Python workflow

Prof. Dr. Thomas Kopinski

The Python shell

- type python into the command line (Windows: Powershell)

```
Python 3.7.5 (default, Nov  1 2019, 02:16:23)
[Clang 11.0.0 (clang-1100.0.33.8)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

[example: Python shell in a Mac terminal]

- get the current version of Python on your system

```
>>> import sys
>>> print("My version Number: {}".format(sys.version))
My version Number: 3.7.5 (default, Nov  1 2019, 02:16:23)
[Clang 11.0.0 (clang-1100.0.33.8)]
>>>
```

- exit the prompt

```
>>> quit()
(ipy-jupyter-venv3) bash-4.4$
```

Python - a minimal example

- code sample - create a list:

```
>>> list = []
>>> for i in range(10,30,3):
...     list.append(i)
...
>>> print(list)
[10, 13, 16, 19, 22, 25, 28]
>>>
```

- possible to do in a shell
- cumbersome, prone to errors (indenting, which variables exist? etc.)
- better:
  - notebooks
  - write code into modules / files etc.

Python - a minimal example

- code sample in a file *demo.py*:

```python
myList = [1,2,3,4,5]
string = 'this is a long string'

def printLength(x):
    subList = x.split()
    for e in subList:
        print(e)

def listByListComprehension(l):
    #this creates lists with uneven numbers by using list comprehensions
    newList = [x for x in l if l%2 == 1]
    return newList



printLength(string)
listByListComprehension(myList)
```

```
bash-4.4$ python demo.py
this
is
a
long
string
[1, 3, 5]
bash-4.4$
```

- result of running the small program *demo.py*

Python - glossary

- https://docs.python.org/2/glossary.html
- contains many important key aspects of Python, most of which you should be familiar with
- e.g.:
  - bytecode, virtual machine, CPython, scope, nested scope, namespace, PEP, Pythonic, memory management etc.
  - argument, complex number (and others), dictonary, duck-typing, integer division, iterator, lambda, list comprehension etc.
  - and many more!

Data types

Text	`str`
Numeric types	`int, float, complex`
Sequences	`list, tuple, range`
Mappings	`dict`
Sets	`set, frozenset`
Boolean	`bool`
Binary types	`bytes, bytearray, memoryview`

Floats in python

pointfloat	`1.0`
exponentfloat	`1e0`

$$\underbrace{123}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-2}^{\text{exponent}}}$$

significand : mantissa : coefficient

base : radix

exponent : characteristic : scale

Precision in Python

- from the Python doc: https://docs.python.org/2/tutorial/floatingpoint.html
- floating-point numbers are represented by base 2 / binary functions

in decimal	in binary
`0.125 = 1/10 + 2/100 + 5/1000`	`0.001 = 0/2 + 0/4 + 1/8`

- most decimal fractions cannot be precisely represented by binary fractions
- —> decimal floats: approximation by binary floats

in decimal	in binary
`1/3`	`1/10`
`0.3`	`0.00011`
`0.33`	`0.000110011`
`0.333`	`0.0001100110011`
`etc.`	`etc.`

Precision in Python

- when to stop being precise? How many digits?
- outputs in the prompt are typically truncated —> illusion of precison
- 0.1 actually is 0.1000000000000000055511151231257827021181583404541015625
- not useful, not necessary for most applications
- 0.1 + 0.2 = 0.30000000000000004
- or: round(2.675, 2) = 2.67

```
from decimal import Decimal
print(Decimal(2.675))

sum = 0.0
for i in range(10):
    sum += 0.1

print(sum)
```

```
2.6749999999999998223643160599749535322189331054687 5
0.9999999999999999
```

Precedence in Python

- exponentiation (**) has higher precedence than multiplication (*) or division (/)

```
print(8 / 4 ** 2)
```

```
0.5
```

- forced precedence by parentheses:

```
print((8 / 4) ** 2)
```

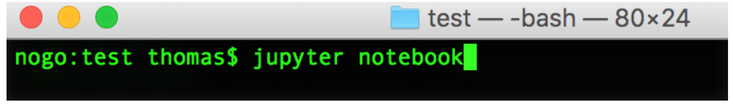
```
4.0
```

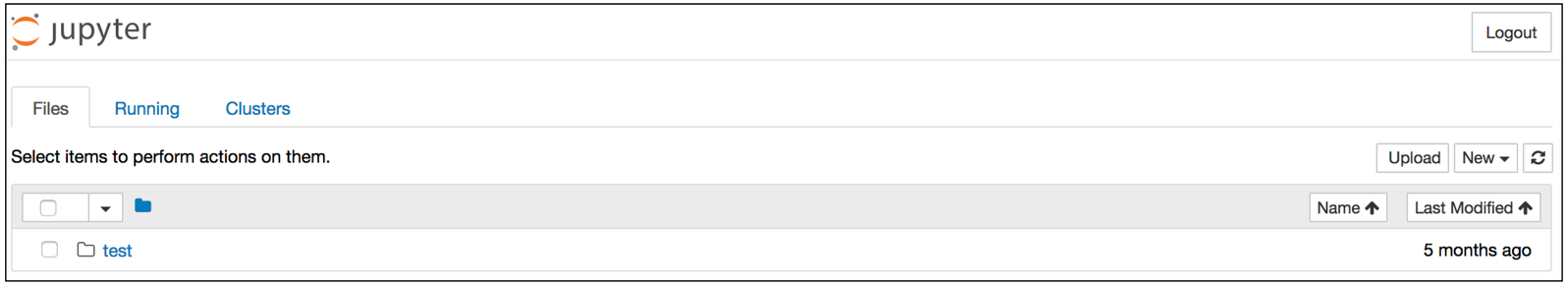
Notebook kernel

- computational engine
- executes the code in a notebook doc
- focus here: ipython kernel
- opening a notebook doc automatically launches the kernel
- executing the (content of a) notebook makes the kernel perform the computation
- makes use of CPU + RAM; RAM is released only after shutting the kernel down

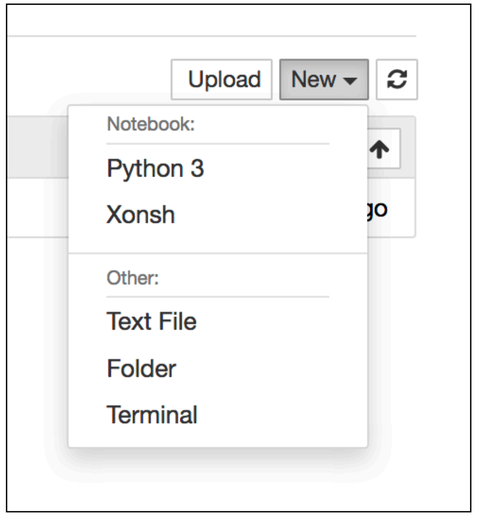
Working with a notebook

- type `jupyter notebook` into the console

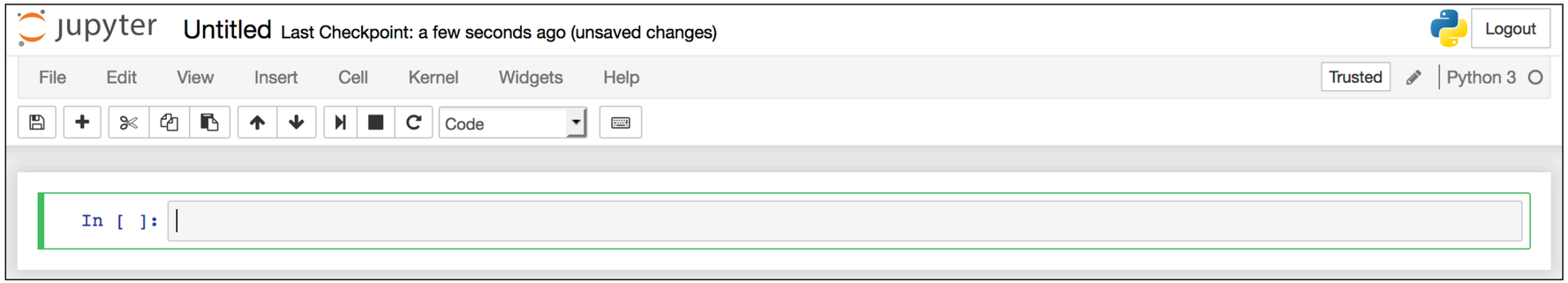




- create a new notebook:



Working with a notebook



- edit mode (green highlighting):

```
In [ ]:
```

- command mode (blue highlighting):

```
In [1]: str = 'this is a string'
```

- command ‚c‘ copies a cell
- command ‚v‘ pastes a copied cell

Working with a notebook

- typically one cell is edited
- run the cell either by pressing 
- or via the menu bar with ‚cell:run‘

```
In [2]: list = [12, 34, 56 ,78]
        list

Out[2]: [12, 34, 56, 78]
```

`Help` -> `Keyboard Shortcuts`

1. Basic navigation: `enter`, `shift-enter`, `up/k`, `down/j`
2. Saving the notebook: `s`
3. Change Cell types: `y`, `m`, `1-6`, `t`
4. Cell creation: `a`, `b`
5. Cell editing: `x`, `c`, `v`, `d`, `z`
6. Kernel operations: `i`, `0` (press twice)

Quick reference guide

```
In [11]: %quickref
```

```
IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %%, and typically take their arguments
without parentheses, quotes or even commas for convenience.  Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10     : time the 'x=10' statement with high precision.
%%timeit x=2**100
x**100           : time 'x**100' with a setup of 'x=2**100'; setup code is not
```

magic functions

```
In [1]: %lsmagic

Out[1]: Available line magics:
        %alias  %alias_magic  %autocall  %automagic  %autosave  %bookmark  %cat  %cd  %clear  %colors  %config  %connect_info
        %cp  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui  %hist  %history  %killbgscripts  %ldir  %less  %lf
        %lk  %ll  %load  %load_ext  %loadpy  %logoff  %logon  %logstart  %logstate  %logstop  %ls  %lsmagic  %lx  %macro  %ma
        gic  %man  %matplotlib  %mkdir  %more  %mv  %notebook  %page  %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2
        %popd  %pprint  %precision  %profile  %prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %qtconsole  %quickref
        %recall  %rehashx  %reload_ext  %rep  %rerun  %reset  %reset_selective  %rm  %rmdir  %run  %save  %sc  %set_env  %sto
        re  %sx  %system  %tb  %time  %timeit  %unalias  %unload_ext  %who  %who_ls  %whos  %xdel  %xmode

        Available cell magics:
        %%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%js  %%latex  %%markdown  %%perl  %%pr
        un  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time  %%timeit  %%writef
        ile

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

```
In [12]: ?%timeit

Docstring:
Time execution of a Python statement or expression

Usage, in line mode:
  %timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o] statement
or in cell mode:
  %%timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o] setup_code
  code
  code...

Time execution of a Python statement or expression using the timeit
module.  This function can be used both as a line and cell magic:

- In line mode you can time a single-line statement (though multiple
  ones can be chained with using semicolons).

- In cell mode, the statement in the first line is used as setup code
  (executed but not timed) and the body of the cell is timed.  The cell
  body has access to any variables created in the setup code.

Options:
-n<N>: execute the given statement <N> times in a loop. If this value
is not given, a fitting value is chosen.
```