MLCV 182:
Practical session 1
Ron Shapira Weber
Computer Science, Ben-Gurion University
Getting Started

• There are two different versions of Python being supported at the moment, 2.7 and 3.6. For compatibility reasons, in this course we shall use the following Python (and packages) versions:
  • Python 2.7
  • Numpy 1.11.*
  • Scipy 0.18.*
  • Matplotlib 1.5.*
  • OpenCV 3.1.2 (with Python bindings).
• You can find instructions for **Python 2.7** installation via Anaconda and OpenCV [here](#). Anaconda provides an easy way to manage environments, packages and scientific notebooks, and includes apps such as Spyder and Jupyter Notebook.
• If you chose to use Anaconda, you can read how to manage your packages version [here](#), otherwise you should probably use pip.
Getting Started

• There are two different versions of Python being supported at the moment, 2.7 and 3.6. For compatibility reasons, in this course we shall use the following Python (and packages) versions:
  • Python 2.7
  • Numpy 1.11.*
  • Scipy 0.18.*
  • Matplotlib 1.5.*
  • OpenCV 3.1.2 (with Python bindings).
• You can find instructions for Python 2.7 installation via Anaconda and OpenCV here. Anaconda provides an easy way to manage environments, packages and scientific notebooks, and includes apps such as Spyder and Jupyter Notebook.
• If you chose to use Anaconda, you can read how to manage your packages version here, otherwise you should probably use pip.
Anaconda

**Jupyter Notebook**

5.4.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

**Qtconsole**

4.3.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

**Spyder**

3.2.6
Scientific Python Development Environment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features.
IPython Console

- Interactive Python Shell – allows for MATLAB like interactive sessions.
- Line by line code execution.
- Supports browser-based notebook.
- Support for interactive data visualization and use of GUI toolkits.
IPython

```
IPython 5.4.1 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help     -> Python's own help system.
object?  -> Details about 'object', use 'object??' for extra details.

In [1]: Ipython = "Wow, so very interactive"

In [2]: Ipython
Out[2]: 'Wow, so very interactive'

In [3]:
```
Jupyter Notebook

- Interactive Python Shell.
- Runs in the browser.
- Data visualization between blocks of code
Spyder

• Open source Python IDE for scientific programming.
• Ipython shell (Could run multiple instances)
• Variable explorer
And now for some Python
Numbers

\[
x = 4 \quad \text{# 'int' is the default type. notice there's no need for ; at the end of a statement.}
\]
\[
print(x) \quad \text{# Prints "2".}
\]
\[
print(type(x)) \quad \text{# Prints "<class 'int'>"}
\]
\[
print(x + 1) \quad \text{# Addition; prints "5"}
\]
\[
print(x - 1) \quad \text{# Subtraction; prints "3"}
\]
\[
print(x * 2) \quad \text{# Multiplication; prints "8"}
\]
\[
print(x ** 2) \quad \text{# Exponentiation; prints "16"}
\]
\[
print(x // 2.5) \quad \text{# (floored) quotient of x and y; prints "1.0"}
\]
\[
print(x % 2.5) \quad \text{# module / remainder of x / y; prints "1.5"}
\]
\[
x += 1 \quad \text{# There's no x++ in Python, so this is the way to go. prints "5"}
\]
\[
x *= 2 \quad \text{# prints "8" # working with floats is similar}
\]
\[
y = 1.5
\]
\[
print(type(y)) \quad \text{# Prints "<class 'float'>" # casting from one type to another:}
\]
\[
x = 2.5 \quad \text{# type(x) = 'float'}
\]
\[
\text{int(x) # casts x to an integer; prints "2" #Note that}
\]
\[
\text{print x,y and print(x,y) is not equal in Python 2.7}
\]
\[
print(x,y) (1, 2) \quad \text{#this is a tuple}
\]
\[
print x,y 1 2 \quad \text{#these are two integers}
\]
Booleans

a = True
b = False
a and b # False; equal to a & b
a or b # True; equal to a | b
not a # False
a != b # True
print int(a) # prints "1"
Strings

```python
hello = 'hello'  # String variables can use single quotes
world = "world"  # or double quotes;
print(hello)  # Prints "hello"
print hello  # Also prints "hello". Python 2.7 print function needs no parentheses
print(len(hello))  # String length; prints "5"
helloWorld = hello + ' ' + world  # String concatenation
print(helloWorld)  # prints "hello world"
print hello, 27  # print "hello 27"
helloWorld27 = '%s %s %d' % (hello, world, 27)  # printf style string formatting
print(helloWorld27)  # prints "hello world 27"
```
Data Structures
Lists

- Lists are used to group together items/values and function similar to arrays. They are capable of storing different types of items and are resizeable.

```python
squares = [1, 4, 9, 16, 25]
squares[0]  # Python is zero-based; returns "1"
squares[-1]  # returns the last item in the list; "25"
mixed list = [4, 2.5, 'nine']  # different types of items could be stored in a list
squares + [36, 49, 64, 81, 100]  # list concatenation;
# "[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]"
squares[2] = 99  # lists are mutable; "[1, 4, 99, 16, 25]"

# a common way to add items to a list is via the append() method:
   squares.append(216)  # add 216 as the last value
   squares.append(7 ** 3)  # add 343 as the last value
# squares: [1, 8, 27, 64, 125, 216, 343]
```
Lists

# slicing is an easy way to access and manipulate items in a list # it returns a new (shallow) copy of the list:

squares[:] # "[1, 4, 9, 16, 25]"
nums = range(5) # built-in function creates a list of numbers; # "[0,1,2,3,4]"
nums_even = range(0,10,2) # "from 0 to 10 (exclusive) in steps of 2;
"[0, 2, 4, 6, 8]"
even_reverse = range(10,0,-2) # "from 10 to 0 (exclusive) in steps of -2; "[10, 8, 6, 4, 2]"

nums[2:4] # Get a slice from index 2 to 4 (exclusive);"[2, 3]
nums[2:] # Get a slice from index 2 to the end; prints "[2, 3, 4]"
nums[:2] # Get a slice from the start to index 2 (exclusive);"[2, 3, 4]"
squares[-3:] # slicing returns a new list; "[9, 16, 25]"
nums[2:4] = [8, 9] # Assign a new sublist to a slice # list could act as multi-dimensional arrays
A = [[1,2],[3,4]] # a 2x2 array
A[0][1] #returns "1". but we'll stop here since numpy is the way to go
Lists - Loops

• Iterating in python feels almost like pseudo code:

```python
# For loops:
bag = ['notebook', 'keys', 'lipstick']
for stuff in bag:
    print(stuff)  # Python uses indentation to identify blocks of code
# prints 'notebook', 'keys', 'lipstick'

# you can also add indices via the enumerate method
for idx, _ in enumerate(bag):  # _ is a throw-away variable
    print(idx)
# prints '0, 1, 2'
# While loops:
count = 0
while (count < 9):
    print(count)  count = count + 1
```

• List Comprehensions: Python supports list comprehensions, which allows for creating and manipulating lists in a single line of code:

```python
S = [x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
M = [x for x in S if x % 2 == 0]
# only even numbers in S # [0, 4, 16, 36, 64]
```
Dictionaries

- A dictionary stores (key, value) pairs. Dictionaries are indexed by key and not by indices, so it is best to think of a dictionary as an unordered set of key: value pairs.

```python
# This code is taken from an IPython session.
# Note that writing a variable name will 'Out' it's value
In[1]: n_seasons = {'GoT': 7, 'Friends': 10}
In[2]: n_seasons['GoT'] # getting the value stored under the key 'GoT'
Out[2]: 7
In[3]: n_seasons['Simpsons'] = 'inf' # adding a new (key, value)
In[4]: n_seasons
Out[4]: {'Friends': 10, 'GoT': 7, 'Simpsons': 'inf'}
In [5]: n_seasons['GoT'] = 8 # dictionary values are mutable
In [6]: n_seasons
Out[6]: {'Friends': 10, 'GoT': 8, 'Simpsons': 'inf'}

- Some useful functions:

```python
del n_seasons['Friends'] # deletes the pair ('Friends', 10)
list(n_seasons.keys()) # returns an unsorted list of keys # ['Simpsons', 'GoT']
sorted(n_seasons.keys()) # returns a sorted list of keys # ['GoT', 'Simpsons']
'GoT' in n_seasons # True
'Fauda' in n_seasons # False
```
Dictionaries - Loops

- You can iterate over dictionary keys, and use list comprehensions as well.

```python
for tv_show, seasons in n_seasons.items():
    print(tv_show, seasons)
# ('Simpsons', 'inf') <-- tuple
# ('GoT', 8)
S = {x:x**2 for x in range(4)}  # note the curly brackets
# {0: 0, 1: 1, 2: 4, 3: 9}
```
Tuples

- A tuple is an (immutable) ordered list of values.

```python
t = (1,2)  
t[0] #prints 1  
t = (1,2 , 'dog')  
t[2] #prints 'dog'  
t[2] = 'cat' #Error! tuples are immutable  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment  
# When there're no brackets, Python will recognize the data as a tuple  
x, y = range(2)  #0,1  
print (x,y)  
#(0,1)
```
Tuples

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```python
empty = ()
singleton = 'hello',  # <-- note trailing comma
print(len(empty))
# 0
print(len(singleton))
# 1
print(empty)
# ()
print(singleton)
# ('hello',)
```
A function is created by the keyword 'def' and is followed by the function name and a list of parameters. For instance:

```python
def powers_of_three(n):
    x = []  # declaring an empty list
    for num in n:
        x.append(num**3)
    return x
```

# Calling the function
numbers = range(4)
print(powers_of_three(numbers))
# [0, 1, 8, 27]
Numpy
Numpy

• NumPy is a core Python package which supports multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.
• Different from Python lists, Numpy array must contains elements of the same type.
• For more information, visit the quick start tutorial. If you are a veteran MATLAB user, Numpy for MATLAB user is also available and is highly recommended, even for non-matlab users.
Numpy Arrays

An array is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called axes. The number of axes is rank.

```python
# First import numpy
import numpy as np  # as np creates an alias
a = np.array([0, 1, 2, 3])  # notice the syntax
b = np.arange(0, 4)  # similar to range, but numpy array
b = np.arange(0, 4).astype(np.float)  # creates an array of floats
print(a, b)  # ([0, 1, 2, 3]), [0., 1., 2., 3.])
a.shape  # returns a tuple; (4,)
a.size  # returns an integer; 4
c = np.array([[[1, 2, 3], [4, 5, 6]]])  # a 2x3 array, rank 2
c.shape  # (2, 3)
c.size  # 6
```
Pre-defined arrays

# We can also create special pre-defined arrays
all_zeros = np.zeros((3,3))  # creates a 3x3 all zeros array
all_ones = np.ones((2,2))    # creates a 2x2 all ones array
all_twos = 2*np.ones((2,2))  # There's a better way to do this...
all_twos = np.full((2,2), 2) # creates a 2x2 all 2 array
identity_matrix = np.eye(3) # creates a 3x3 identity matrix
print(identity_matrix)
# (array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]]))
Reshaping

```python
a = np.arange(0, 12)  # [0, 1 ... , 11]
print(a.reshape((3, 4)))
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
a.reshape((6, -1))  # here the -1 stands for: numpy, please do math for me...
# [[ 0  1]
#  [ 2  3]
#  [ 4  5]
#  [ 6  7]
#  [ 8  9]
#  [10 11]]
a.reshape(2, 3, -1)
# [[[ 0  1]
#  [ 2  3]
#  [ 4  5]]
#  
#  [[ 6  7]
#  [ 8  9]
#  [10 11]]]
a.shape  #(2, 3, 2)
```
Reshaping

# flattening an a multi-dimensional:

```python
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.ravel(x)
print(b)  # prints [1 2 3 4 5 6]
print(b.shape)  # prints '(9L,)'  
```

# np.squeeze removes single-dimensional entries from the shape of an array.

c = np.array([[[0], [1], [2]]])

```python
print(c.shape)  # (1L, 3L, 1L)
print(c)  # [[[0] [1] [2]]]
print(np.squeeze(c).shape)  # (3L,)
print(np.squeeze(c))  # [0 1 2]
```
Indexing

```python
a = np.arange(10)**3  # prints (1000, 1, 8, 27, 64, 125, 216, 343, 512, 729)
print(a)  # prints array([0, 1, 8, 27, 64, 125, 216, 343, 512, 729])
print(a[2])  # prints 8
print(a[2:5])  # prints array([8, 27, 64])
a[6:2] = -1000
# equivalent to a[0:6:2] = -1000; from start to position 6(exclusive), set every 2nd element to -1000
print(a)  # prints array([-1000, 1, -1000, 27, -1000, 125, 216, 343, 512, 729])
a[: : -1]  # reversed 'a'
# prints array([729, 512, 343, 216, 125, -1000, 27, -1000, 1, -1000])
```

```python
a = np.linspace(0, 1, 11)  # from 0 to 1, with 11 steps
print(a)  # prints [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]
idx = np.array([0, 2, 5, 3])
print(idx)  # prints [0 2 5 3]
print(a[idx])  # prints [0. 0.2 0.5 0.3]
```
Indexing.

Multi-dimensional arrays can have one index per axis. These indices are given in a tuple separated by commas.

```python
# The lines below are equivalent
tmp = np.arange(12).reshape(2,3)
print tmp
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
print tmp[1][3]
print (tmp[1])[3]
print tmp[1,3]
print tmp[1][-1] # negative indexing
print tmp[1,-1] # negative indexing
# prints 7
```
Indexing..

# Array from function
def f(x,y):
    return 10*x+y

b = np.fromfunction(f,(5,4),dtype=int)
# creates an array from a function
print(b)
# array([[ 0,  1,  2,  3],
#         [10, 11, 12, 13],
#         [20, 21, 22, 23],
#         [30, 31, 32, 33],
#         [40, 41, 42, 43]])
b[2,3] # '23'
b[0:5, 1] # each row in the second column of b
#array([ 1, 11, 21, 31, 41])
b[:,1] # equivalent to the previous example
#array([ 1, 11, 21, 31, 41])
Indexing...

# b:
# array([[ 0, 1, 2, 3],
# [10, 11, 12, 13],
# [20, 21, 22, 23],
# [30, 31, 32, 33],
# [40, 41, 42, 43]])

b[1:3, : ]  # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])

# Iterating over multidimensional arrays is done with respect to the first axis:
for row in b:
    print(row)
# [0 1 2 3]
# [10 11 12 13] etc..
Indexing with arrays of indices

```python
a = np.arange(12)**2  # the first 12 square numbers
i = np.array([ 1, 1, 3, 8, 5 ])  # an array of indices

print(a[i])  # i can be of different shape than `a`
# array([ 1, 1, 9, 64, 25])

# another possible syntax:
print(a[[0], a[3]])  # [0,9]

j = np.array([ [ 3, 4], [ 9, 7 ] ])
# a bidimensional array of indices
a[j]  # the same shape as j
# array([[ 9, 16],
#         [81, 49]])
```
Boolean indexing

Boolean indexing can be done explicitly:

```python
a = np.arange(5)  # [0, 1, 2, 3, 4];
b = np.array([0,0,1,0,1], dtype=np.bool)  # needs to be the same shape as 'a'
# [False, False, True, False, True]
print(a[b])  # array([2, 4])
# Different shape than 'a'
```
Boolean indexing

Or by logical operations:

```python
a = np.arange(12).reshape(3,4)
# ([[ 0, 1, 2, 3], # [ 4, 5, 6, 7], # [ 8, 9, 10, 11]])
b = a > 4
print(b)  # b is a boolean with a's shape
# array([[False, False, False, False],
#        [False, True, True, True],
#        [True, True, True, True]], dtype=bool)

print(a[b])  # 1d array with the selected elements
array([5, 6, 7, 8, 9, 10, 11])  # This property can be very useful in assignments:
a[b] = 0  # All elements of 'a' higher than 4 become 0
print(a)
# array([[0, 1, 2, 3],
#        [4, 0, 0, 0],
#        [0, 0, 0, 0]])
```
Linear indexing

Sometimes we may want to flatten multi-dimensional array but still use its original coordinates and vice versa. For this we can use the `unravel_index` and `ravel_multi_index` methods.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>1,0</th>
<th>1,1</th>
<th>1,2</th>
<th>2,0</th>
<th>2,1</th>
<th>2,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Linear indexing

```python
a_arr = np.arange(12).reshape(3,-1) #array([[ 0, 1, 2, 3],
# [ 4, 5, 6, 7],
# [ 8, 9, 10, 11]])
a_flat = np.ravel(a_arr) #array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
idx = np.argwhere(a_flat%3==0) # returns indicies for a condition
print a_flat[idx].T # T for transpose - returns row vector
# array([[0, 3, 6, 9]]) # We want the indicies in the dim of a_arr

idx_arr = np.unravel_index(idx, a_arr.shape) # (array([[0],
# [0],
# [1],
# [2]], dtype=int64),
# array([[0],
# [3],
# [2],
# [1]], dtype=int64))
print a_arr[idx_arr].T #[[0 3 6 9]]
```

# The other way around...
#np.ravel_multi_index Converts a tuple of index arrays into an array of flat indices
idx_flat = np.ravel_multi_index(idx_arr, a_arr.shape)
print idx_flat.T # array([[0 3 6 9]], dtype=int64)
Some math

```python
a = np.arange(6).reshape(2,3)
# array([[0, 1, 2],
# [3, 4, 5]])

# same goes for min(), argmin() and minimum() functions
a.max(0)  # maximum element along an axis  # array([3, 4, 5])
a.max(1)  # array([2, 5])
a.argmax(0)  # Returns the indices of the maximum values along an axis.
# array([1, 1, 1])
a.argmax(1)  # array([2, 2])
a.argmax()  # if no axis is given, the index is of the flattened array
# 5

# np.maximum is a bit different - # It compares two arrays and returns a
# new array containing the element-wise maxima:
np.maximum(d[0,:], d[1,:])  # maximum between first and second rows of ‘a’
# array([3, 4, 5])
```
Some math

# a:
array([[0, 1, 2],
       [3, 4, 5]])

np.sum(a)  # Compute sum of all elements; '10'
np.sum(a, axis=0)  # Compute sum of each column; '[4 6]'  
np.sum(a, axis=1)  # Compute sum of each row; '[3 7]'  
np.e  #2.718281828459045
np.exp(1)  #2.718281828459045
np.exp(np.arange(5))  # handle arrays
# array([1. , 2.71828183, 7.3890561 , 20.08553692, 54.59815003])
np.log([1, np.e, np.e**2, 0])  #natural log in base e = lan
# array([ 0., 1., 2., -Inf])
np.log2(8)  #base 2 log # 3
Linear Algebra
Linear Algebra

- Numpy has many built-in linear algebra operations that could be used on numpy arrays.

```python
a = np.arange(1, 5, dtype=float).reshape(2, 2)
# [[ 1.  2.]
#  [ 3.  4.]]
a.T  # matrix transpose
# [[ 1.  3.],
#  [ 2.  4.]]
a.transpose()  # also, matrix transpose, allows for more than 2-dimensions
# [[ 1.  3.],
#  [ 2.  4.]]
```
Linear Algebra

c = np.arange(8).reshape(2,2,-1)  #shape 2x2x2
array([[0, 1],
       [2, 3]],
       [[4, 5],
       [6, 7]])
c.transpose([2,1,0])  #order of axis to transpose
array([[0, 4],
       [2, 6]],
       [[1, 5],
       [3, 7]])
```python
np.arange(8).reshape(2, 2, -1) # shape 2x2x2
```

```python
c = np.arange(8).reshape(2, 2, -1) # shape 2x2x2
# array([[[0, 1],
#         [2, 3]],
#        [[4, 5],
#         [6, 7]]])
c.transpose([0, 2, 1])
# array([[[0, 2],
#         [1, 3]],
#        [[4, 6],
#         [5, 7]]])
```
# a:
# [[ 1. 2.]
#  [ 3. 4.]]

np.linalg.inv(a)  # find the matrix inverse of 'a', usually computationally expensive
# [[-2. , 1. ],
#  [ 1.5, -0.5]]

b = np.full((2,2), 2)
a*b  # element-wise multiply
# array([[2., 4.],
#        [6., 8.]])
Linear Algebra

I = np.eye(2)  # unit 2x2 matrix; "eye" represents "I"
j = np.array([[0.0, -1.0], [1.0, 0.0]])
np.dot (j, j)  # matrix product
# array([[[-1., 0.],
# [ 0., -1.]]])

np.trace(I)  # trace # 2.0
np.diag(a)  # vector of diagonal elements of 'a'
# [1., 4.]

v = np.array([2, 3])
np.linalg.norm(v)  # L2 norm of vector v; # 3.605551275463989
D,V = linalg.eig(a)  # eigenvalues and eigenvectors of a
D,V = np.linalg.eig((a,b))  # eigenvalues and eigenvectors of a, b
Vector stacking:

- It is possible to stack vectors on top of each other.

```python
c = np.ones((1,3))  #array([[1., 1., 1.]])
d = 2*np.ones((1,3))  #array([[2., 2., 2.]])

vertical_stack = np.vstack([c,d])  
#array([[1., 1., 1.],
#        [2., 2., 2.]]).T

horizontal_stack = np.hstack([c,d])
# array([[1., 1., 1., 2., 2., 2.]])

tile(c, (2, 3))  #create 2 by 3 copies of a
#array([[1., 1., 1., 1., 1., 1., 1., 1., 1.],
#       [1., 1., 1., 1., 1., 1., 1., 1., 1.]]).T
```
Probability and statistics
random_arr = np.random.random((2,2))  # creates an array with random values
random_normal = np.random.randn((2,2)) # a 2x2 sampled from N(0,1)
# might output:
# [[-1.25527029, 1.12880546],
#  [-0.78455754, -0.34960907]]

sigma = 2.5
mu = 3
random_normal2 = sigma*np.random.randn(2,2)+mu
# a 2x2 sampled from N(3,2.5)
# [[1.28169047, 1.64080373],
#  [4.76906697, 3.05345461]]

v = np.array([1,1,2,2,2,3,3,4]);
np.random.permutation(v)
# [4, 1, 2, 3, 2, 2, 1, 2, 3]

np.median(a) # 2.5 np.average(a) # 2.5
np.std(a) # 1.1180339887
np.var(a) # 1.25
# Sample from an array with corresponding probabilities array

# Generate a non-uniform random sample from np.arange(5) of size 3:
np.random.choice(np.arange(5), 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
# might output array([2, 3, 0])

# replacing np.arange(5) with 5 yield the same result
np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
# might output array([2, 0, 3])

# replacing the replace=True allows for sampling the same value
np.random.choice(5, 3, replace=True, p=[0.1, 0, 0.3, 0.6, 0])
# might output array([3, 3, 0])