Computer Graphics

1 - Lab: Environment Setting for Lectures - Introduction to NumPy

Yoonsang Lee, Taesoo Kwon

Spring 2019

Topics Covered

- Why Python in this Computer Graphics class?
- Python 2 & Python 3
- Installing Python Interpreter & Additional Modules
- Running Python Interpreter
- Introduction to NumPy

Why Python?

- Productivity
  - Easy to learn.
  - You can write code much faster.
  - You can focus on "logic", not language-specific issues.

- Powerful modules
  - A wide range of reliable modules are available.
  - NumPy for scientific computing, matplotlib for data visualization, …

- **Python allows you to implement key computer graphics concepts**

Why Python?

- Popular in the research communities
  - Most ML / DL frameworks provide Python APIs.
    - TensorFlow, PyTorch, Keras, Theano, ...
  - Most game engines and physics engines also provide Python APIs.

- **Python allows you to**

Install Python Interpreter

- Python **3.5** or later
  - https://www.python.org/downloads/

- Note that all submissions for assignments should work in Python **3.5**.

- You can use any OS that runs Python3

Python 2 & Python 3

- Python 2 is still in active use.

- Python 3 is rapidly gaining popularity especially in the research communities.
  - A lot of very useful features & fixes for well-known problems
  - To do this, **Python 3 breaks backward compatibility**.

- If you're familiar with Python 2, you only need to know the differences between Python 2 and 3.
  - The following link would be helpful:
  - http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
    http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

Install Additional Modules

- We’ll use a few python modules in this class
  - NumPy, PyOpenGL, glfw
- My recommendation for installing python modules is using **pip** (Python Package Index)
- NumPy
  - Windows

```
> py –3 -m pip install numpy
```

  - Ubuntu

```
# if you don't have pip, install it first.
$ sudo apt-get install python3-pip

$ python3 -m pip install numpy
```

Install Additional Modules

- PyOpenGL
  - Windows
    - Download proper *PyOpenGL-3.1.2-cp**3x**-cp**3x**m_xxx.whl* for your system from https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopengl

```
> py -3 -m pip install PyOpenGL-<version in your file>.whl
```

  -

```
$ python3 -m pip install PyOpenGL
```

Install Additional Modules

- GLFW
  - Windows

```
> py -3 -m pip install glfw
```

  - Ubuntu

```
$ sudo apt-get install libglfw3
$ python3 -m pip install glfw
```

Running Python Interpreter 1

- **Interactive mode**
  - Windows: Start, type “cmd”, `> py -3`
  - Ubuntu: Start, type “terminal”, `$ python3`
  - Suitable for simple tests
  - To exit the interpreter, type exit() and press enter key.

```
명령 프롬프트 - py -3
Microsoft Windows [Version 10.0.16299.192]
(c) 2017 Microsoft Corporation. All rights reserved.

C:₩Users₩yoonsang>py -3
Python 3.5.4 (v3.5.4:3f56838, Aug  8 2017, 02:17:05) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 34 - 23
>>> x
11
>>> 
```

Running Python Interpreter 2

- Non-interactive mode (runs a source file)
  - Windows `> py -3 test.py`
  - Ubuntu `$ python3 test.py`
  - In most cases, you will use this mode.

- You can write a Python source file using your favorite editor.
  - Vim, Notepad++, Sublime Text, Atom, IDLE …
  - I’m personally using vim & gvim

Python References

- https://docs.python.org/ko/3/tutorial/index.html
- https://docs.python.org/3/tutorial/index.html
- https://www.tutorialspoint.com/python3/

Introduction to NumPy

- What is NumPy?
  - How to use NumPy
  - Handling vectors & matrices using NumPy

What is NumPy?

- NumPy is a Python module for scientific computing.
  - Written in C
  - Fast vector & matrix operations

- NumPy is **de-facto standard** for numerical computing in Python.

- Very useful for computer graphics applications, which are made of vectors & matrices.

NumPy usage

- Now, let's launch python3 interpreter in the interactive mode and import numpy like this:
- The following NumPy slides are from:
  - https://github.com/enthought/Numpy-Tutorial-SciPyConf-2017/blob/master/slides.pdf

```
>>> import numpy as np
```

: use ‘np’ as the local name for the module numpy

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

CHECKING THE TYPE

```
>>> type(a)
numpy.ndarray
```

NUMERIC "TYPE" OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```

Array Operations

ENTHOUGHT

SIMPLE ARRAY MATH

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([2, 3, 4, 5])
>>> a + b
array([3, 5, 7, 9])

>>> a * b
array([ 2, 6, 12, 20])

>>> a ** b
array([  1,   8,   81,   1024])
```

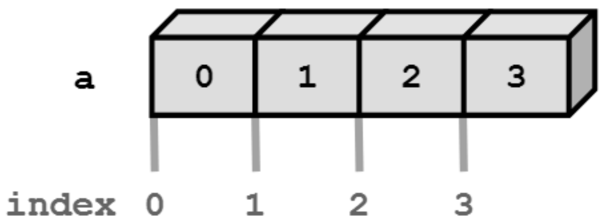
NumPy defines these constants:
pi = 3.14159265359
e = 2.71828182846

```
# multiply entire array by
# scalar value
>>> 0.1 * a
array([0.1, 0.2, 0.3, 0.4])

# in-place operations
>>> a *= 2
>>> a
array([2, 4, 6,
8])
# apply functions to array
>>> x = 0.1*a
>>> x
array([0.2, 0.4, 0.6, 0.8])
>>> y = np.sin(x)
>>> y
array([0.19866933, 0.38941834,
0.56464247, 0.71735609])
```

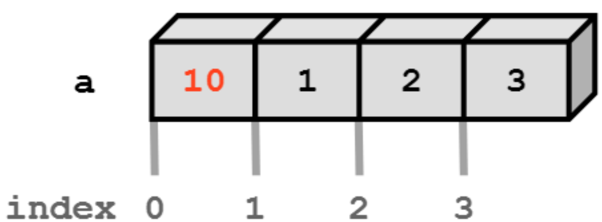
Setting Array Elements

ARRAY INDEXING

```
>>> a[0]
0
```




```
>>> a[0] = 10
>>> a
array([10, 1, 2, 3])
```



BEWARE OF TYPE COERCION

```
>>> a.dtype
dtype('int32')

# assigning a float into
# an int32 array truncates
# the decimal part
>>> a[0] = 10.6
>>> a
array([10, 1, 2, 3])
```

Numpy array: All elements have the same type and the size.

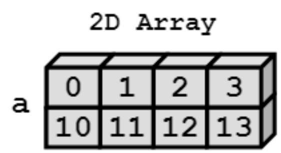
Python list: Elements can have various sizes and types.

Multi-Dimensional Arrays

ENTHOUGHT

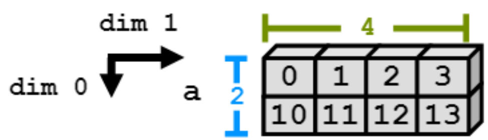
MULTI-DIMENSIONAL ARRAYS

```
>>> a = np.array([[ 0, 1, 2, 3],
...               [10,11,12,13]])
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
```



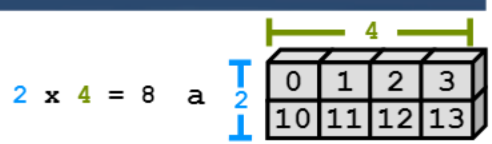
SHAPE = (ROWS, COLUMNS)

```
>>> a.shape
(2, 4)
```



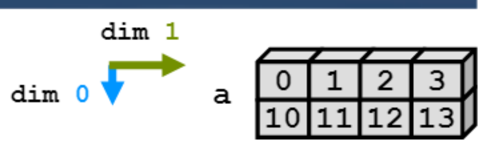
ELEMENT COUNT

```
>>> a.size
8
```



NUMBER OF DIMENSIONS

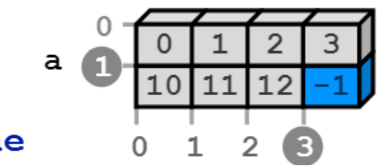
```
>>> a.ndim
2
```



GET / SET ELEMENTS

```
>>> a[1, 3]
13
```


```
>>> a[1, 3] = -1
>>> a
array([[ 0,  1,  2,  3],
       [10, 11, 12, -1]])
```




```
# Shape returns a tuple
# listing the length of the
# array along each dimension.
```

Slicing

ENTHOUGHT

`var[lower:upper:step]`

Extracts a portion of a sequence by specifying a lower and upper bound.
The lower-bound element is included, but the upper-bound element is **not** included.
Mathematically: [lower, upper). The step value specifies the stride between elements.

SLICING ARRAYS

```
#                    -5 -4 -3 -2 -1
# indices:            0  1  2  3  4
>>> a = np.array([10,11,12,13,14])

# [10, 11, 12, 13, 14]
>>> a[1:3]
array([11, 12])

# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

OMITTING INDICIES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list

# grab first three elements
>>> a[:3]
array([10, 11, 12])

# grab last two elements
>>> a[-2:]
array([13, 14])

# every other element
>>> a[::2]
array([10, 12, 14])
```

Array Slicing

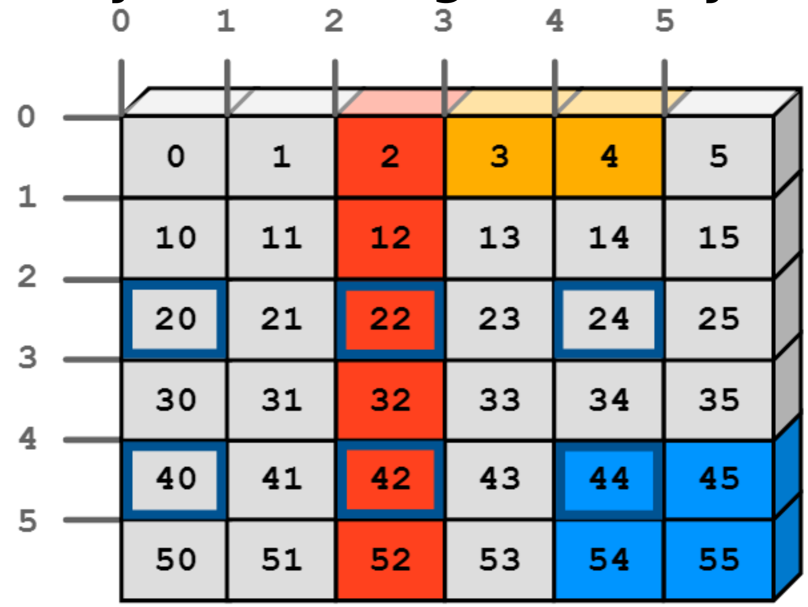
SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
array([2, 12, 22, 32, 42, 52])
```

STRIDED ARE ALSO POSSIBLE

```
>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

a = np.array([[i+10*j for i in range(6)] for j in range(6)])



Array Constructor Examples

ENTHOUGHT

FLOATING POINT ARRAYS

```
# Default to double precision
>>> a = np.array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

REDUCING PRECISION

```
>>> a = np.array([0,1.,2,3],
...              dtype='float32')
>>> a.dtype
dtype('float32')
>>> a.nbytes
16
```

Array Creation Functions

ENTHOUGHT

IDENTITY

$n \times n$ square matrix with ones on the main diagonal and zeros elsewhere.

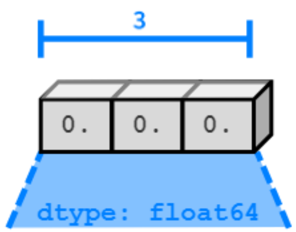
```
# Generate an n by n identity
# array. The default dtype is
# float64.
>>> a = np.identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> a.dtype
dtype('float64')
>>> np.identity(4, dtype=int)
array([[ 1, 0, 0, 0],
       [ 0, 1, 0, 0],
       [ 0, 0, 1, 0],
       [ 0, 0, 0, 1]])
```

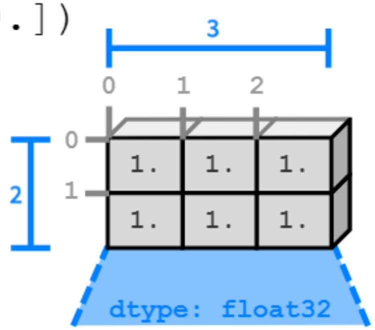
ONES, ZEROS

```
ones(shape, dtype='float64')
zeros(shape, dtype='float64')
```

`shape` is a number or sequence specifying the dimensions of the array. If **dtype** is not specified, it defaults to `float64`.

```
>>> np.ones((2, 3),
...         dtype='float32')
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]],
      dtype=float32)
>>> np.zeros(3)
array([ 0.,  0.,  0.])
```





`zeros(3)` is equivalent to `zeros((3, ))`

Array Creation Functions (cont'd)

LINSPACE

```
# Generate N evenly spaced
# elements between (and including)
# start and stop values.
>>> np.linspace(0, 1 ,5)
array([0., 0.25., 0.5, 0.75, 1.0])
```

ARANGE

```
arange([start,] stop[, step],
                dtype=None)
```

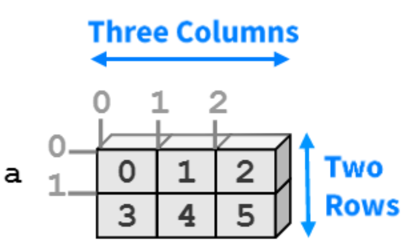
- Nearly identical to Python’s range()
- Creates an array of the interval including *start* but **excluding *stop***
- When using a **non-integer step**, the results will **often not be consistent due to finite machine precision**. It is **better to use linspace() for this case**.

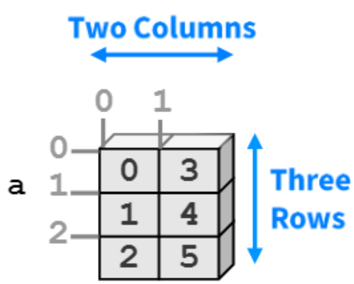
```
>>> np.arange(4)
array([0, 1, 2, 3])

>>> np.arange(1.5, 2.1, 0.3)
array([ 1.5, 1.8, 2.1])
```

Transpose

TRANSPOSE

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

# Transpose swaps the order
# of axes.

>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> a.T.shape
(3,2)
```





Reshaping Arrays

RESHAPE

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

# Return a new array with a
# different shape (a view
# where possible)
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])

# Reshape cannot change the
# number of elements in an
# array
>>> a.reshape(4,2)
ValueError: total size of new
array must be unchanged
```

Vector & Matrix with NumPy

- Vectors are just 1d arrays:

```
>>> v = np.arange(3)
>>> v
array([0,1,2])
```

- Matrices are just 2d arrays:

```
>>> M = np.arange(9).reshape(3,3)
>>> M
array([[0,1,2],
       [3,4,5],
       [6,7,8]])
```

Matrix & Vector Multiplication

- * is an element-wise multiplication operator.

```
>>> v*v
array([0,1,4])
>>> M *M
array([[ 0, 1, 4],
    [ 9,16,25],
    [36,49,64]])
```

- Not so much used in computer graphics.

Matrix & Vector Multiplication

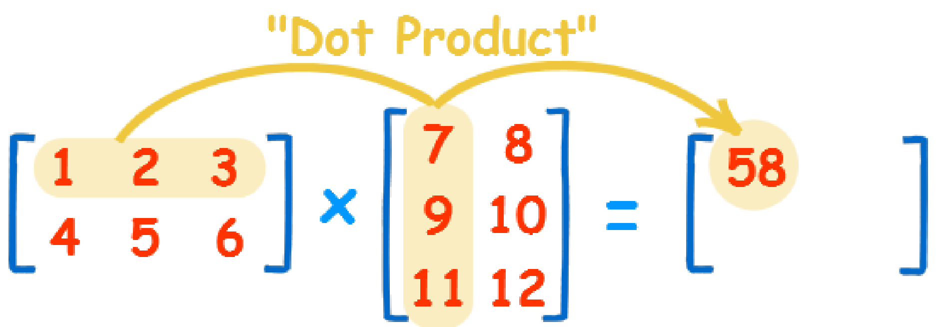
- @ is a *matrix multiplication operator*.

```
>>> v @ v
5
>>> M @ M
array([[ 15, 18, 21],
       [ 42, 54, 66],
       [ 69, 90,111]])
>>> M @ v
array([5,14,23])
```

- Very often used in computer graphics!

Matrix & Vector Multiplication

- Matrix multiplication requires "dot product" (inner product in Euclidian space)



The "Dot Product" is where we **multiply matching members**, then sum up:

$$(1, 2, 3) \bullet (7, 9, 11) = 1\times7 + 2\times9 + 3\times11$$
$$= 58$$

https://www.mathsisfun.com/algebra/matrix-multiplying.html