.lines.Line2D at 0x7f9ee61ec588>]
```



%recall

The `%recall` command puts a history of the command executed in the next cell. I can give it an input with a range of integer and it'll put that many commands from history in the next cell.

```
%recall 4
```

```
%quickref
```

```
%recall 1-4
```

```
%magic prun
%quickref
```

%rerun

The `%rerun` command reruns the previously executed cell.

```
%rerun
```

```
=== Executing: ===
%recall 1-4
=== Output: ===
```

```
%lsmagic
%magic prun
%magic prun
%quickref
```

%reset

The `%reset` command resets namespace by removing all user-defined names.

```
%who_ls
```

```
['a',
 'addition',
 'arr',
 'arr1',
 'b',
 'c',
 'd',
 'main_func',
 'out',
 'profile',
 'rng',
 'top_val',
 'val1',
 'val2',
 'val3']
```

```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

```
---------------------------------------------------------------
----------
NameError                                 Traceback (most recent
call last)
<ipython-input-87-9b02d54fbb1e> in <module>
----> 1 val1

NameError: name 'val1' is not defined
```

%reset_selective

The `%reset_selective` works like `%reset` but let us specify a pattern to remove only names that match that pattern. Below I am only removing variables that have the string `val` in their name.

```
%who_ls
```

```
[]
```

```
val1 = 10
val2 = 20
val3 = 50
top_val = 10000
a = 10
rng = range(10)
%who_ls
```

```
['a', 'rng', 'top_val', 'val1', 'val2', 'val3']
```

```
%reset_selective -f  val
```

```
%who_ls
```

```
['a', 'rng']
```

%run

The `%run` command lets us run the python file in the Jupyter notebook. We can also pass arguments to it followed by a file name as we do from shell/command prompt. I have created a simple profiling example mentioned below and run it for explanation purposes. It also accepts the `-t` option which measures the running time of the file.

profiling_example.py

```
@profile(precision=4)
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

if __name__ == "__main__":
    main_func()
```

```
%run profiling_example.py
```

```
1102300
Filename: /home/sunny/profiling_example.py

Line #    Mem usage    Increment   Line Contents
================================================
     4 105.6602 MiB 105.6602 MiB   @profile(precision=4)
     5                             def main_func():
     6 105.6602 MiB   0.0000 MiB       import random
          7  105.6602  MiB      0.0000  MiB              arr1  =
[random.randint(1,10) for i in range(100000)]
          8  105.6602  MiB      0.0000  MiB              arr2  =
[random.randint(1,10) for i in range(100000)]
      9 105.6602 MiB    0.0000 MiB        arr3 = [arr1[i]+arr2[i]
for i in range(100000)]
    10 105.6602 MiB   0.0000 MiB       tot = sum(arr3)
    11 105.6602 MiB   0.0000 MiB       print(tot)
```

```
%run -t profiling_example.py
```

```
1100017
Filename: /home/sunny/profiling_example.py

Line #    Mem usage    Increment   Line Contents
================================================
     4 105.6602 MiB 105.6602 MiB   @profile(precision=4)
     5                             def main_func():
     6 105.6602 MiB   0.0000 MiB       import random
          7  105.6602  MiB      0.0000  MiB              arr1  =
[random.randint(1,10) for i in range(100000)]
          8  105.6602  MiB      0.0000  MiB              arr2  =
[random.randint(1,10) for i in range(100000)]
      9 105.6602 MiB    0.0000 MiB        arr3 = [arr1[i]+arr2[i]
for i in range(100000)]
    10 105.6602 MiB   0.0000 MiB       tot = sum(arr3)
    11 105.6602 MiB   0.0000 MiB       print(tot)

IPython CPU timings (estimated):
  User   :      14.68 s.
```

Cell Magic Commands

Cell magic commands are given at the starting of the cell and applied to the whole cell. It can be very useful when we want to perform some functionality at the cell level like measuring the running time of cells or profiling cell code. I'll now explain useful cell commands available in the Jupyter notebook.

%%bash

The `%%bash` cell command lets us execute shell commands from the Jupyter notebook. We can include the whole shell script into the cell and it'll execute it like it was executed in a shell.

```
%%bash

ls -lrt | grep "python"
```

```
-rw-r--r--  1 sunny sunny  3393723 Mar  4  2020 How to build dashboard using Python (Dash & Plotly) and deploy online (pythonanywhere.com).html~
```

%%script

The `%%script` cell command lets us execute scripts designed in different languages like Perl, python, ruby, and Linux shell scripting. We need to give the language name followed by the command and it'll execute shell contents using the interpreter of that language.

```
%%script bash
ls -lrt | grep "python"
```

```
-rw-r--r--  1 sunny sunny  3393723 Mar  4  2020 How to build dashboard using Python (Dash & Plotly) and deploy online (pythonanywhere.com).html~
```

```
%%script sh
ls -lrt | grep "python"
```

```
-rw-r--r--  1 sunny sunny  3393723 Mar  4  2020 How to build dashboard using Python (Dash & Plotly) and deploy online (pythonanywhere.com).html~
```

%%sh

The `%%sh` cell command lets us execute UNIX shell commands into the Jupyter notebook.

```
-rw-r--r--  1 sunny sunny   3393723 Mar  4  2020 How to build
dashboard using Python (Dash & Plotly) and deploy online
(pythonanywhere.com).html~
```

%%html

The `%%html` cell command renders the contents of the cell as HTML. We can keep HTML tags as input and it'll render them as HTML.

```
%%html
<h1>Heading Big</h1>
<h2>Heading Medium</h2>
<h3>Heading Small</h3>
# Comment
```

Heading Big

Heading Medium

Heading Small

Comment

%%javascript

The `%%javascript` cell command will execute the contents of the cell as javascript. Below I have explained how we can use this cell command with a simple example. The output cell is available as `element` and we can modify it to append HTML. Please make a note that this command currently works only with Jupyter Lab, not with Jupyter Notebook.

```
%%javascript
// program to find the largest among three numbers
// take input from the user
const num1 = 12
const num2 = 10
const num3 = 35
let largest;
// check the condition
if(num1 >= num2 && num1 >= num3) {
    largest = num1;
}
else if (num2 >= num1 && num2 >= num3) {
    largest = num2;
}
else {
    largest = num3;
}
// display the result
element.innerHTML =  '<h1>The Largest Number is : ' + largest +
'</h1>'
```

%%js

The `%%js` cell command works exactly like `%%javascript`.

```
%%js

// program to find the largest among three numbers

// take input from the user
const num1 = 12
const num2 = 10
const num3 = 35
let largest;

// check the condition
if(num1 >= num2 && num1 >= num3) {
    largest = num1;
}
else if (num2 >= num1 && num2 >= num3) {
    largest = num2;
}
else {
    largest = num3;
}
// display the result
element.innerHTML   = '<h1>The Largest Number is : ' + largest +
'</h1>';
```

The Largest Number is : 35

%%perl

The `%%perl` cell command executes cell content using Perl interpreter. We can use this command to execute Perl script in Jupyter notebook.

```
%%perl
#!/usr/bin/perl
use strict;
use warnings;
print "Hello Bill\n";
```

```
Hello Bill
```

```
%%perl
$size=15;              # give $size value of 15
$y = -7.78;            # give $y value of -7.78
$z = 6 + 12;
print $y, "\n";
print $z, "\n";
print $size, "\n";
$num = 7;
```

```
-7.78
18
15
It is 7
```

%%ruby

The `%%ruby` cell command executes cell content using a ruby interpreter. We can use this command to execute the ruby script in the Jupyter notebook.

```
%%ruby
print "Hello, World!"
```

```
Hello, World!
```

```
%%ruby
tigers = 50
lions = 45
puts "There are #{tigers} Tigers and #{lions} Lions in the Zoo."
```

```
There are 50 Tigers and 45 Lions in the Zoo.
```

%%latex

The `%%latex` cell command lets us execute cell content as latex code. We can write latex code and it'll create formulas out of it. I have explained the usage of the same below with simple examples.

```
%%latex
\begin{equation*}
   e^{\pi i} + 1 = 0
\end{equation*}
```

$$e^{\pi i} + 1 = 0$$

```
%%latex
$
idf(t)  = {\log_{} \dfrac {n_d} {df(d,t)}} + 1
$
```

$$idf(t) = \log \frac{n_d}{df(d,t)} + 1$$

%%markdown

The `%%markdown` cell command lets us execute cell contents as markdown.

```
# H2 Heading
* List 1
* List 2
**Bold Text**
```

H1 Heading

H2 Heading

- List 1
- List 2

Bold Text

%%writefile

The `%%writefile` cell command lets us save the contents of the cell to an output file.

```
%%writefile profiling_example.py
from memory_profiler import profile

@profile(precision=4)
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

if __name__ == "__main__":
    main_func()
```

```
Writing profiling_example.py
```

```
!cat profiling_example.py
```

```
from memory_profiler import profile

@profile(precision=4)
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

if __name__ == "__main__":
    main_func()
```

%%time

The `%%time` cell command works exactly like the `%time` line command but measures the time taken by code in the cell.

```
%%time
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

main_func()
```

```
1101325
CPU times: user 162 ms, sys: 6.25 ms, total: 169 ms
Wall time: 171 ms
```

%%timeit

The `%%timeit` cell command works exactly like the `%timeit` line command but measures the time taken by code in the cell.

```
%%timeit
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
main_func()
```

```
153 ms ± 3.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit -n 5 -r 5
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
main_func()
```

```
153 ms ± 1.82 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

```
%%timeit -t -n 5 -r 5
def main_func():
```

```
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)

main_func()
```

```
157 ms ± 2.99 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

```
%%timeit -c -n 5 -r 5
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
main_func()
```

```
152 ms ± 2.97 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

%%prun

The `%%prun` cell command profiles code of the cell exactly like the `%prun` profiles one line of code.

```
%%prun
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)

main_func()
```

```
1098986
```

```
         1119655 function calls in 0.271 seconds

   Ordered by: internal time

         ncalls      tottime      percall      cumtime      percall
filename:lineno(function)
       200000        0.082        0.000        0.196        0.000
random.py:174(randrange)
       200000        0.079        0.000        0.115        0.000
random.py:224(_randbelow)
       200000        0.037        0.000        0.233        0.000
```

```
random.py:218(randint)
     2    0.000    0.000    0.027    0.027    0.000    0.027    0.000  {method
'getrandbits' of '_random.Random' objects}
               1    0.017    0.017    0.142    0.142
<string>:3(<listcomp>)
               1    0.014    0.014    0.122    0.122
<string>:4(<listcomp>)
     200000    0.009    0.000    0.009    0.000  {method
'bit_length' of 'int' objects}
               1    0.005    0.005    0.005    0.005
<string>:5(<listcomp>)
         1    0.000    0.000    0.000    0.000 {built-in method
builtins.sum}
               1    0.000    0.000    0.270    0.270
<string>:1(<module>)
         1    0.000    0.000    0.271    0.271 {built-in method
builtins.exec}
        3    0.000    0.000    0.000    0.000 socket.py:337(send)
               1    0.000    0.000    0.270    0.270
<string>:1(main_func)
               2    0.000    0.000    0.000    0.000
iostream.py:323(_schedule_flush)
               3    0.000    0.000    0.000    0.000
iostream.py:197(schedule)
         1    0.000    0.000    0.000    0.000 {built-in method
builtins.print}
               2    0.000    0.000    0.000    0.000
iostream.py:310(_is_master_process)
               2    0.000    0.000    0.000    0.000
iostream.py:386(write)
               3    0.000    0.000    0.000    0.000
threading.py:1080(is_alive)
               3    0.000    0.000    0.000    0.000
threading.py:1038(_wait_for_tstate_lock)
         3    0.000    0.000    0.000    0.000 {method 'acquire'
of '_thread.lock' objects}
               3    0.000    0.000    0.000    0.000
iostream.py:93(_event_pipe)
         2    0.000    0.000    0.000    0.000 {built-in method
posix.getpid}
         2    0.000    0.000    0.000    0.000 {built-in method
builtins.isinstance}
         1    0.000    0.000    0.000    0.000 {method 'disable'
of '_lsprof.Profiler' objects}
               3    0.000    0.000    0.000    0.000
threading.py:507(is_set)
        3    0.000    0.000    0.000    0.000 {method 'append' of
'collections.deque' objects}
```

```
%%prun
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)
main_func()
```

```
         1120032 function calls in 0.277 seconds

   Ordered by: internal time

         ncalls      tottime      percall      cumtime      percall
filename:lineno(function)
       200000        0.082        0.000        0.199        0.000
random.py:174(randrange)
       200000        0.081        0.000        0.117        0.000
random.py:224(_randbelow)
       200000        0.037        0.000        0.235        0.000
random.py:218(randint)
     319989       0.027       0.000       0.027       0.000  {method
'getrandbits' of '_random.Random' objects}
               1        0.018        0.018        0.146        0.146
<string>:3(<listcomp>)
               1        0.014        0.014        0.122        0.122
<string>:4(<listcomp>)
     200000       0.009       0.000       0.009       0.000  {method
'bit_length' of 'int' objects}
               1        0.008        0.008        0.008        0.008
<string>:5(<listcomp>)
         1     0.001     0.001     0.001     0.001 {built-in method
builtins.sum}
               1        0.000        0.000        0.277        0.277
<string>:1(<module>)
         1     0.000     0.000     0.277     0.277 {built-in method
builtins.exec}
               1        0.000        0.000        0.276        0.276
<string>:1(main_func)
        3    0.000    0.000    0.000    0.000 socket.py:337(send)
               3        0.000        0.000        0.000        0.000
iostream.py:197(schedule)
               2        0.000        0.000        0.000        0.000
iostream.py:386(write)
               3        0.000        0.000        0.000        0.000
threading.py:1038(_wait_for_tstate_lock)
         1     0.000     0.000     0.000     0.000 {built-in method
builtins.print}
               2        0.000        0.000        0.000        0.000
iostream.py:310(_is_master_process)
               3        0.000        0.000        0.000        0.000
threading.py:1080(is_alive)
         2     0.000     0.000     0.000     0.000 {built-in method
posix.getpid}
               3        0.000        0.000        0.000        0.000
iostream.py:93(_event_pipe)
         3     0.000     0.000     0.000     0.000 {method 'acquire'
of '_thread.lock' objects}
         2     0.000     0.000     0.000     0.000 {built-in method
builtins.isinstance}
               2        0.000        0.000        0.000        0.000
iostream.py:323(_schedule_flush)
        3    0.000    0.000    0.000    0.000 {method 'append' of
'collections.deque' objects}
         1     0.000     0.000     0.000     0.000 {method 'disable'
of '_lsprof.Profiler' objects}
```

```
%%prun -l 10 -s tottime -T prof_res.out
def main_func():
    import random
    arr1 = [random.randint(1,10) for i in range(100000)]
    arr2 = [random.randint(1,10) for i in range(100000)]
    arr3 = [arr1[i]+arr2[i] for i in range(100000)]
    tot = sum(arr3)
    print(tot)
main_func()
```

```
1100623
*** Profile printout saved to text file 'prof_res.out'.
```

```
         1120156 function calls in 0.276 seconds

   Ordered by: internal time
   List reduced from 27 to 10 due to restriction <10>

         ncalls      tottime      percall      cumtime      percall
filename:lineno(function)
      200000        0.085        0.000        0.201        0.000
random.py:174(randrange)
      200000        0.079        0.000        0.116        0.000
random.py:224(_randbelow)
      200000        0.037        0.000        0.238        0.000
random.py:218(randint)
     320113       0.028       0.000       0.028       0.000  {method
'getrandbits' of '_random.Random' objects}
               1       0.017       0.017       0.146       0.146
<string>:3(<listcomp>)
               1       0.014       0.014       0.123       0.123
<string>:4(<listcomp>)
     200000       0.009       0.000       0.009       0.000  {method
'bit_length' of 'int' objects}
               1       0.006       0.006       0.006       0.006
<string>:5(<listcomp>)
         1     0.000     0.000     0.000     0.000 {built-in method
builtins.sum}
               1       0.000       0.000       0.276       0.276
<string>:1(<module>)
```

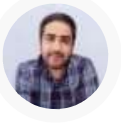
```
!cat prof_res.out
```


```
         1120156 function calls in 0.276 seconds

   Ordered by: internal time
   List reduced from 27 to 10 due to restriction <10>

         ncalls      tottime      percall      cumtime      percall
filename:lineno(function)
      200000        0.085        0.000        0.201        0.000
```

```
       200000         0.037         0.000         0.238         0.000
random.py:218(randint)
     320113       0.028       0.000       0.028       0.000  {method
'getrandbits' of '_random.Random' objects}
                1        0.017        0.017        0.146        0.146
<string>:3(<listcomp>)
                1        0.014        0.014        0.123        0.123
<string>:4(<listcomp>)
     200000       0.009       0.000       0.009       0.000  {method
'bit_length' of 'int' objects}
                1        0.006        0.006        0.006        0.006
<string>:5(<listcomp>)
          1     0.000     0.000     0.000     0.000 {built-in method
builtins.sum}
                1        0.000        0.000        0.276        0.276
<string>:1(<module>)
```

Source:

https://coderzcolumn.com/tutorials/python/list-of-useful-magic-commands-in-jupyter-notebook-lab



Amir Masoud Sefidian

Machine Learning Engineer

Related posts

2023-01-11

Machine Learning for Big Data using PySpark with real-world projects

Read more

2022-11-26

Coursera Deep Learning Specialization Notes

Read more

Comments are closed.