Python Debugging & Numpy Basics

CS 5670

Qianqian Wang, Kai Zhang and the CS5670 Staff

1. PyCharm Debugging Techniques

See here for basic tutorials

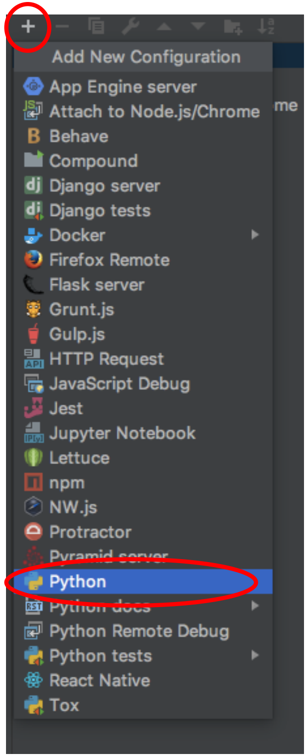
Virtualenv Environment Configurations

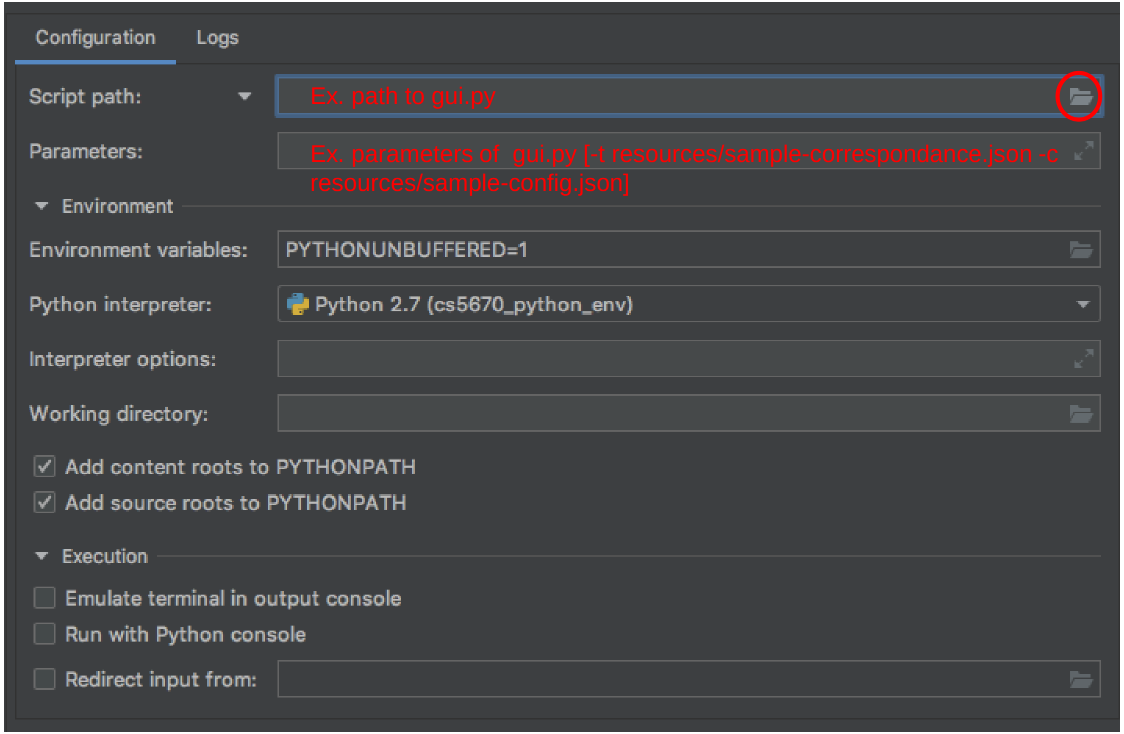
1. In **Settings/Preferences** dialog (⌘,), select **Project: <project name> | Project Interpreter**.
2. In the Project Interpreter page, click and select **Add**.
3. In the left-hand pane of the Add Python Interpreter dialog box, select **Virtualenv Environment.**
4. Select **Existing environment**, Specify the virtual environment in your file system, e.g., `{full path to}/cs5670_python_env/bin/python2.7`

Reference: [Pycharm Help Page]

Run/Debug Configurations

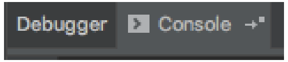
1. Open the Run/Debug Configuration dialog [via **Run | Edit Configurations**]





Reference: [Pycharm Help Page]

Use Pycharm Debugger

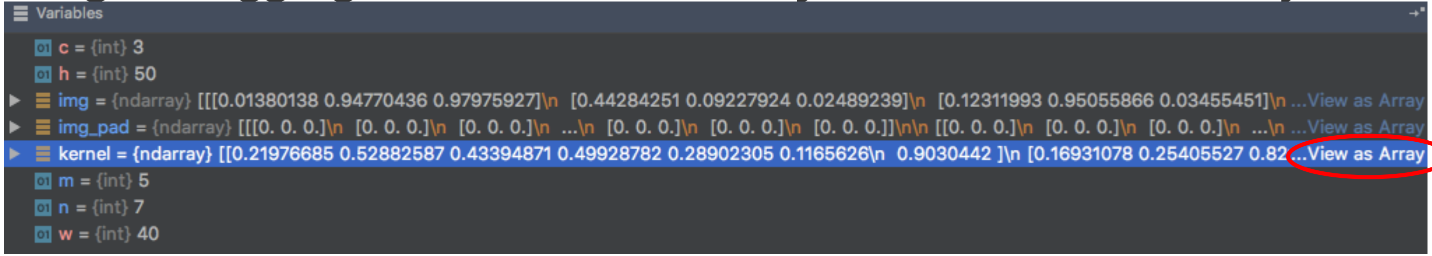
1. Set **breakpoints**: just click in the left gutter
2. Click **Debug** Button
3. Start Debugging!
    a. Step through your program 
    b. Create a watch
    c. Evaluate an expression  or enable the Python console 

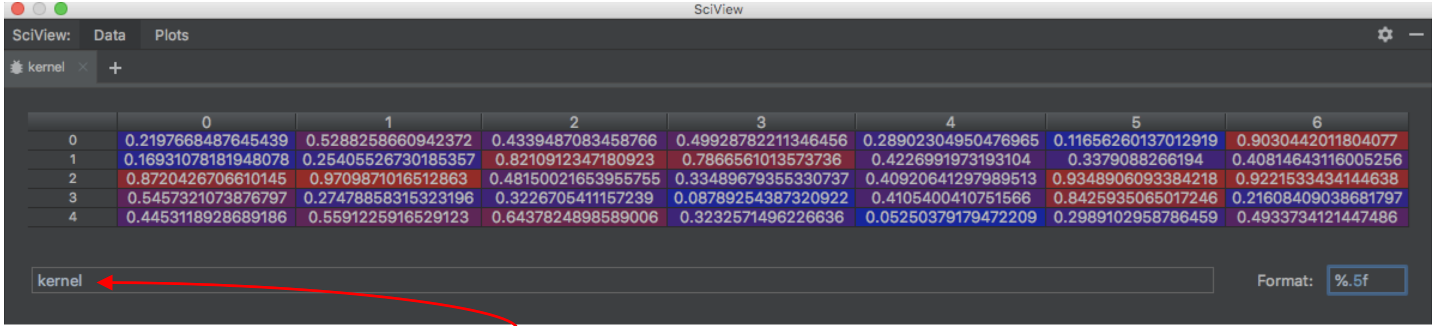
    in the Debugger

Reference: [Pycharm Help Page]

Numpy array visualization

1. During debugging, click ‘**View as Array**’ to visualize the array





	0	1	2	3	4	5	6
0	0.2197668487645439	0.5288258660942372	0.4339487083458766	0.4992878221134656	0.2890230495047696	0.1165626013701291	0.903044201180407
1	0.1693107818194807	0.2540552673018535	0.821091234718092	0.786656101357373	0.422699197319310	0.3379088266194	0.4081464311600525
2	0.872042670661014	0.970987101651286	0.4815002165395575	0.3348967935533073	0.4092064129798951	0.934890609338421	0.922153343414463
3	0.545732107387679	0.2747885831532319	0.322670541115723	0.0878925438732092	0.410540041075156	0.842593506501724	0.2160840903868179
4	0.445311892868918	0.559122591652912	0.643782489858900	0.323257149622663	0.0525037917947220	0.298910295878645	0.493373412144748

Want to visualize high-dimensional array? Try proper slicing

2. Virtual Machine
vs.
Python Virtual Environment

1. Different levels of isolation:
    a. Python Virtual Environment: isolate only python packages
    b. VMs: isolate everything
2. Applications running in a **virtual environment** share an underlying operating system, while **VM** systems can run different operating systems.

3. Numpy Basics

Slicing [Manual]

What is an N Dimensional array?

Write explicitly, `X[0:`m_1`, 0:`m_2`, …, 0:`m_N`]`

N: number of dimensions (axes)

m_1, m_2, …, m_N: length of each dimension (axis)

Tips:

- Slicing is simply setting **an ordered subset**.
  - **range**: `a:b`, ‘`:`’ is a special character that represents the range
  - **logical mask**
  - **any subset**
- Indexing a single element can be viewed as slicing.
  - Compare `X[a, b, c]` with `X[a:a+1, b:b+1, c:c+1]`.
- Dimension loss and expansion.
  - Loss:
    - set the slicing range for a dimension to a single scalar
    - np.sum, np.mean, np.median, ...
  - Expansion:
    - np.newaxis, np.reshape, ...

Slicing Examples

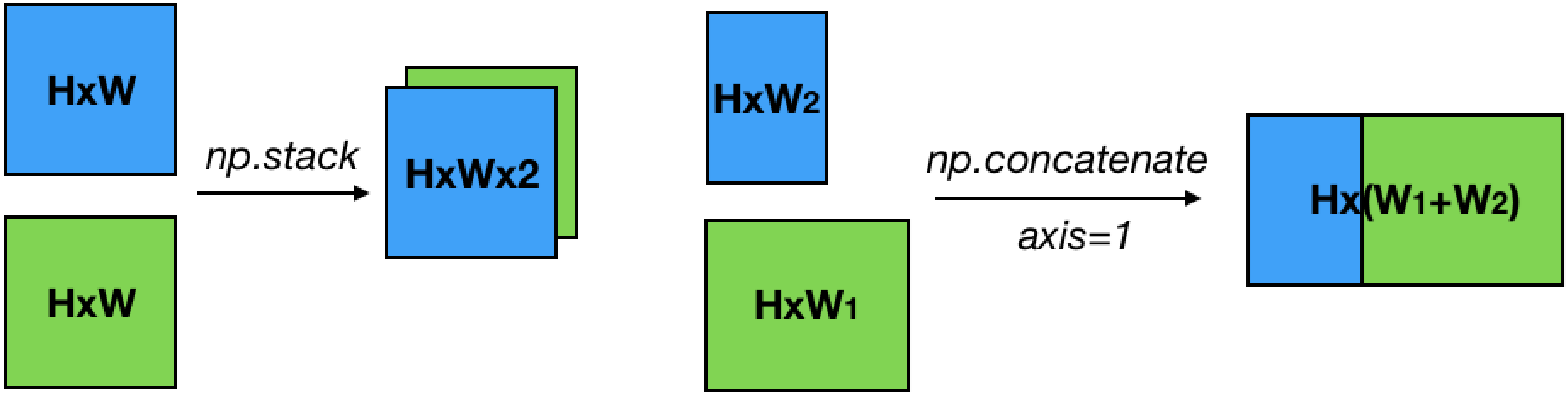
```
>>> import numpy as np
>>> grey = np.random.rand(4, 5)
>>> grey
array([[0.53843361, 0.57248719, 0.15094141, 0.25546857, 0.24077735],
       [0.30343379, 0.27474219, 0.39006497, 0.13884071, 0.30546357],
       [0.01240259, 0.14430918, 0.94647453, 0.32942122, 0.35585103],
       [0.82999887, 0.7653893 , 0.37365659, 0.6500917 , 0.28086595]])
>>> grey[2,3]
0.3294212171303872
>>> grey[2:3,3:4]
array([[0.32942122]])
>>> grey[:, 1]
array([0.57248719, 0.27474219, 0.14430918, 0.7653893 ])
>>> grey[1:3, 2:4]
array([[0.39006497, 0.13884071],
       [0.94647453, 0.32942122]])
>>> np.sum(grey, axis=0)
array([1.68426886, 1.75692786, 1.86113749, 1.37382219, 1.18295791])
>>> mask = grey > 0.5
>>> mask
array([[ True,  True, False, False, False],
       [False, False, False, False, False],
       [False, False,  True, False, False],
       [ True,  True, False,  True, False]])
>>> grey[mask]
array([0.53843361, 0.57248719, 0.94647453, 0.82999887, 0.7653893 ,
       0.6500917 ])
>>> grey[mask].reshape(2,3)
array([[0.53843361, 0.57248719, 0.94647453],
       [0.82999887, 0.7653893 , 0.6500917 ]])
```

Examples:

- Given an RGB image `X[0:h, 0:w, 0:3]`
- Get G channel of a RGB image:
  `X[:, :, 1]`
- RGB to BGR
  `X[:, :, [2, 1, 0]]`
- Center-crop an RGB image
  $X[r_1:r_2,\ c_1:c_2\ ,\ :]$
- Downsample an RGB image by 2x
  `X[0:h:2, 0:w:2, :]`

Stacking [Manual] and Concatenating [Manual]

1. `np.stack()`, `np.concatenate()`
2. `np.stack()` requires that all input array must have the **same shape**, and the stacked array has **one more dimension** than the input arrays.
3. `np.concatenate()` requires that the input arrays must have the same shape, **except in the dimension** corresponding to *axis*



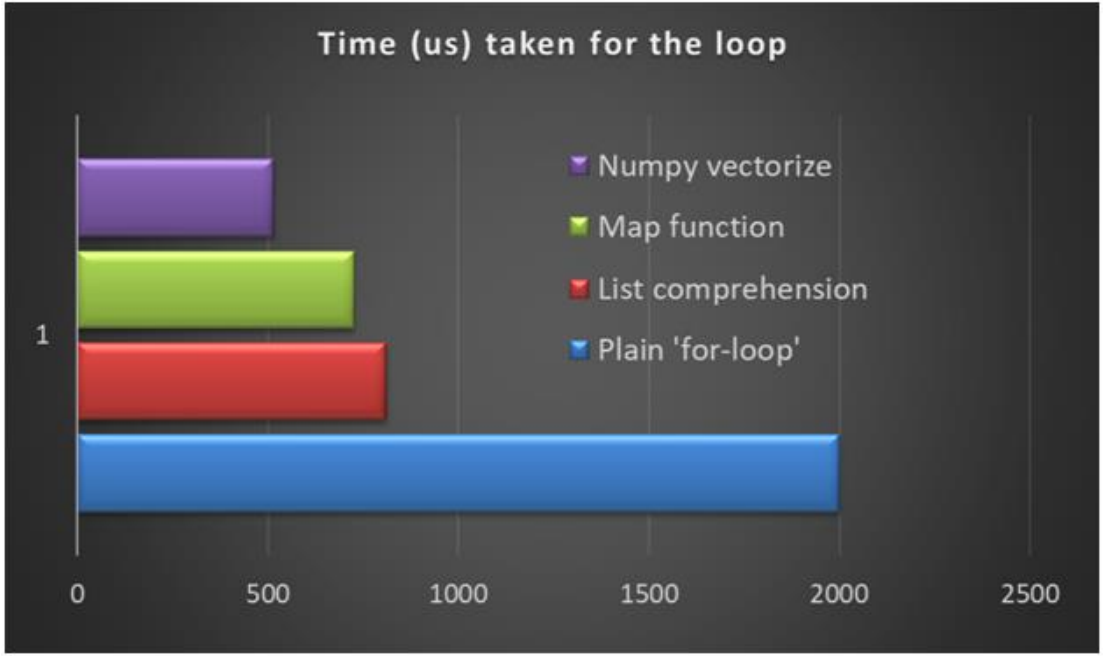
Concatenation Examples

```
>>> R=np.random.randn(300, 400)
>>> G=np.random.randn(300, 400)
>>> B=np.random.randn(300, 400)
>>> RGB=np.concatenate((R[:, :, np.newaxis], G[:, :, np.newaxis], B[:, :, np.newaxis]), axis=2)
>>>
>>>
>>> R.shape
(300, 400)
>>> G.shape
(300, 400)
>>> B.shape
(300, 400)
>>> RGB.shape
(300, 400, 3)
>>>
```

```
>>> another_RGB=np.concatenate((R[np.newaxis, :, :], G[np.newaxis, :, :], B[np.newaxis, :, :]), axis=0)
>>> another_RGB.shape
(3, 300, 400)
>>>
```

Vectorization

1. Turn your loops to Numpy vector manipulation
2. Vectorization enables fast **parallel computation**



Vectorization

Example 1: element-wise multiplication

For-Loop -- Inefficient	Numpy Vector -- Efficient!

`>>> a = [1, 2, 3, 4, 5]`
`>>> b = [6, 7, 8, 9, 10]`
`>>> [x * y for x, y in zip(a, b)]`
`[6, 14, 24, 36, 50]`
	`>>> import numpy as np`
	`>>> a = np.array([1, 2, 3, 4, 5])`
	`>>> b = np.array([6, 7, 8, 9, 10])`
	`>>> a * b`
	`array([ 6, 14, 24, 36, 50])`

Vectorization

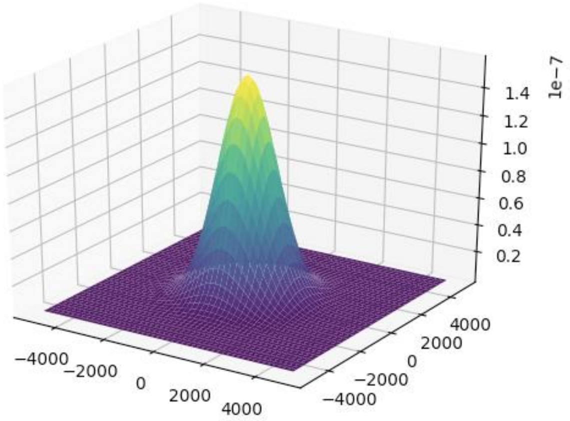
Example 2: compute gaussian kernel

<table>
<tr><td>For Loop</td><td>
<pre><code>hc = height // 2
wc = width // 2
gaussian = np.zeros((height, width))
for i in range(height):
    for j in range(width):
        gaussian[i, j] = np.exp(-((i - hc)**2 + (j - wc)**2)/(2.0*sigma**2))
gaussian /= np.sum(gaussian)</code></pre>
</td></tr>
<tr><td>Numpy Vector</td><td>
<pre><code>hc = height // 2
wc = width // 2
grid = np.mgrid[-hc:hc+1, -wc:wc+1] # 2 x height x width
gaussian = np.exp(-np.sum(grid**2, axis=0)/(2.0*sigma**2))
gaussian /= np.sum(gaussian)</code></pre>
</td></tr>
</table>

Vectorization

Example 2: compute gaussian kernel and plot

Height = width = 9999, sigma = 1000
For Loop: ~106s
Vectorization: ~12s



Other useful functions:

1. vector operations: inner product [`np.inner()`], outer product [`np.outer()`], cross product [`np.cross()`], matrix multiplication [`np.dot()`] , matrix inverse [`np.linalg.inv()`]
2. special matrices/vectors: `np.zeros(), np.ones(), np.identity(), np.linspace(), np.arange()`
3. matrix reshaping: `np.reshape(), np.transpose()`

`(row_axis, column_axis, channel_axis) → (channel_axis, row_axis, column_axis): np.transpose(X,[2, 0, 1])`

1. statistics: `np.min(), np.max(), np.mean(), np.median(), np.sum()`
2. logical arrays: `np.logical_and(), np.logical_or(), np.logical_not()`