JN-Python-quick-start

January 9, 2020

1 Python Language Basics and Jupyter Notebooks

Modified from Wes McKinney's code

[1]:
```
2 + 3
```

[1]: 5

[2]:
```
a =5
print(a)
```

```
5
```

[3]:
```
print("hello world")
```

```
hello world
```

1.1 The Python Interpreter

demo in the shell

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

```
print('Hello world')
```

```
$ python hello_world.py
Hello world
```

2 Starting the Jupyter Notebook from command line

demo in the shell

[4]:
```
import numpy as np
np.random.seed(12345)
```

```
data = {i : np.random.randn() for i in range(7)}
data
```

[4]:
```
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

2.0.1 Tab Completion

[5]:
```
an_apple = 27
```

[ ]:
```
an_apple
```

2.0.2 Introspection

[6]:
```
b = [1, 2, 3]
```

[8]:
```
b?
```

[16]:
```
print(1, b, "no", "hello", sep='|')
```

```
1|[1, 2, 3]|no|hello
```

[12]:
```
?b
```

[13]:
```
?print
```

[19]:
```
def add_numbers(a, b):
   return a + b
```

[20]:
```
?add_numbers
```

[21]:
```
add_numbers(1,5)
```

[21]:
```
6
```

[22]:
```
import os
os.listdir()
```

[22]:
```
['JN-Python-quick-start.ipynb',
 '.ipynb_checkpoints',
```

```
 'helloworld.py',
 'add_number.py']
```

[24]:
```
%run helloworld.py
```

```
hellow world
```

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

```
In [11]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Add two numbers together

Returns
-------
the_sum : type of arguments
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

```
In [12]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

```
In [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

2.0.3 The %run Command

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

```
In [14]: %run ipython_script_test.py
```

```
In [15]: c
Out [15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

```
>>> %load ipython_script_test.py

    def f(x, y, z):
        return (x + y) / z

    a = 5
    b = 6
    c = 7.5

    result = f(a, b, c)
```

Interrupting running code

2.0.4 Executing Code from the Clipboard

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

2.0.5 Terminal Keyboard Shortcuts

2.0.6 About Magic Commands

```
In [20]: a = np.random.randn(100, 100)

In [20]: %timeit np.dot(a, a)
10000 loops, best of 3: 20.9 µs per loop
```

[8]:
```
import numpy as np
a = np.random.randn(1000, 1000)
%timeit np.dot(a, a)
```

```
10.9 ms ± 798 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [21]: %debug?
Docstring:
::

  %debug [--breakpoint FILE:LINE] [statement [statement ...]]

Activate the interactive debugger.

This magic command support two ways of activating debugger.
One is to activate debugger before executing code.  This way, you
can set a break point, to step through the code from the point.
You can use this mode by giving statements to execute and optionally
a breakpoint.

The other one is to activate debugger in post-mortem mode.  You can
activate this mode simply running %debug without any argument.
If an exception has just occurred, this lets you inspect its stack
frames interactively.  Note that this will always work only on the last
traceback that occurred, so you must call this quickly after an
exception that you wish to inspect has fired, because if another one
occurs, it clobbers the previous one.

If you want IPython to automatically do this on every exception, see
the %pdb magic for more details.
```

```
positional arguments:
  statement             Code to run in debugger. You can omit this in cell
                        magic mode.

optional arguments:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                        Set break point at LINE in FILE.
```

```
In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book
```

```
In [23]: foo = %pwd
```

```
In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'
```

2.1 Python Language Basics

2.1.1 Language Semantics

Indentation, not braces

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

```
a = 5; b = 6; c = 7
```

Everything is an object

Comments

```
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #   continue
    results.append(line.replace('foo', 'bar'))
```

```
print("Reached this line")  # Simple status report
```

Function and object method calls

```
result = f(x, y, z)
g()
```

```
obj.some_method(x, y, z)
```

```
result = f(a, b, c, d=5, e='foo')
```

Variables and argument passing

[ ]:
```
a = [1, 2, 3]
```

[ ]:
```
b = a
```

[ ]:
```
a.append(4)
b
```

```
def append_element(some_list, element):
    some_list.append(element)
```

```
In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

Dynamic references, strong types

[9]:
```
a = 5
type(a)
```

[9]:
```
int
```

[10]:
```
a = 'foo'
type(a)
```

[10]:
```
str
```

[13]:
```
'5' + 5
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-13-4dd8efb5fac1> in <module>
----> 1 '5' + 5


TypeError: can only concatenate str (not "int") to str
```

[ ]:
```
a = 4.5
b = 2
```

```
# String formatting, to be visited later
print('a is {0}, b is {1}'.format(type(a), type(b)))
a / b
```

[14]:
```
a = 5
isinstance(a, int)
```

[14]: True

[15]:
```
a = 5; b = 4.5
isinstance(a, (int, float))
isinstance(b, (int, float))
```

[15]: True

Attributes and methods

```
In [1]: a = 'foo'

In [2]: a.<Press Tab>
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

[14]:
```
a = 'foo'
```

[16]:
```
a.upper().lower()
```

[16]: 'foo'

[ ]:
```
getattr(a, 'split')
```

Duck typing If it walks like a duck and it quacks like a duck, then it must be a duck

[17]:
```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```

[18]:
```
isiterable('a string')
```

[18]:
```
True
```

[19]:
```
isiterable([1, 2, 3])
```

[19]:
```
True
```

[20]:
```
isiterable(5)
```

[20]:
```
False
```

if not isinstance(x, list) and isiterable(x): x = list(x)

Imports

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

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

import some_module result = some_module.f(5) pi = some_module.PI

from some_module import f, g, PI result = g(5, PI)

import some_module as sm from some_module import PI as pi, g as gf

r1 = sm.f(pi) r2 = gf(6, pi)

Binary operators and comparisons

[22]:
```
5 - 7
12 + 21.5
5 <= 2
```

[22]:
```
False
```

[ ]:
```
a = [1, 2, 3]
b = a
c = list(a)
a is b
a is not c
```

[ ]:
```
a == c
```

[ ]:
```
a = None
a is None
```

Mutable and immutable objects

[23]:
```
a_list = ['foo', 2, [4, 5]]
a_list[2] = (3, 4)
a_list
```

[23]:
```
['foo', 2, (3, 4)]
```

[24]:
```
a_tuple = (3, 5, (4, 5))
a_tuple[1] = 'four'
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-24-2c9bddc8679c> in <module>
         1 a_tuple = (3, 5, (4, 5))
   ----> 2 a_tuple[1] = 'four'


       TypeError: 'tuple' object does not support item assignment
```

[25]:
```
a_tuple = a_tuple[0], 'four',  a_tuple[2:]
a_tuple
```

[25]:
```
(3, 'four', ((4, 5),))
```

2.1.2 Scalar Types

A scalar is a type that can have a single value such as 1.235, 3.1415926, or 'UTC'.

Python's types are similar to what you'd find in other dynamic languages. This section will move pretty quickly, just showing off the major types and an example or two of their usage.

Numeric types

[ ]:
```
ival = 17239871
ival ** 6
```

[ ]:
```
type(ival)
```

[ ]:
```
fval = 7.243
fval2 = 6.78e-5
```

[ ]:
```
3 / 2
```

[ ]:
```
3 // 2
```

Strings a = 'one way of writing a string' b = "another way"

[ ]:
```
c = """
This is a longer string that
spans multiple lines
"""
```

[ ]:
```
c.count('\n')
```

[ ]:
```
a = 'this is a string'
a[10] = 'f'
b = a.replace('string', 'longer string')
b
```

[ ]:
```
a
```

[ ]:
```
a = 5.6
s = str(a)
print(s)
```

[ ]:
```
s = 'python'
list(s)
s[:3]
```

[38]:
```
s = '12\\34'
print(s)
```

```
12\34
```

[39]:
```
s = r'this\has\no\special\characters'
s
```

[39]:
```
'this\\has\\no\\special\\characters'
```

[ ]:
```
a = 'this is the first half '
b = 'and this is the second half'
a + b
```

[ ]:
```
template = '{0:.2f} {1:s} are worth US${2:d}'
```

[ ]:
```
template.format(4.5560, 'Argentine Pesos', 1)
```

Bytes and Unicode

[ ]:
```
val = "español"
val
```

[ ]:
```
val_utf8 = val.encode('utf-8')
val_utf8
type(val_utf8)
```

[ ]:
```
val_utf8.decode('utf-8')
```

[ ]:
```
val.encode('latin1')
val.encode('utf-16')
val.encode('utf-16le')
```

[ ]:
```
bytes_val = b'this is bytes'
bytes_val
decoded = bytes_val.decode('utf8')
decoded  # this is str (Unicode) now
```

Booleans

[ ]:
```
True and True
False or True
```

Type casting

[26]:
```
s = '3.14159'
fval = float(s)
type(fval)
int(fval)
bool(fval)
bool(0)
```

[26]:
```
False
```

None

[43]:
```
a = None
a is not None
```

[43]:
```
False
```

[42]:
```
b = 5
b is not None
```

[42]:
```
True
```

[48]:
```
def f(x):
    x = x + 1
```

```
    #return x
```

[49]:
```
print(f(10))
```

```
None
```

def add_and_maybe_multiply(a, b, c=None): result = a + b

```
if c is not None:
    result = result * c

return result
```

[ ]:
```
type(None)
```

Dates and times

[55]:
```
from datetime import datetime, date, time
dt = datetime(2011, 10, 29, 20, 30, 21)
print(dt.day)
print(dt.minute)
print(dt.microsecond)
```

```
29
30
0
```

[56]:
```
print(dt.date())
print(dt.time())
```

```
2011-10-29
20:30:21
```

[57]:
```
dt.strftime('%m/%d/%Y %H:%M')
```

[57]:
```
'10/29/2011 20:30'
```

[59]:
```
datetime.strptime('20091031', '%Y%m%d')
```

[59]:
```
datetime.datetime(2009, 10, 31, 0, 0)
```

[65]:
```
print(dt)
dt = dt.replace(minute=0, second=0)
print(dt)
# help(dt.replace())
```

```
2011-10-29 20:00:00
2011-10-29 20:00:00
```

[67]:
```
dt2 = datetime(2011, 11, 15, 22, 30)
delta = dt2 - dt
print(delta)
type(delta)
```

```
17 days, 2:30:00
```

[67]:
```
datetime.timedelta
```

[ ]:
```
dt
dt + delta
```

2.1.3 Control Flow

if, elif, and else if x < 0: print('It's negative')

if x < 0: print('It's negative') elif x == 0: print('Equal to zero') elif 0 < x < 5: print('Positive but smaller than 5') else: print('Positive and larger than or equal to 5')

[ ]:
```
a = 5; b = 7
c = 8; d = 4
if a < b or c > d:
    print('Made it')
```

[68]:
```
4 > 3 > 2 > 1
```

[68]:
```
True
```

[71]:
```
4 > 3 > 5 > 1
```

[71]:
```
False
```

[72]:
```
4 > 3 or 3 < 5 or 5 > 1
```

[72]:
```
True
```

for loops for value in collection: # do something with value

sequence = [1, 2, None, 4, None, 5] total = 0 for value in sequence: if value is None: continue total += value

sequence = [1, 2, 0, 4, 6, 5, 2, 1] total_until_5 = 0 for value in sequence: if value == 5: break total_until_5 += value

[73]:
```
for i in range(4):
    for j in range(4):
        if j > i:
            break
        print((i, j))
```

```
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

for a, b, c in iterator: # do something

while loops x = 256 total = 0 while x > 0: if total > 500: break total += x x = x // 2

pass if x < 0: print('negative!') elif x == 0: # TODO: put something smart here pass else: print('positive!')

range

[74]:
```
range(10)
list(range(10))
```

[74]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[75]:
```
list(range(0, 20, 2))
```

[75]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

[76]:
```
list(range(5, 0, -1))
```

[76]:
```
[5, 4, 3, 2, 1]
```

seq = [1, 2, 3, 4] for i in range(len(seq)): val = seq[i]

sum = 0 for i in range(100000): # % is the modulo operator if i % 3 == 0 or i % 5 == 0: sum += i

Ternary expressions value =

if

[77]:
```
x = 5
'Non-negative' if x >= 0 else 'Negative'
```

[77]:
```
'Non-negative'
```

[25]:
```
x = 5
y = 10
x, y = y, x
```

[26]:
```
print(x, y)
```

```
10 5
```

[ ]:
```

```