CS584: Introduction to Python

G. Agam

January 14, 2016

Contents

1 Introduction

1.1 Overview

- Python 2 vs. 3 (support vs. next generation)
- IDE: PyCharm, Spyder, Canopy.
- Command line interpreters: python, ipython
- Running a script from a terminal:

```
$ python myscript.py
```

1.2 Basic syntax

- No need for semicolon. Use if needed to separate commands on the same line.

```
a=3
b=4; c=5
```

- Blocks start with a colon and are defined by indentation (no begin/end). Normally 4 spaces (no tabs) per indentation level.

```
if a > 10:
    a = 10
```

- Break long lines to prevent wrapping. Use implied line continuation inside parentheses, brackets and braces, or backslash.

```
a , b = (5,
         6)
c , d =  7,\
         8
```

1.3 Hello world

```
#!/usr/bin/python
""" Prints the string "Hello World" and exits . """

my_text = "Hello World"

def main ():
    """ The main function . """
```

```
    print my_text

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

2 Programming in Python

2.1 Printing

- In an interactive python session enter the name of a variable to see its content.
- From a script use the `print` command.

```
print ("Hello World") # required in python 3
print "Hello World"

a, b, c = 1, 2, 3
print "a = " , a , " , b = " , b , " , c = " , c

print "item %05d has value %g" % (23 , 3.14**2)  # operator % works like sprintf
print str(1.0/7.0)                               # '0.142857142857'
```

2.2 Basic arithmetic

```
2+2      # = 4    integer addition
2.0+2.0  # = 4.0  floating point addition
5.0/2    # = 2.5  floating point division
5/2      # = 2    integer division
5.0//2.0 # = 2.0  force integer division
5 % 2    # = 1    modulo division
3 ** 2   # = 9    exponentiation
_ + 2    # = 11   add to the last returned value
```

2.3 Built-in types

- `int` - Integer type with limited precision (in Python 2) and unlimited precision (in Python 3).
- `long` - Integer type with unlimited precision (in Python 2 only).
- `float` - Floating point type (usually double precision).
- `complex` - Complex number with real and imaginary float components.
- `bool` - Boolean with values False or True.
- `None` - A null object.
- `str` - A string of characters.

- `tuple` - A fixed sequence of objects.
- `list` - A mutable sequence of objects.
- `dict` - A dictionary mapping one set of objects to another.
- …

```
type(1)         # int
type(1.0)       # float
type(True)      # bool
type('hello')   # str
```

2.4 Type conversion

```
float (7)           # 7.0
int (3.14159)       # 3
str (28)            # '28'
float ('5.5')       # 5.5
complex (4.5)       # (4.5 + 0 j)
bool (0)            # False
float ( True )      # 1.0
```

2.5 Variables

- No declaration. Created during assignment.

```
my_var = 10                 # create variable and assign value
x , y , z = 10 , 20 , 30    # multiple assignment , uses tuples
x += 5
y *= 2
my_var = 'str'              # reassign
who                         # show variables in workspace (ipython)
whos                        # show variables in workspace with type (ipython)
```

2.6 Bolean operators

- The boolean constants are: 'True' and 'False'
- A zero value (0 or 0.0 or "") is False, and a non-zero value is True
- Comparisons (return a boolean value): '>', '<', '>=', '<=', '==', '!='

```
x or y                  # if (x is True)  return x
x and y                 # if (x is false) return x
not x                   # if (x is True)  return False
```

2.7 Tuples

```
t = 1 , 'hello' , (3.14 , None ) ,   2.5
t [0]            # access item by index
t [1:3]          # access a slice of the objects ( start : end )
t [0] = 2        # ERROR: item assignment not allowed
len (t)          # the length of the tuple (4)
2.5 in t         # test for inclusion (True)
t + (5 ,)        # concatenate tuples (add 5 at end)
(1 , 2) * 3      # concatenate n shallow copies  (1 , 2 , 1 , 2 , 1 , 2)
```

2.8 Strings

```
s = "Hello World"
s = 'Hello World'
s.upper()                        # to upper case ('HELLO WORLD')

s[0:2]                           # slicing substring ('He')
s[:5]                            # substring ('Hello')
s[6:]                            # substring ('World')
s[0:5:2]                         # substring with stride ('Hlo')
s[4::-1]                         # substring with inversion ('olleH')
s[4::-2]                         # substring with inversion and stride ('olH')
s[::-1]                          # complete inversion ('dlroW olleH')

s[0]='J'                         # ERROR strings are immutable
s = 'J'+s[1:]                    # 'Jello World'

s[0:2] + '+-' * 3                # concatenation ('He+-+-+-')
s.split()                        # string splitting (['Hello', 'World'])
'-'.join(('a' , 'b' , 'c' ))     # join strings with a separator ('a-b-c')
s.replace("Hello", "Goodbye")    # substring replacement ('Goodbye World')
```

2.9 Lists

```
v = [1 , 'hello' , (3.14 , None ) , 2.5]
v [0] = 2             # item assignment
v.append (0.5)        # append items to the end
v=v + [33,]           # append items at the end
del v[2]              # delete an item
v.sort ()             # sort items in place
```

2.10 Dictionaries

- Dictionaries are mutable mapping of one object to another. Contain `Key:value` pairs that are comma separated. Enclosed in curly braces.

```
d = { 'a' : 1 , 'b' : 2 , 'c' : 3}
d ['a']               # access a value by key (1)
d ['d'] = 4           # change a value or insert a new key : value
d.keys()              # ['a ' ,'c ' ,'b' ,'d']
d.values()            # [1 , 3 , 2 , 4]
d.items()             # [( 'a' , 1) , ( 'c' , 3) , ( 'b' , 2) , ( 'd' , 4)]
```

2.11 Conditionals

```
if a > 10:
    a = 10

if b > 10: b = 10

if a > 10:
     print 'a greater than 10'
elif a < 5:
     print 'a less than 5'
else:
     print 'a is' , a
```

2.12 Loops

```
x = 1
while x < 10:
    x = x * 2
    if x==4 : continue
    if x >8 : break
    print(x)
# 2 8


X = [5 , 4 , 3 , 2 , 1]
for v in X :
    print v


for v in range(0,5):            # [0, 1, 2, 3, 4]
    print v

for v in range(5,0,-1):         # [5, 4, 3, 2, 1]
    print v

for v in arange(0.2,0.5,0.1):   # array([ 0.2,  0.3,  0.4])
    print v
```

2.13 Functions

Example

```
def sum_diff (x , y ):
    """ Sum and diff of two values .
    x is the first value
    y is the second value
    """
    return x + y , x - y


s , d = sum_diff (7 , 4)
both  = sum_diff (7 , 4)
print "s = " , s , " , d = " , d , " , both = " , both
```

Arguments

```
def my_func (req , opt1 =3.14 ,
             opt2 = 'default string'):
    print opt2
    return req * opt1

my_func(1)
my_func(1, 3.14, 'hi')
my_func(1, opt2='hi')
my_func(1, opt2='hi', opt1=10)
```

2.14 Importing packages and modules

Importing packages/modules

- Examples:

```
import math
math.sqrt(4)
math.sin(math.pi)
```

- Renaming during import:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

np.dot(a, a)
```

3 Scientific Computing in Python

3.1 Overview

- NumPy:
  - multidimensional arrays with homogeneous data types
  - specific numeric data types (e.g. int8, uint32, float64)
  - array manipulation and generation
  - element-wise math operations (e.g. add, multiply, max, sin)
  - basic matrix math operations
- SciPy:
  - advanced math
- Matplotlib: plotting capability
- PyLab: a meta-package that includes NumPy, SciPy, and Matplotlib.

3.2 Using PyLab

- Import conventions:

```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
```

- Options for importing functions :

```
from pylab import *              # Import everything .
svd (eye (3))

from numpy import eye , array    # Import specific into a global namespace
from numpy.linalg import svd
svd (eye (3))

import numpy as np               # Import everything into a namespace
np.linalg.svd(np.eye(3))
```

3.3 Using NumPy

NumPy arrays (ndarray)

```
from pylab import *

A = array ([1. , 2. , 3.])        # create a one dimensional array from a list

B = array ([[1 , 2 , 3] ,
            [4 , 5 , 6]])         # create a two dimensional array from a list of lists

C = array ([ B , B , B , B ])     # create a 3 D array from a list of 2 D arrays

A.ndim                            # get the number of dimensions:       1 / 2 / 3
A.shape                           # get the size of each dimension:     (3,) / (2,3) /
    (4,2,3)
A.dtype                           # get the data type of each element:  float64 / int64 /
    int64
type(A)                           # get the object type: numpy.ndarray
```

- NumPy arrays can be initialized using lists or tuples:

```
A = array( [[1. , 2. , 3.], [5, 6, 7]] )
B = array( ((1. , 2. , 3.), (5, 6, 7)) )
```

- Operations on NumPy arrays are elementwise.

```
M = array ([[1 , 2] ,[2 , 1]])
print M
print M*M                         # elementwise multiplication
print dot(M,M)                    # matrix (or tensor) product
```

NumPy Numerical Data Types

- Data types:
- `bool` - Boolean (True or False) stored in a byte
- `int` - Platform integer (usually int32 or int64)
- `int8, int16, int32, int64` - 8, 16, 32 and 64 bit signed integers
- `uint8, uint16, uint32, uint64` - 8, 16, 32 and 64 bit unsigned integers
- `float` - Shorthand for float64
- `float16, float32, float64` - half, single, and double precision floats

- `complex` - Shorthand for complex128
- `complex64, complex128` - complex numbers with single and double precision

Type conversion

```
import numpy as np

A = np.array([1, 2, 3])                          # construct as int
B = np.array ([1 , 2 , 3], dtype = np.float32)  # construct as float32
C = A.astype('complex')                          # convert the type of an existing array

A.dtype                                          #  int64 / float32 / complex128
```

NumPy Builder functions

```
from pylab import *

M = zeros ((3 ,4))
M = ones ((3 ,4))
M = eye (3)
M = diag ([1 , 2 , 3])
V = diag ( M )
M = random((3,4))                    # uniform distribution in [0,1]

A = array([[1, 2],[3,4]])
M = tile(A,(2,2))                    # replicate a block matrix

A = np.arange(6)
A = np.arange(6).reshape(2, 3)

A = r_[1:5]                          # array([0, 1, 2, 3, 4])
A = r_[1:3:0.5]                      # array([ 1., 1.5, 2., 2.5]) - step=0.5
A = r_[5:10:5j]                      # array([ 5., 6.25, 7.5, 8.75, 10.]) - samples=5
```

Slicing NumPy arrays

```
import numpy as np

A = np.array([[1,2,3],[4,5,6],[7,8,9]])

print A[0,0]                                    # row 0 column 0
print A[1,:]                                    # second row
print A[:,1]                                    # second column [2 5 8]

print A[0:2,:]                                  # first two rows
print A[:,:2]                                   # first two columns

print A[0:3:2,:]                                # every other row for rows 0-2
print A[::2,:]                                  # every other row for rows 0-2
```

```
print A[::-1,:]                                    # rows in reverse order

print A[-1,:]                                      # lat row
print A[:,-1]                                      # lat column
```

Accessing sub-blocks

- Sub-block:

```
A = zeros((4,4))
A[0:2,0:2] = ones((2,2))
# array([[ 1.,  1.,  0.,  0.],
#        [ 1.,  1.,  0.,  0.],
#        [ 0.,  0.,  0.,  0.],
#        [ 0.,  0.,  0.,  0.]])
```

- Sub-block with skipping:

```
A = np.arange(20).reshape(4, 5)
#array([[ 0,  1,  2,  3,  4],
#       [ 5,  6,  7,  8,  9],
#       [10, 11, 12, 13, 14],
#       [15, 16, 17, 18, 19]])

A[ix_([0,2,3],[0,2])]  # extract rows 0,2,3 and columns 0, 2
#array([[ 0,  2],
#       [10, 12],
#       [15, 17]])
```

Stacking NumPy arrays

```
A =  r_ [ eye (2) , 2* ones ((2 ,2)) ]  # stack as rows
B =  c_ [ eye (2) , 2* ones ((2 ,2)) ]  # stack as columns
```

Matrix operations with NumPy arrays

- Transpose:

```
from pylab import *

x=arange(3).reshape(3,1)
xt=arange(3).reshape(1,3)
x.shape       # (3,1)
xt.shape      # (1,3)

x1d=array([1,2,3])
shape(x1d)    # (3,)
shape(x1d.T)  # (3,) There is no meaning to transposing a 1D array
```

- Products:

```
from pylab import *

# matrix product
y=dot(A,x)
y.shape       # (3,1)
y=dot(A,xt)                          # ERROR: mismatched dimensions

# inner (dot) product
s=dot(xt,x)
s.shape       # (1, 1)

# outer product
B=dot(x,xt)
B.shape       # (3, 3)

# cross product
xt1=xt+[0,0,1]
z=cross(xt,xt1)
z.shape       # (1, 3)
```

- Addition:

```
from pylab import *

# Addition with broadcasting (careful)
x+x.T               # x=[0, 1, 2]
#array([[0, 1, 2],  # [0, 1, 2] + 0
#       [1, 2, 3],  # [0, 1, 2] + 1
#       [2, 3, 4]]) # [0, 1, 2] + 2

# sum along rows/columns
A = array([[0, 1, 2],
           [3, 4, 5],
           [6, 7, 8]])

A.sum(0)    # sum along rows:    array([ 9, 12, 15])
A.sum(1)    # sum along columns: array([ 3, 12, 21])
```

- Invert/ solve:

```
from pylab import *

det(A)                              # determinant
inv(A)                              # inverse
x = solve(A,b)                      # solve Ax=b
```

- Factor:

```
from pylab import *

e, V = eig (S)                    # eigenvalues
U,d,Vt = svd(A)                   # SVD
Q, R = qr(A)                      # QR factorization
```

NumPy Matrix

- `numpy.matrix` is a subclass of `numpy.array`. The default operations there are matrix operations and not elementwise operations. It can only handle 2D arrays.

```
from pylab import *

M1 = matrix('1 2; 2 1')   # initialize matrix with string (not available for arrays)
M2 = matrix(A)            # Initialize from array with copy
M3 = mat(A)               # Initialize from array without copy
```

- Convert between ndarray and matrices:

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
M = np.asmatrix(A)          # convert to matrix without copy

N = np.matrix('1 2; 3 4')
B = np.asarray(N)           # convert to array without copy
```

- The default multiplication is normal matrix multiplication:

```
import numpy as np

A = np.matrix('1 1; 1 1')
print A*A                       # matrix multiplication
print A**3                      # A*A*A
```

- To override common array constructors (`zeros()`, `ones()`, `eye()`, `rand()`, `randn()`, `repmat()` etc.) with matrix versions, use `matlib`

```
from numpy.matlib import *
```

3.4 Using MatplotLib

Plot with blocking

```
import numpy as np
import matplotlib.pyplot as plt

x=arange(0,2*np.pi,0.1)
y=np.sin(x)
plt.plot(x, y, lw=2)       # linewidth=2

xlabel('x')
ylabel ('y')
title ('my title')
grid (True )

plt.show()                 # plot is not displayed until here
                           # you have to close the plot to continue
```

Plot without blocking

- Use 'ion()' to enter interactive plotting. The plots are updated after each command, and there is no need for 'show()'. Use 'ioff()' to disable this mode.

Subplots

```
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0, 10, 0.01)

plt.clf()                          # clear figure
plt.close('all')

ax1 = plt.subplot(211)             # 2x1 plot array - plot 1
ax1.clear()
ax1.plot(t, np.sin(2*np.pi*t))

ax2 = plt.subplot(212, sharex=ax1)  # 2x1 plot array - plot 2
ax2.plot(t, np.sin(4*np.pi*t))

plt.draw()
```