Python and Bioinformatics

Pierre Parutto

November 23, 2016

Contents

Chapter 1

Matrices

1.1 Matrix Representation

A matrix is an n dimensional ordered structure that can store only data of one type. When not specified otherwise, we call matrix a 2 dimensional matrix.

1.1.1 Basic Representation

Matrices can be constructed in Python as nested lists. For example the 2×2 matrix below:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

can be constructed in Python as:

```
>>> m = [[1,2], [3,4]]
```

`m` is thus a list of size 2 (the number of lines) where each element is also a list of size 2 (the number of columns). With this construction we can access to each elements with the syntax:

- $m_{1,1}$:

```
>>> m[0][0]
1
```

- $m_{1,2}$:

```
>>> m[0][1]
2
```

- ...

Remark

We could also have represented the columns first, thus producing the list:

```
>>> m = [[1,3], [2,4]]
```

Then the first index corresponds to the column and the second to the line:

```
>>> m[1,0]
2
>>> m[0,1]
3
```

The most common representation used in computer-science is the line-based representation (the first one presented). In the following we will only use this representation.

1.1.2 Using The Library `numpy`

`numpy` is a library that was specifically created to create, manipulate and store matrices. There are two major advantages in using `numpy` matrices over the Python-based matrices:

1. The efficiency: `numpy` functions are written mostly in `C++` and use specific libraries (called `blas` and `lapack`) to efficiently handle matrices.
2. The functions: `numpy` already define a lot of useful functions on matrices thus you don't have to re-code them yourself.

Warning

If you want to use `numpy` functions, you will have to import the names from the `numpy` module.

`Numpy` provides the type `array` to represent matrices.

Warning

In the following, and in the whole class, we only consider `numpy` matrices.

1.2 Creating Matrices

There are four functions that are mostly used to create matrices:

- `zeros(shape: tuple, dtype: type) -> array`, fills the new matrix with 0 values.
- `ones(shape: tuple, dtype: type) -> array`, fills the new matrix with 1 values.

- `empty(shape: tuple) -> array, dtype: type`, do not fill the matrix with any value, the matrix contains whatever value was in the new memory spot.
- `array(l: list, dtype: type) -> array`: convert a Python list representing an array (as presented at the beginning of the class) into a `numpy` matrix.

The first three functions takes two arguments:

1. `shape`, a tuple as argument corresponding to the size of the matrix. The new matrix will have `len(shape)` dimensions, each dimension `d` with the size `shape[d]`.
2. `dtype`, specify the type of the values in the matrix. Three main types you main want to use are: `bool`, `float` or `int`.

Remark

The `dtype` value is not mandatory in the function, you can specify it or not. By default, for the `zeros`, `ones` and `empty` function, the type will be `float` and for the `array` function, it will be the type of the values provided in the list.

Remark

If you use the `array` function and specify a `dtype` different from the type of the list, the values will be converted into the specified type:

- `bool`, the values: `None`, `0.0`, `0` will be converted to `False` while all other value will be considered as `True`.
- `int` and you provide float values, the values will be rounded to the **smallest** integer.

1.2.1 Examples

For the three first functions:

```
>>> from numpy import zeros, ones, empty
>>> zeros((2,2))
array([[ 0., 0.],
       [ 0., 0.]])
>>> zeros((2,2), dtype=int)
array([[ 0, 0],
       [ 0, 0]])
>>> ones((3,2), dtype=bool)
array([[ True, True],
```

```
       [ True, True],
       [ True, True]], dtype=bool)
>>> empty((2,3)) #this will change with each new call
array([[ 6.94533685e-310, 1.99775009e-316, 1.98811700e-316],
       [ -1.12981890e-282, 2.02300673e-316, 2.02300752e-316]])
```

For the `array` function:

```
>>> from numpy import array
>>> array([[1,2], [3,4]])
array([[1, 2],
       [3, 4]])
>>> array([[0,1], [2,3]], dtype=bool)
array([[False, True],
       [ True, True]], dtype=bool)
>>> array([[0.9, 7.4, 5.3], [0.0, -0.1, 0.0005]], dtype=int)
array([[0, 7, 5],
       [0, 0, 0]])
```

1.3 Matrix Characteristics

If `m` is a `numpy` matrix (of type array), the following values are associated to `m`:

Value	Type	Definition
`m.size`	`int`	Number of elements
`m.ndim`	`int`	Number of dimensions
`m.shape`	`tuple`	Number of elements per dimension
`m.dytpe`	`str`	Type of the matrix elements

Remark

`m.shape` is a tuple of size `m.ndim` where each element is the size of the corresponding dimension.

Remark

For a `numpy` matrix, We always have the following relation:

$$m.size = \prod_{i=0}^{i<\text{m.ndim}} \text{m.shape[i]}$$

1.3.1 Example

```
>>> from numpy import array
>>> m = array([[1,3,4,6], [-1, 0, 5, -2]])
>>> m.size
8
>>> m.ndim
2
>>> m.shape
(2,4)
>>> m.dtype
'int64'
```

1.4 Accessing Matrix Elements

For an N dimensional matrix:

```
M[val_dim1, ..., val_dimN]
```

For a 2 dimensional matrix, by definition the first dimension represents the lines of the matrix and the second dimension the columns. Thus the syntax:

```
M[i,j]
```

accesses to the element located on the ith line and jth column.

You can also use slicing to access to a full line or column:

- `m[i,:]`: accesses to the ith line;
- `m[:,j]`: accesses to the jth column.

Warning

Be careful, in math, matrix indices starts at 1 while in `numpy` they start at 0:

- $m_{1,1} \rightarrow$ `m[0,0]`;
- $m_{2,1} \rightarrow$ `m[1,0]`;
- ...

1.5 Arithmetic Operations On Matrix

1.5.1 Integer And Matrix

Intuitively, if you have and integer `n` and a matrix `A` of size $m \times n$, the following operations are defined:

- `n + A` produces the matrix `C` of size $m \times n$ such that:

$$c_{i,j} = n + a_{i,j}$$

- `n - A` produces the matrix `C` of size $m \times n$ such that:

$$c_{i,j} = n - a_{i,j}$$

- `n * A` produces the matrix `C` of size $m \times n$ such that:

$$c_{i,j} = n * a_{i,j}$$

- `n / A` produces the matrix `C` of size $m \times n$ such that:

$$c_{i,j} = n/a_{i,j}$$

- `A ** n` produces the matrix `C` of size $m \times n$ such that:

$$c_{i,j} = (a_{i,j})^n$$

1.5.2 Matrix And Matrix

All classical operations are elementwise, with `A`, `B` two matrices of size $m \times n$:

- `A + B` produces a matrix `C` of size $m \times n$, such that:

$$c_{i,j} = a_{i,j} + b_{i,j}$$

- `A * B` produces a matrix `C` of size $m \times n$, such that:

$$c_{i,j} = a_{i,j} * b_{i,j}$$

- `A / B` produces a matrix `C` of size $m \times n$, such that:

$$c_{i,j} = a_{i,j}/b_{i,j}$$

- `A ** B` produces a matrix `C` of size $m \times n$, such that:

$$c_{i,j} = (a_{i,j})^{b_{i,j}}$$

Warning

With these operators, the two matrices must have the same size.

1.5.2.1 Matrix Multiplication

An important operation on matrices is the multiplication. You can do it using the function `dot`:

```
>>> from numpy import dot, array
>>> A = array([[1,2], [3,4]])
>>> B = array([[5,6], [7,8]])
>>> dot(A,B)
array([[19,22], [43,50]])
```

Warning

Be careful with the dimensions of the matrices. If you want to manipulate both 2D matrices and vectors (a 2D matrix with only 1 dimension), the vector must also be a 2D matrix, and not a 1D one.

For example:

```
>>> from numpy import array, dot
>>> A = array([[1,2,3], [4,5,6], [7,8,9]]) #this is a 3×3 matrix
>>> v = array([[1,2,3]]) #this is a row vector 1×3
>>> dot(v,A) #1×3 * 3×3 is OK, creates a 1×3 array
array([[30, 36, 42]])
>>> dot(v,A) #3×3 * 1×3 is NOT OK
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
>>> vv = array([[1], [2], [3]]) #this is a column vector 3×1
>>> dot(vv, A) #3×1 * 3×3 is NOT OK
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
>>> dot(A, vv) #3×3 * 3×1 is OK, creates a 3x1 array
array([[14],
       [32],
       [50]])
```

1.6 Other Functions On Matrices

1.6.1 Transposition

Transposing a matrix consists in transforming the lines into columns and *vice versa*. It can be done easily for a matrix `m` with the syntax `m.T`.

1.6.1.1 Example

```
>>> from numpy import array
>>> m = array([[1,2],[3,4],[5,6]])
>>> m
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> m.size
(3,2)
>>> n = m.T
>>> n
array([[1, 3, 5],
       [2, 4, 6]])
>>> n.shape
(2,3)
```

1.6.2 Reshaping A Matrix

You can reshape matrices, changing the number of dimensions and the number of elements per dimension. The only constraint is that the size of the matrix remains the same. `Numpy` provides the function:

```
m.reshape(s: tuple) -> array
```

Where `m` is a variable of type array.

1.6.2.1 Example

```
>>> from numpy import array
>>> A = array([1,2,3,4])
>>> A.reshape((2,2))
array([[1,2],[3,4]])
```

1.6.3 Linear Index To 2D Index

If you are given a linear (1D value, corresponding to a flattened version of the array) index from a function from `numpy`, you can transform it to a 2D index using the following function:

```
unravel_index(idx: int, shape: tuple) -> tuple
```

Takes the 1D index and transforms it to an ND index using the shapes provided in `shape`.

1.6.3.1 Example

```
>>> from numpy import array, unravel_index
>>> A = array([[1,2,3], [4,5,6]])
>>> unravel_index(4, A.shape)
(1,1)
```

1.6.4 Generating A 1D Array

The family of functions:

- `arange(m: float) -> array`
- `arange(m: float, n: float) -> array`
- `arange(m: float, n: float, s: float) -> array`

Are similar to the corresponding `range` functions from Python except for two points:

1. The return value is of type `array` (from `numpy`) and not `list`.
2. `m,n,s` can be of type `float` and not just integers.

1.6.4.1 Example

```
>>> from numpy import arange
>>> arange(0,1,0.2)
array([0, 0.2, 0.4, 0.6, 0.8])
```

1.7 More On Numpy

There are a lot lot more that can be told about the `numpy` library. To find more about it, visit the online documentation.