CS 231n Python & NumPy Tutorial

Python 3 and NumPy will be used extensively throughout this course, so it's important to be familiar with them.

A good amount of the material in this notebook comes from Justin Johnson's Python & NumPy Tutorial: http://cs231n.github.io/python-numpy-tutorial/. At this moment, not everything from that tutorial is in this notebook and not everything from this notebook is in the tutorial.

Python 3

If you're unfamiliar with Python 3, here are some of the most common changes from Python 2 to look out for.

Print is a function

In [1]:
```python
print("Hello!")
```

```
Hello!
```

Without parentheses, printing will not work.

In [2]:
```python
print "Hello!"
```

```
  File "/var/folders/7_/dw3jfv_s2vqby4klxk9qy39w0000gq/T/ipykernel_9627/370071
6981.py", line 1
    print "Hello!"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hell
o!")?
```

Floating point division by default

In [3]:
```python
5 / 2
```

Out[3]:
```
2.5
```

To do integer division, we use two backslashes:

In [4]:
```python
5 // 2
```

Out[4]:
```
2
```

No xrange

The xrange from Python 2 is now merged into "range" for Python 3 and there is no xrange in Python 3. In Python 3, range(3) does not create a list of 3 elements as it would in Python 2, rather just creates a more memory efficient iterator.

Hence,
xrange in Python 3: Does not exist

range in Python 3: Has very similar behavior to Python 2's xrange

In [5]:
```
for i in range(3):
    print(i)
```
```
0
1
2
```

In [6]:
```
for i in range(1, 5, 2):
    print(i)
```
```
1
3
```

In [7]:
```
list(range(3)) + list(range(3))
```
Out[7]:
```
[0, 1, 2, 0, 1, 2]
```

In [8]:
```
# If need be, can use the following to get a similar behavior to Python 2's r
print(list(range(3)))
```
```
[0, 1, 2]
```

NumPy

"NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more"
-https://docs.scipy.org/doc/numpy-1.10.1/user/whatisnumpy.html.

In [9]:
```
import numpy as np
```

Let's run through an example showing how powerful NumPy is. Suppose we have two lists a and b, consisting of the first 100,000 non-negative numbers, and we want to create a new list c whose *i*th element is a[i] + 2 * b[i].

Without NumPy:

In [10]:
```
%%time
a = [i for i in range(100000)]
b = [i for i in range(100000)]
```
```
CPU times: user 8.76 ms, sys: 4.33 ms, total: 13.1 ms
Wall time: 15.5 ms
```

In [11]:
```
%%time
c = []
for i in range(len(a)):
    c.append(a[i] + 2 * b[i])
```

```
CPU times: user 34.4 ms, sys: 3.97 ms, total: 38.4 ms
Wall time: 38.1 ms
```

With NumPy:

In [12]:

```
%%time
a = np.arange(100000)
b = np.arange(100000)
```

```
CPU times: user 2.89 ms, sys: 2.33 ms, total: 5.23 ms
Wall time: 3.59 ms
```

In [13]:

```
%%time
c = a + 2 * b
```

```
CPU times: user 2.48 ms, sys: 1.47 ms, total: 3.95 ms
Wall time: 3.06 ms
```

The result is 10 to 15 times faster, and we could do it in fewer lines of code (and the code itself is more intuitive)!

Regular Python is much slower due to type checking and other overhead of needing to interpret code and support Python's abstractions.

For example, if we are doing some addition in a loop, constantly type checking in a loop will lead to many more instructions than just performing a regular addition operation. NumPy, using optimized pre-compiled C code, is able to avoid a lot of the overhead introduced.

The process we used above is **vectorization**. Vectorization refers to applying operations to arrays instead of just individual elements (i.e. no loops).

Why vectorize?

1. Much faster
2. Easier to read and fewer lines of code
3. More closely assembles mathematical notation

Vectorization is one of the main reasons why NumPy is so powerful.

ndarray

ndarrays, n-dimensional arrays of homogenous data type, are the fundamental datatype used in NumPy. As these arrays are of the same type and are fixed size at creation, they offer less flexibility than Python lists, but can be substantially more efficient runtime and memory-wise. (Python lists are arrays of pointers to objects, adding a layer of indirection.)

The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

In [14]:

```
# Can initialize ndarrays with Python lists, for example:
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1, 2, 3],
                [4, 5, 6]])    # Create a rank 2 array
print(b.shape)                       # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])    # Prints "1 2 4"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```

In [ ]:

```

```

In [15]:

```
print(type(b.shape))
```

```
<class 'tuple'>
```

There are many other initializations that NumPy provides:

In [16]:

```
a = np.zeros((2, 2))   # Create an array of all zeros
print(a)               # Prints "[[ 0.  0.]
                       #          [ 0.  0.]]"

b = np.full((2, 2), 7)  # Create a constant array
print(b)                # Prints "[[ 7.  7.]
                        #          [ 7.  7.]]"

c = np.eye(2)         # Create a 2 x 2 identity matrix
print(c)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

d = np.random.random((2, 2))  # Create an array filled with random values
print(d)                      # Might print "[[ 0.91940167  0.08143941]
                              #               [ 0.68744134  0.87236687]]"
```

```
[[0. 0.]
 [0. 0.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.9279557  0.80037539]
 [0.67138692 0.50428401]]
```

How do we create a 2 by 2 matrix of ones?

In [17]:

```
a = np.ones((2, 2))    # Create an array of all ones
print(a)               # Prints "[[ 1.  1.]
                       #          [ 1.  1.]]"
```

```
[[1. 1.]
 [1. 1.]]
```

Useful to keep track of shape; helpful for debugging and knowing dimensions will be very useful when computing gradients, among other reasons.

NumPy supports an object-oriented paradigm, such that ndarray has a number of methods and attributes, with functions similar to ones in the outermost NumPy namespace. For example, we can do both:

In [18]:

```
nums = np.arange(8)
```

```
print(nums.min())      # Prints 0
print(np.min(nums))    # Prints 0
```

```
0
0
```

Array Operations/Math

NumPy supports many elementwise operations:

In [19]:

```
x = np.array([[1, 2],
              [3, 4]], dtype=np.float64)
y = np.array([[5, 6],
              [7, 8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

How do we elementwise divide between two arrays?

In [20]:

```
x = np.array([[1, 2], [3, 4]], dtype=np.float64)
y = np.array([[5, 6], [7, 8]], dtype=np.float64)

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

Note * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

In [21]:

```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])

v = np.array([9, 10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

```
219
219
[29 67]
[29 67]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

There are many useful functions built into NumPy, and often we're able to express them across specific axes of the ndarray:

In [22]:

```
vv = np.array([[9], [10]])
print(vv.shape)
print(x.dot(vv))
```

```
(2, 1)
[[29]
 [67]]
```

In [23]:

```
x = np.array([[1, 2, 3],
              [4, 5, 6]])

print(np.sum(x))          # Compute sum of all elements; prints "21"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[5 7 9]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[6 15]"

print(np.max(x, axis=1))  # Compute max of each row; prints "[3 6]"
```

```
21
[5 7 9]
```

```
[ 6 15]
[3 6]
```

How can we compute the index of the max value of each row? Useful, to say, find the class that corresponds to the maximum score for an input image.

In [24]:

```
x = np.array([[1, 2, 3],
              [4, 6, 5]])

print(np.argmax(x, axis=1)) # Compute index of max of each row; prints "[2 2]
```

```
[2 1]
```

Note the axis you apply the operation will have its dimension removed from the shape. This is useful to keep in mind when you're trying to figure out what axis corresponds to what.

For example:

In [25]:

```
x = np.array([[1, 2, 3],
              [4, 5, 6]])

print(x.shape)                # Has shape (2, 3)
print((x.max(axis=0)).shape) # Taking the max over axis 0 has shape (3,)
                              # corresponding to the 3 columns.

# An array with rank 3
x = np.array([[[1, 2, 3],
               [4, 5, 6]],
              [[10, 23, 33],
               [43, 52, 16]]
             ])

print(x)
print(x.shape)                # Has shape (2, 2, 3)
print((x.max(axis=1)).shape) # Taking the max over axis 1 has shape (2, 3)

print((x.max(axis=(1, 2))))       # Can take max over multiple axes; prints [
print((x.max(axis=(1, 2))).shape) # Taking the max over axes 1, 2 has shape (
```

```
(2, 3)
(3,)
[[[ 1  2  3]
  [ 4  5  6]]

 [[10 23 33]
  [43 52 16]]]
(2, 2, 3)
(2, 3)
[ 6 52]
(2,)
```

In [26]:

```
print(x.max(axis=2))
```

```
[[ 3  6]
 [33 52]]
```

Indexing

NumPy also provides powerful indexing schemes.

In [27]:

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
print('Original:\n', a)

# Can select an element as you would in a 2 dimensional Python list
print('Element (0, 0) (a[0][0]):\n', a[0][0])   # Prints 1
# or as follows
print('Element (0, 0) (a[0, 0]) :\n', a[0, 0])  # Prints 1

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print('Sliced (a[:2, 1:3]):\n', b)

# Steps are also supported in indexing. The following reverses the first row:
print('Reversing the first row (a[0, ::-1]) :\n', a[0, ::-1]) # Prints [4 3 2
```

```
Original:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Element (0, 0) (a[0][0]):
 1
Element (0, 0) (a[0, 0]) :
 1
Sliced (a[:2, 1:3]):
 [[2 3]
 [6 7]]
Reversing the first row (a[0, ::-1]) :
 [4 3 2 1]
```

Often, it's useful to select or modify one element from each row of a matrix. The following example employs **fancy indexing**, where we index into our array using an array of indices (say an array of integers or booleans):

In [28]:
```
# Create a new array from which we will select elements
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                 [17,  8,  9],
          #                 [10, 21, 12]])
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 1  6  7 11]
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

In [29]:

```
c = [np.arange(4), b]
print(c)
```

```
[array([0, 1, 2, 3]), array([0, 2, 0, 1])]
```

We can also use boolean indexing/masks. Suppose we want to set all elements greater than MAX to MAX:

In [30]:

```
MAX = 5
nums = np.array([1, 4, 10, -1, 15, 0, 5])
print(nums > MAX)            # Prints [False, False, True, False, True, False

nums[nums > MAX] = MAX
print(nums)                  # Prints [1, 4, 5, -1, 5, 0, 5]
```

```
[False False  True False  True False False]
[ 1  4  5 -1  5  0  5]
```

In [31]:

```
MAX = 5
nums = np.array([1, 4, 10, -1, 15, 0, 5])
print(nums > MAX)
nums[nums > MAX]
```

```
[False False  True False  True False False]
```

Out[31]:

```
array([10, 15])
```

Finally, note that the indices in fancy indexing can appear in any order and even multiple times:

In [32]:

```
nums = np.array([1, 4, 10, -1, 15, 0, 5])
print(nums[[1, 2, 3, 1, 0]])  # Prints [4 10 -1 4 1]
```

```
[ 4 10 -1  4  1]
```

Broadcasting

In [33]:

```
nums = np.array([1, 4, 10, -1, 15, 0, 5])
print(nums[[1, 2]])  # Prints [4 10 -1 4 1]
```

```
[ 4 10]
```

Many of the operations we've looked at above involved arrays of the same rank.
However, many times we might have a smaller array and use that multiple times to update an array of a larger rank.

For example, consider the below example of shifting the mean of each column from the elements of the corresponding column:

In [34]:
```
x = np.array([[1, 2, 3],
              [3, 5, 7]])
print(x.shape)  # Prints (2, 3)

col_means = x.mean(axis=0)
print(col_means)          # Prints [2. 3.5 5.]
print(col_means.shape)    # Prints (3,)
                          # Has a smaller rank than x!

mean_shifted = x - col_means
print('\n', mean_shifted)
print(mean_shifted.shape)  # Prints (2, 3)
```

```
(2, 3)
[2.  3.5 5. ]
(3,)

 [[-1.  -1.5 -2. ]
 [ 1.   1.5  2. ]]
(2, 3)
```

Or even just multiplying a matrix by 2:

In [35]:
```
x = np.array([[1, 2, 3],
              [3, 5, 7]])
print(x * 2) # Prints [[ 2   4   6]
             #         [ 6  10  14]]
```

```
[[ 2  4  6]
 [ 6 10 14]]
```

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

For example, when subtracting the columns above, we had arrays of shape (2, 3) and (3,).

1. These arrays do not have same rank, so we prepend the shape of the lower rank one to make it (1, 3).
2. (2, 3) and (1, 3) are compatible (have the same size in the dimension, or if one of the arrays has size 1 in that dimension).
3. Can be broadcast together!
4. After broadcasting, each array behaves as if it had shape equal to (2, 3).
5. The smaller array will behave as if it were copied along dimension 0.

Let's try to subtract the mean of each row!

```
In [36]:  x = np.array([[1, 2, 3],
                        [3, 5, 7]])

          row_means = x.mean(axis=1)
          print(row_means)  # Prints [2. 5.]

          mean_shifted = x - row_means
```

```
[2. 5.]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/7_/dw3jfv_s2vqby4klxk9qy39w0000gq/T/ipykernel_9627/4090913965.py
 in <module>
      5 print(row_means)  # Prints [2. 5.]
      6 
----> 7 mean_shifted = x - row_means

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

To figure out what's wrong, we print some shapes:

```
In [37]:  x = np.array([[1, 2, 3],
                        [3, 5, 7]])
          print(x.shape)  # Prints (2, 3)

          row_means = x.mean(axis=1)
          print(row_means)        # Prints [2. 5.]
          print(row_means.shape)  # Prints (2,)

          # Results in the following error: ValueError: operands could not be broadcast
          mean_shifted = x - row_means
```

```
(2, 3)
[2. 5.]
(2,)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/7_/dw3jfv_s2vqby4klxk9qy39w0000gq/T/ipykernel_9627/1751571366.py
 in <module>
      8 
      9 # Results in the following error: ValueError: operands could not be br
oadcast together with shapes (2,3) (2,)
---> 10 mean_shifted = x - row_means

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

What happened?

Answer: If we following broadcasting rule 1, then we'd prepend a 1 to the smaller rank array ot get (1, 2). However, the last dimensions don't match now between (2, 3) and (1, 2), and so we can't broadcast.

Take 2, reshaping the row means to get the desired behavior:

```
In [38]:  x = np.array([[1, 2, 3],
                        [3, 5, 7]])
          print(x.shape)  # Prints (2, 3)

          row_means = x.mean(axis=1).reshape((-1, 1))
          print(row_means)        # Prints [[2.], [5.]]
          print(row_means.shape)  # Prints (2, 1)

          mean_shifted = x - row_means
```

```
print(mean_shifted)
print(mean_shifted.shape)  # Prints (2, 3)
```

```
(2, 3)
[[2.]
 [5.]]
(2, 1)
[[-1.  0.  1.]
 [-2.  0.  2.]]
(2, 3)
```

More broadcasting examples!

In [39]:
```
# Compute outer product of vectors
v = np.array([1, 2, 3])  # v has shape (3,)
w = np.array([4, 5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1, 2, 3], [4, 5, 6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[2 4 6]
 [5 7 9]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
```

Views vs. Copies

Unlike a copy, in a **view** of an array, the data is shared between the view and the array. Sometimes, our results are copies of arrays, but other times they can be views. Understanding when each is generated is important to avoid any unforeseen issues.

Views can be created from a slice of an array, changing the dtype of the same data area (using arr.view(dtype), not the result of arr.astype(dtype)), or even both.

In [40]:
```
x = np.arange(5)
print('Original:\n', x)  # Prints [0 1 2 3 4]

# Modifying the view will modify the array
view = x[1:3]
view[1] = -1
print('Array After Modified View:\n', x)  # Prints [0 1 -1 3 4]
```

```
Original:
 [0 1 2 3 4]
Array After Modified View:
 [ 0  1 -1  3  4]
```

In [41]:
```
x = np.arange(5)
view = x[1:3]
view[1] = -1

# Modifying the array will modify the view
print('View Before Array Modification:\n', view)  # Prints [1 -1]
x[2] = 10
print('Array After Modifications:\n', x)          # Prints [0 1 10 3 4]
print('View After Array Modification:\n', view)   # Prints [1 10]
```

```
View Before Array Modification:
 [ 1 -1]
Array After Modifications:
 [ 0  1 10  3  4]
View After Array Modification:
 [ 1 10]
```

However, if we use fancy indexing, the result will actually be a copy and not a view:

In [42]:
```
x = np.arange(5)
print('Original:\n', x)  # Prints [0 1 2 3 4]

# Modifying the result of the selection due to fancy indexing
# will not modify the original array.
copy = x[[1, 2]]
copy[1] = -1
print('Copy:\n', copy) # Prints [1 -1]
print('Array After Modified Copy:\n', x)  # Prints [0 1 2 3 4]
```

```
Original:
 [0 1 2 3 4]
Copy:
 [ 1 -1]
Array After Modified Copy:
 [0 1 2 3 4]
```

In [43]:
```
# Another example involving fancy indexing
x = np.arange(5)
print('Original:\n', x)  # Prints [0 1 2 3 4]

copy = x[x >= 2]
print('Copy:\n', copy) # Prints [2 3 4]
x[3] = 10
print('Modified Array:\n', x)  # Prints [0 1 2 10 4]
print('Copy After Modified Array:\n', copy)  # Prints [2 3 4]
```

```
Original:
 [0 1 2 3 4]
Copy:
 [2 3 4]
Modified Array:
 [ 0  1  2 10  4]
Copy After Modified Array:
 [2 3 4]
```

Summary

1. NumPy is an incredibly powerful library for computation providing both massive efficiency gains and convenience.
2. Vectorize! Orders of magnitude faster.
3. Keeping track of the shape of your arrays is often useful.
4. Many useful math functions and operations built into NumPy.
5. Select and manipulate arbitrary pieces of data with powerful indexing schemes.
6. Broadcasting allows for computation across arrays of different shapes.
7. Watch out for views vs. copies.

In [44]:
```
# Kan Horst - PKU - 干皓丞
```