Brief Introduction to Python

Jean-Baptiste Bordes, Fragkiskos Malliaros, Nikolaos Tziortziotis and Michalis Vazirgiannis

INF582: Data Science – Learning from Data

Friday, 4 December 2015

Outline

- **Python basics**
- Numpy basics

Overview of Python

- Python is an interpreted language, meaning there is no compilation or linking
- Python can be used in two different modes
  - **Interactive mode** makes it easy to experiment with the language
  - **Standard mode** is for running executable scripts and programs
- Typically, Python programs are much shorter than equivalent C, C++, or Java programs
  - High-level language and data types allow expressing complex operations concisely
  - Grouping of statements is done by **indentation** (e.g., tabs) instead of using brackets
  - No variable declarations

Modules and Built-in Numeric Types

- Python modules are **libraries** of code
  - They are imported using the `import` function, which runs the file

```
>>> from math import pi, sqrt

>>> import math
>>> math.pi
>>> math.e
```

- Python provides integers, floating-point numbers, complex numbers, etc.
  - **int**
  - **float**
  - **long** (long integers have unlimited precision)
  - **complex** (they have a real and imaginary part, which are each a floating point number)

Operators

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	(floored) quotient of *x* and *y*
`x % y`	remainder of x / y
`-x`	*x* negated
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`long(x)`	*x* converted to long integer
`float(x)`	*x* converted to floating point
`complex(re,im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`pow(x, y), x ** y`	*x* to the power *y*

Comparisons

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Examples of Operations

```
>>> 125 + 25
>>> 125 * 25
>>> 125 ** 25
>>> 4 1/3
>>> 1/ float (3)
>>> 1/3.0

>>> import math
>>> math.sqrt (math .pi)
>>> math.sin (_)
>>> 1 + _
```

- In the interactive mode, the _ operator contains the result of the last operation, which is very handy for subsequent operations

Python Statements (1/3)

The Python if statement

```
if test:
     block of code
elif test:
     block of code
else:
     block of code
```

Example: Computation of the absolute value

```
x = 4
y = 5

if x > y:
     absval = x - y
elif y > x:
     absval = y - x
else:
     absval = 0

Print "The absolute value is ", absval
```

Python Statements (2/3)

The Python for statement

```
for target in sequence:
     block of code
```

Examples

```
names = ['Peter ', 'John ', 'Mary ', 'Helen ', 'Tom ',
'Nicholas ']

for name in names:
     print name

for x in [0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10]:
     print x

for x in range (11):
     print x

for x in range (10 ,21) : print x
```

The range function creates a list of integers [start, stop) with the following syntax

```
range([ start ,] stop [, step ])
```

Python Statements (3/3)

The Python while statement

```
while expression:
     block of code
```

Example

```
temperature = 60

while (temperature > 40):
     print('The water is hot enough')
     temperature = temperature - 1

print ('Water's temperature is now OK')
```

How to execute python programs

Save a block of code in a file with extension "py": test1.py

Example

```
#test1.py
temperature = 60
while (temperature > 40):
     print('The water is hot enough')
     temperature = temperature - 1
print ('Water's temperature is now OK')
```

Execute the program from the OS

```
$python test1.py
```

Execute the program from within the python environment

```
>>> execfile(test1.py)
```

Lists (1/2)

- A list is a container that holds a number of other objects, in a given order
- It can contain more than one basic data types which are surrounded by square brackets

```
>>> mylist = [0,10,'data','mining']
```

- Accessing particular elements of a list simply requires giving it an index. Python indices start at 0

```
>>> print mylist[0]
0
>>> print mylist[3]
mining
```

- You can also index from the end using a minus sign

```
>>> print mylist[-2]
data
```

Lists (2/2)

- Sections of a list can be easily accessed using the *slice* operator. It can take three operators: [start:stop:step]

```
>>> print mylist[1:3]
[10,'data']
>>> print mylist[1:]
[10,'data','mining']
>>> print mylist[:2]
[0 10]
>>> print mylist[1:4:2]
[10,'mining']
```

- Some functions that are available to operate on lists are:
  - append(x) Adds x to the end of the list
  - count(x) Counts how many times x appears in the list
  - pop(i) Removes the item at index I
  - remove(x) Deletes the first element that matches x
  - reverse(x) Reverses the order of the list

Dictionaries

- A dictionary is another container that can store any number of objects, including other container types
- Dictionaries consist of pairs of keys and their corresponding values which are surrounded by curly brackets

```
>>> months = {'Jan':31,'Feb':28,'Mar':31}
```

- The elements of the dictionary can be accessed using their key

```
>>> print months['Jan']
31
```

- The functions *keys()* and *values()* return a list of all the keys and values of the dictionary respectively

```
>>> print months.keys()
['Jan','Mar','Feb']
```

Outline

- Python basics
- **NumPy basics**

NumPy

- **NumPy** is a Python package that adds support for large multi-dimensional arrays and matrices, along with a large library of high level mathematical functions to operate on these arrays
- To import the NumPy library we use

```
>>> from numpy import *
```

- The basic data structure of NumPy is the array
  - It consists of one or more dimensions of numbers or chars
  - Unlike lists, the elements of the array all have the same type

Simple Array Creation

Arrays are made using a function call, and the values are passed in as a list or set of lists for higher dimensions

```
>>> array1 = array([4,3,2])
```

Create an one-dimensional array

```
>>> print array1
[4 3 2]
```

Print array

```
>>> array2 = array([[3,2,4],[3,3,2],[4,5,2]])
```

Create a two-dimensional array

```
>>> print array2
[[3 2 4]
 [3 3 2]
 [4 5 2]]
```

Print array

Arrays can be accessed in the same way as lists

```
>>> print array1[1]
3
>>> print array2[2,0]
4
```

Print indices

Array Creation Functions (1/2)

```
>>> print zeros((2,2))
[[ 0.  0.]
 [ 0.  0.]]
```

Produces an array containing all zeros

```
>>> print ones((3,4))
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

Similar to zeros(), except that all elements of the matrix are ones

```
>>> print eye(3)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Produces the identity matrix, the matrix that is zero everywhere except down the leading diagonal, where it is one

Array Creation Functions (2/2)

```
>>> print arange(5)
[0 1 2 3 4]
>>> print arange(3,7,2)
[3 5]
```

Produces an array containing the specified values, acting as an array version of range()

```
>>> print linspace(3,7,3)
[ 3.  5.  7.]
```

Produces a matrix with linearly spaced elements. The user specifies the number of elements, not the spacing

Getting Information about Arrays

```
>>> a = array([[0,1],[2,3],[4,5]])
>>> print a
[[0 1]
 [2 3]
 [4 5]]
```

The array that will be used in our examples

```
>>> print ndim(a)
2
```

Returns the number of dimensions

```
>>> print size(a)
6
```

Returns the number of elements

```
>>> print shape(a)
(3 2)
```

Returns the size of the array in each dimension. You can access the first element of the result using shape(a)[0]

Changing the Shape of an Array

```
>>> print reshape(a,(2,3))
[[0 1 2]
 [3 4 5]]
```

Reshapes the array as specified

```
>>> print ravel(a)
[0 1 2 3 4 5]
```

Makes the array one-dimensional

```
>>> print transpose(a)
[[0 2 4]
 [1 3 5]]
```

Computes the transpose of the matrix

```
>>> print a[::-1]
[[4 5]
 [2 3]
 [0 1]]
```

Reverses the elements of each dimension

Operations on Arrays (1/2)

```
>>> b = arange(3,9).reshape(3,2)
>>> print b
[[3 4]
 [5 6]
 [7 8]]
```

The second array that will be used in the examples

```
>>> print a+b
[[ 3  5]
 [ 7  9]
 [11 13]]
```

Matrix addition

```
>>> print a*b
[[ 0  4]
 [10 18]
 [28 40]]
```

Element-wise multiplication

Operations on Arrays (2/2)

```
>>> c = transpose(b)
>>> print c
[[3 5 7]
 [4 6 8]]
```

The third array that will be used in the examples

```
>>> print dot(a,c)
[[ 4  6  8]
 [18 28 38]
 [32 50 68]]
```

Matrix multiplication

```
>>> print pow(a,2)
[[ 0  1]
 [ 4  9]
 [16 25]]
```

Computes exponentials of elements of matrix

```
>>> print pow(2,a)
[[ 1  2]
 [ 4  8]
 [16 32]]
```

Computes number raised to matrix elements

The *min()*, *max()* and *sum()* Functions

```
>>> print a.min()
0
```

Returns the smallest element of a

```
>>> print a.max()
5
```

Returns the largest element of a

```
>>> print a.sum()
15
>>> print a.sum(axis=0)
[6  9]
>>> print a.sum(axis=1)
[1 5 9]
```

Returns the sum of elements. Often used to sum the rows or columns using the axis option

The *where()* Command

```
>>> x = where(a>2)
>>> print x
(array([1, 2, 2]), array([1, 0, 1]))
```

Returns the indices where the logical expression is true in the variable x

```
>>> print where(a>2,0,1)
[[1 1]
 [1 0]
 [0 0]]
```

Returns a matrix with the same size as a that contains 0 in those places the expression was true and 1 everywhere else

- Two or more conditions can be chained together using bitwise logical operators like | and &

Linear Algebra

- NumPy has a reasonable linear algebra package that performs standard linear algebra functions
- Some frequently used functions are:
  - **linalg.inv(a)** Computes the inverse of square array **a**
  - **linalg.pinv(a)** Computes the pseudo-inverse, which is defined even if **a** is not square
  - **linalg.det(a)** Computes the determinant of **a**
  - **linalg.eig(a)** Computes the eigenvalues and eigenvectors of **a**
  - **linalg.svd(a)** Computes the SVD decomposition of **a**