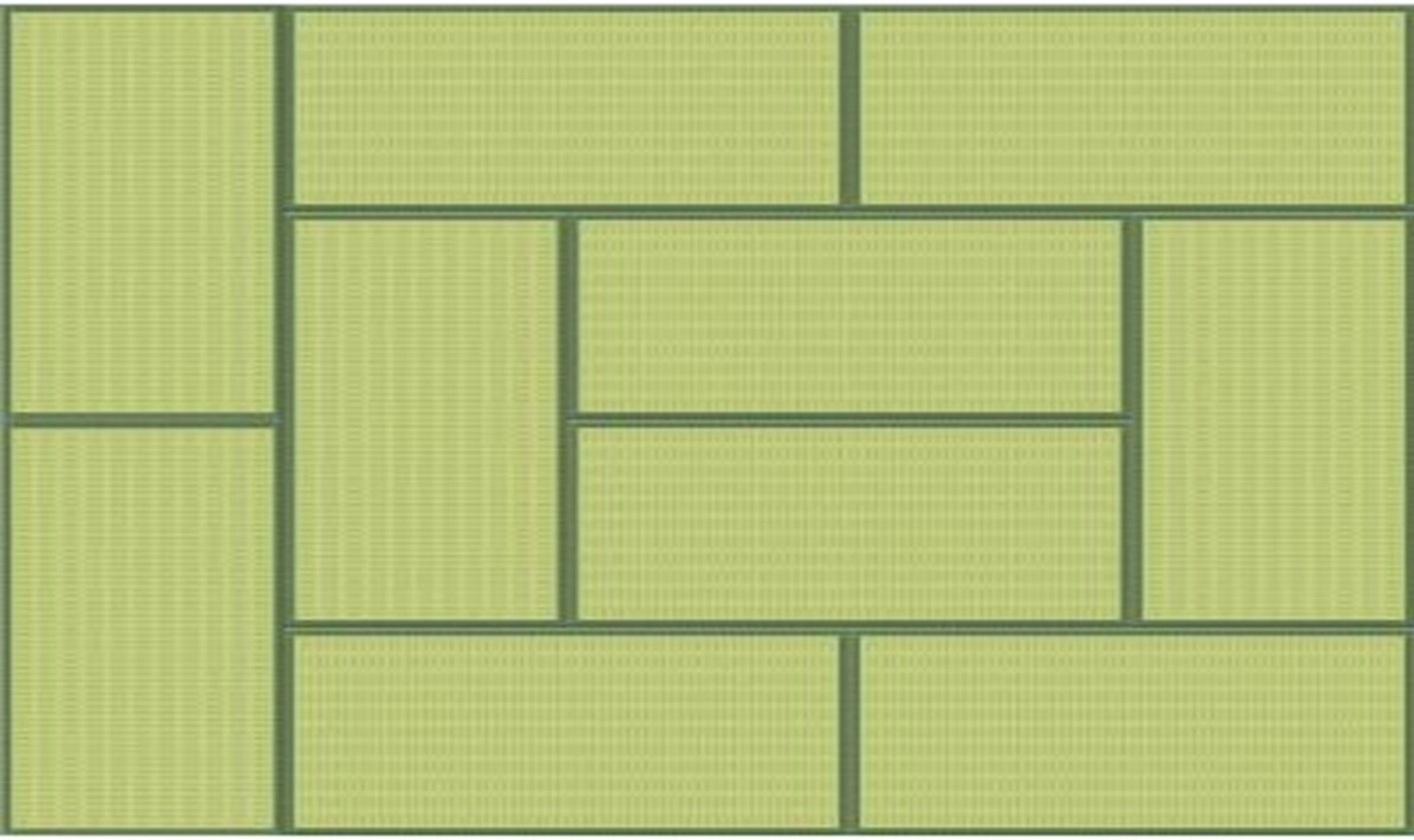
Series
10k copies
have been
sold

Python

Numpy Exercises

Skyrocket your Python skill



Joshua K. Cage

Python Numpy Exercises
-Skyrocket your Python skill
Joshua K. Cage

Introduction.

Target audience

Thank you for picking up this book. This book is a practical introduction to "Numpy" for first-time Python users. The goal of this book is to give you the freedom to write code that takes full advantage of the capabilities of Numpy and Python by walking you through the 101 questions while you are writing a real-world program in Python.

The following readers are envisioned

1) You've learned the basic Python grammar, so you want to take the next step

2) If you want to write fast-running, concise Python programs.

3) If you're also a little curious about the mechanics behind deep learning and machine learning.

4) Those who get defensive when they hear the words vector and matrix.

5) Those who want to handle large scale data.

6) Those who want to study a little bit every day

7) If you have started to solve the numpy 100 exercises, but are frustrated

This book starts with "import numpy as np" and lays the foundation for doing things like linear algebra and basic statistics in machine learning.

Programming is often said to be "better to get used to it than to learn it," but if you don't take the time to build an environment to get used to it and get to the point where you can't get to the point, there is no point. This book includes links to the executable Google Colaboratory source code, so you can actually run the code and modify it as you solve problems without the hassle of setting up an environment.

However, explanations are omitted for questions that may not require explanation if you read the source material. However, explanations are omitted for problems that may not be necessary if you read the source material. If you find something difficult to understand, please let us know by email using feedback.
We also tweet about supplements and corrections to the book on Twitter (@JoshuaKCage1).

Exemption from responsibility

The information contained in this document is for informational purposes only. Therefore, the use of this book is always at the reader's own risk and discretion. The use of the Google Colaboratory described in this book is at the reader's own risk after reviewing Google's Terms of Service and Privacy Policy. In no event shall the reader be liable for any consequential, incidental, or lost profits or other indirect damages, whether foreseen or foreseeable, arising out of or in connection with the use of the source code accompanying this book or the Google Colaboratory service.

Please use this book if you agree to the above precautions. Please note that the author will not be able to respond to any inquiries you make without reading these notes. Please be aware that you may not be able to contact us if you do not read these terms and conditions.

About Trademarks and Registered Trademarks

All product names appearing in this manual are generally registered trademarks or trademarks of the respective companies. ™, ® and other marks may be omitted from the text.

Feedback

While the utmost care has been taken in the writing of this book, you may notice errors, inaccuracies, misleading or confusing language, or simple typographical errors and mistakes. In such cases, we would appreciate your feedback to the following address so that we can improve future editions. Suggestions for future revisions are also welcome. The contact information is below.

Joshua K. Cage

joshua.k.cage@gmail.com

Jupyter Notebook

The Jupyter Notebook, which allows you to run the code described in this book, is now available on Google Colaboratory. You can access it from the following link, so please refer to it when you read this book (Chrome is recommended*) .

https://colab.research.google.com/drive/13_MEoP-TDQrYVhTbMESufyyuw_YRlADu#scrollTo=1KT7lPjN_s31

*Google Colaboratory is a service of Google and may be terminated by the author without notice.

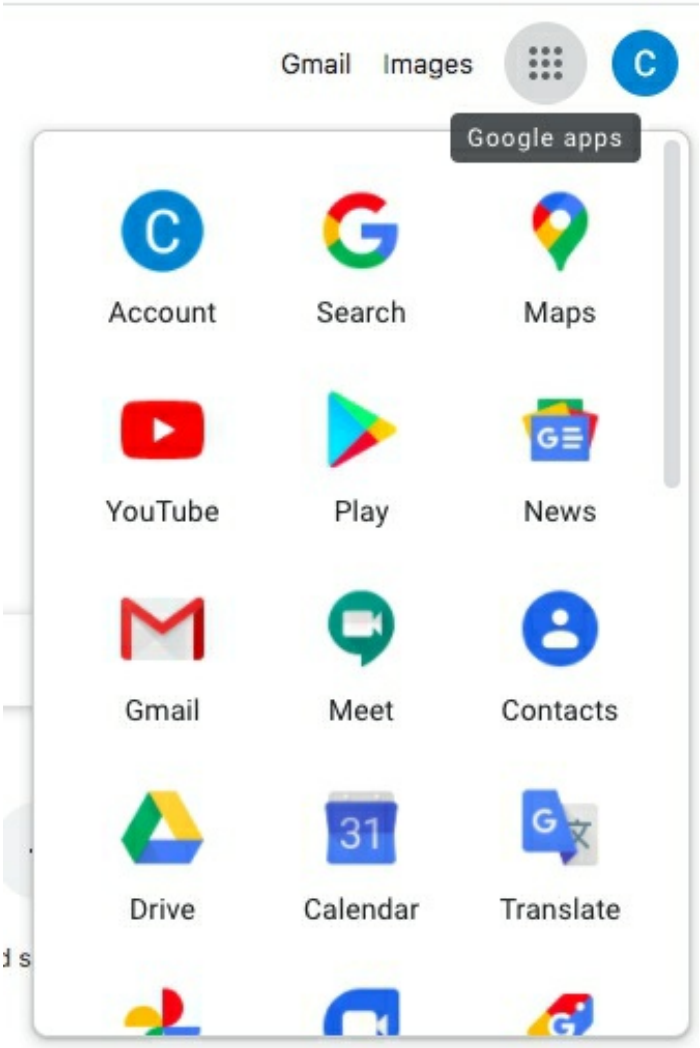
Setting up a Python environment - How to useColab

In this book, I have published the steps to set up a GPU environment on Google Colaboratory, as well as source code with explanations for this problem. Basically, this book is designed to work only by executing the cells from the top, so that you don't have to spend time on building an extra environment.

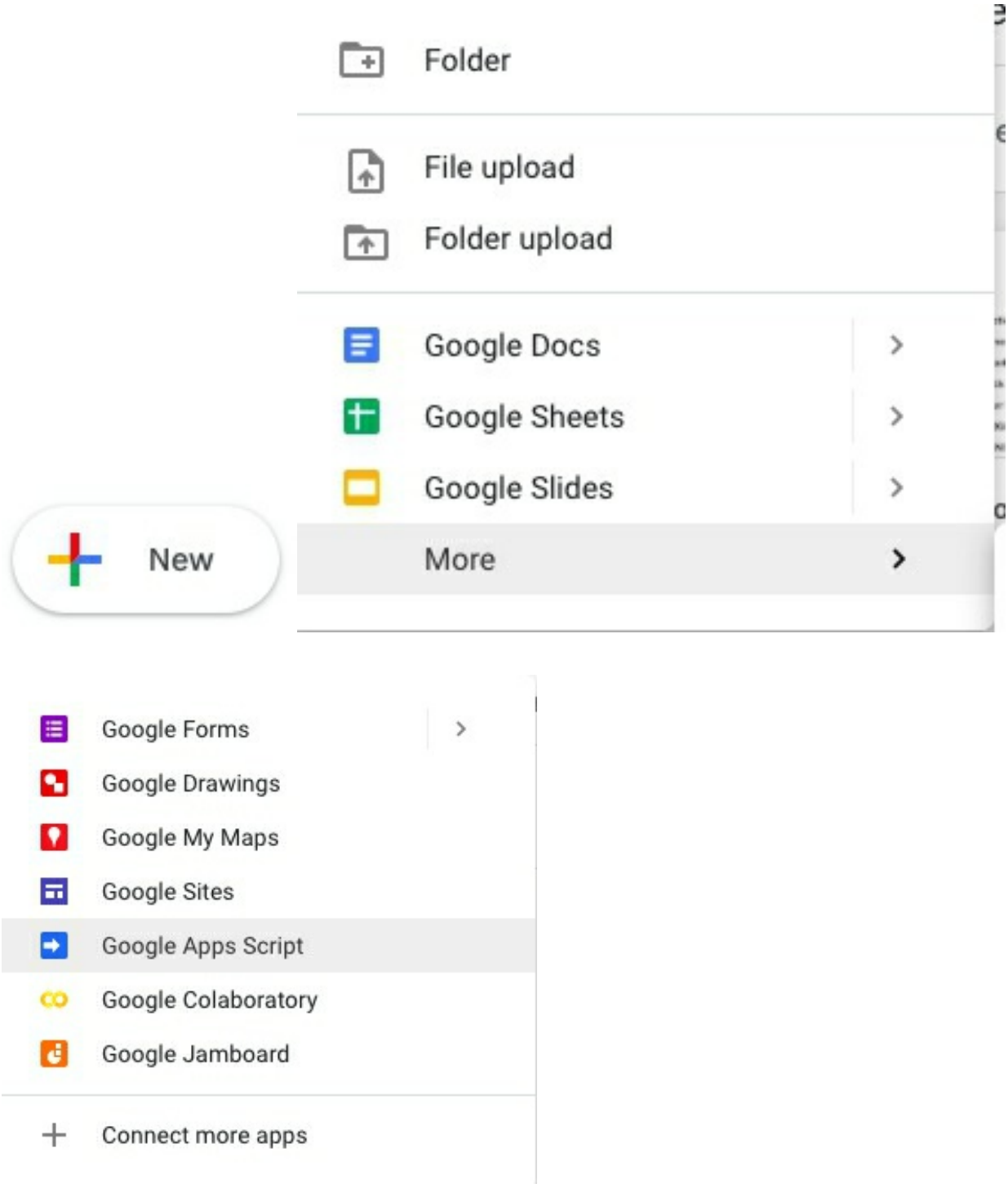
If you don't have a GMAIL account, you will need to create one by clicking on the link here. The following explanation goes on assuming that you already have a gmail account.

How to Setup Colab

(1) When you access GMAIL, in the upper right corner of the screen you will see a Bento Menu with nine squares, click on that and then click on the "Drive" icon.

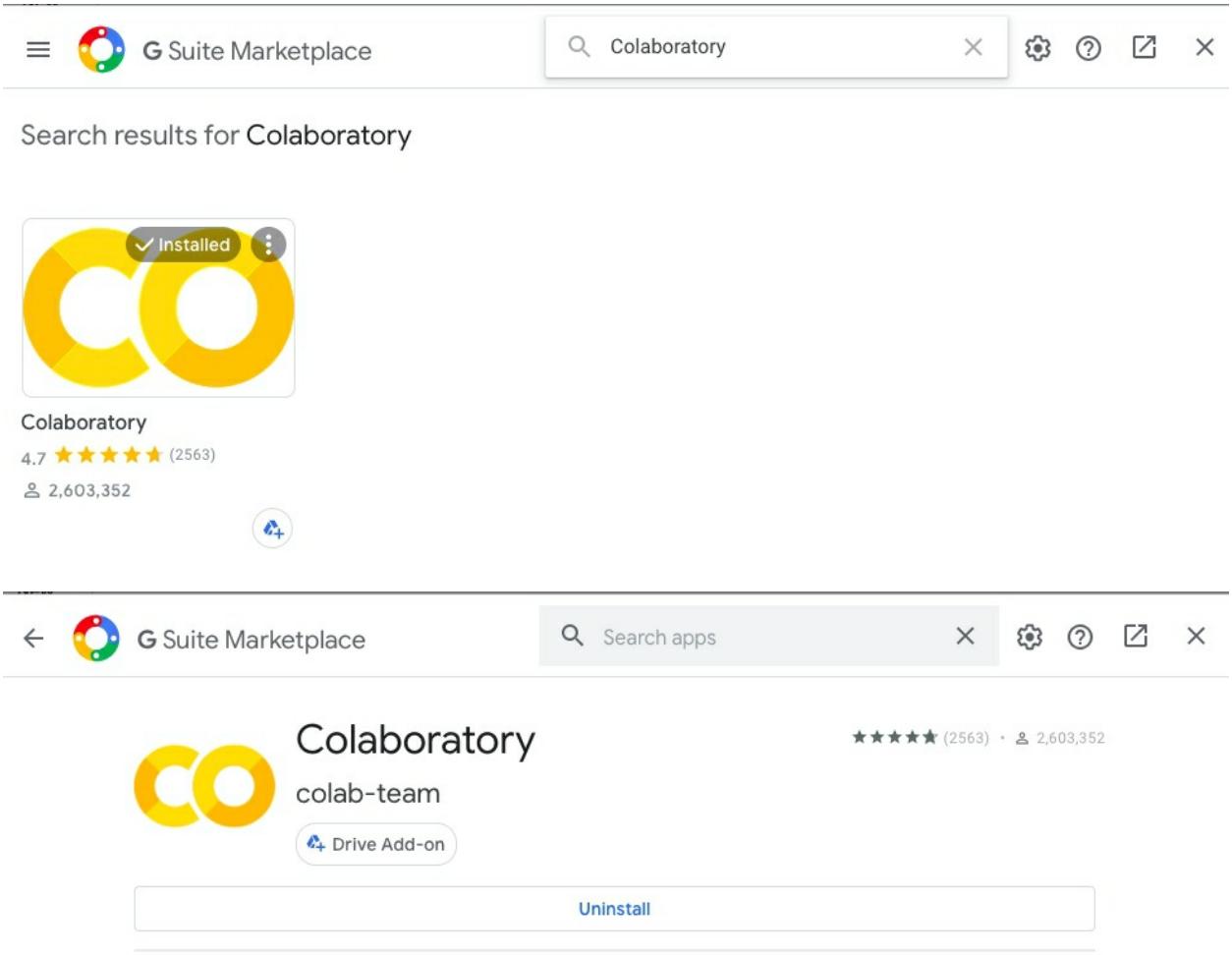


(2) Press the "+ New" button at the bottom of the drive and select "More >" from the menu, then click "Google Colaboratory" if it exists, otherwise choose "Connect more apps".



(3) When "G Suite Marketplace" is displayed, click on the magnifying glass mark, and in the text box to search in the app, type "Colaboratory". Please click the "+" button at the bottom right of the logo, and then click the "Install" button on the screen that appears.

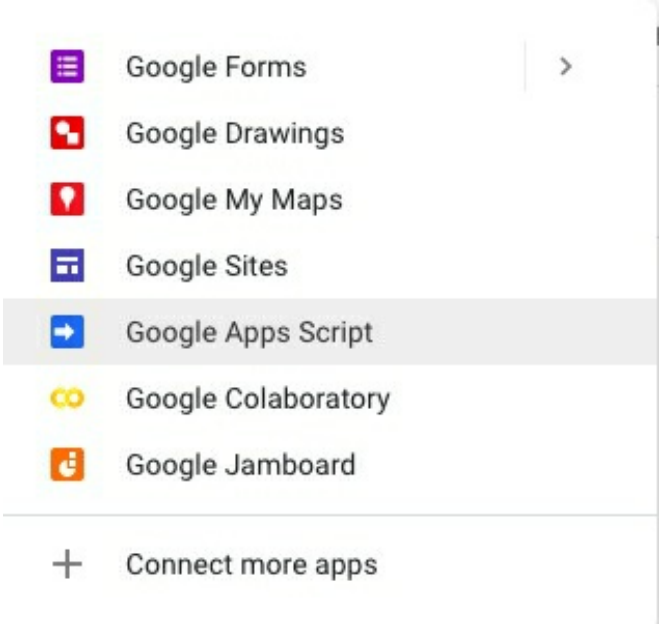
Please click the "Install" button on the screen displayed at the bottom right of the logo.



(4) You may be asked to log in again, please continue. When the screen of "Google Colaboratory is now connected to Google Drive. When the screen of "Google Colaboratory has been connected to Google Drive" appears, check the box of "Make Google Colaboratory the default application" and click the "OK" button. A modal window that says "Colaboratory has been installed. When you see the modal window "You have installed Colaboratory", you can use Colab. Now, when you upload a file with the Colab extension (.ipynb file) to Google Drive, it should open in Colab by default.

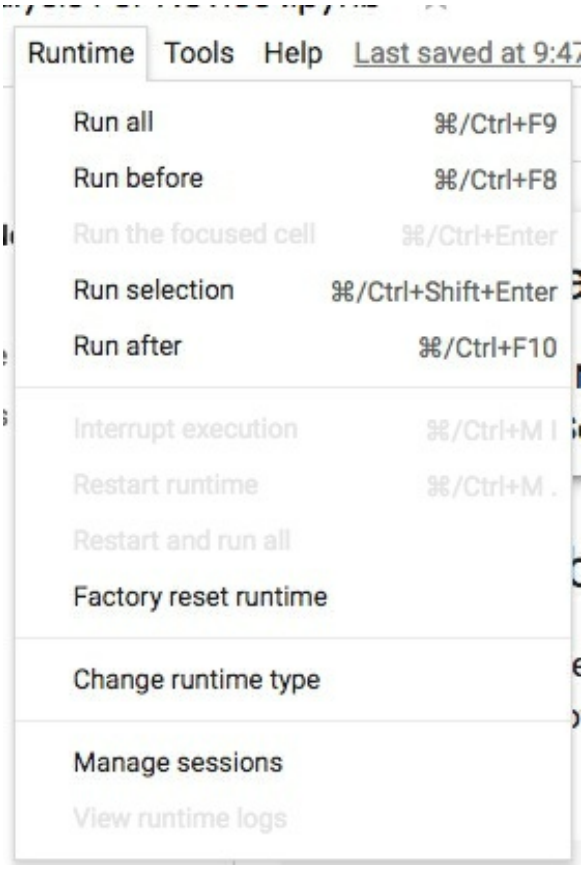
(5) Close the modal window and once again, click the "New +" button and

select the "Other >" app. Now you can select "Google Colaboratory".



(6) When you select Google Colaboratory, the following screen will open up, but by default, Colab is in CPU-using mode, which means it will take longer to run deep learning. So, go to the "Runtime" menu, click "Change runtime type" and select "GPU" in the "Hardware Accelerator" section and click the Save button.

It is also possible to use TPU here, but it is a bit difficult to get the performance out of it, and for most applications there is not much difference in execution speed between GPU and GPU, so we will use "GPU" in this manual.



Notebook settings

Hardware accelerator

GPU

To get the most out of Colab, avoid using a GPU unless you need one. Learn more

Omit code cell output when saving this notebook

CANCEL SAVE

(7) To make sure the GPU is available, copy the following code into a cell and run it. You can execute it by pressing the play button on the left side of the cell, or you can use the shortcut "Shift + Enter". If you see "device_type: "GPU" in the execution result, it means that the GPU is recognized.

```
from tensorflow.python.client import
device_libdevice_lib.list_local_devices()
```

Output:

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 8604083664829407890, name: "/device:XLA_CPU:0"
device_type: "XLA_CPU"
memory_limit: 17179869184
locality {
}
incarnation: 18180926124650645506
physical_device_desc: "device: XLA_CPU device", name:
"/device:XLA_GPU:0"
device_type: "XLA_GPU"
memory_limit: 17179869184
locality {
}
incarnation: 18355618728471253196
physical_device_desc: "device: XLA_GPU device", name:
"/device:GPU:0"
device_type: "GPU"
memory_limit: 11146783616
locality {
    bus_id: 1
    links {
    }
```

```
}
incarnation: 18112086373768308297
physical_device_desc: "device: 0, name: Tesla K80, pci bus id:
0000:00:04.0, compute capability: 3.7"]
```

Chapter 1: Numpy is fast, cheap and good

numpy is intuitive, concise, and fast to write, and it's a great thing that provides basic linear algebra methods as standard. What it lacks in the standard Python library, it makes up for in numpy. Enjoy the world of (excellent design).

01: IMPORT Numpy with the name np.

```
import numpy as np
np
```

Output:

```
<module 'numpy' from '/usr/local/lib/python3.6/dist-packages/numpy/__init__.py'>
```

It is possible to import a module with the syntax "import library formal name as shorthanded system" and call it as "shorthanded system. It is possible to import a module with the syntax "shorthanded" and call something like "shorthanded function name()". For example, we can import numpy as np and then use np.sqrt(2) to find √2.

```
np.sqrt(2)
```

Output:

```
1.4142135623730951
```

02: Check the version of Numpy.

```
print(np.__version__)
```

Output:

```
1.18.5
```

The version of Numpy that was installed by default on the Colab environment the author is currently testing was 1.18.5.

03: Identify the type of BLAS that Numpy is using.

```
print(np.__config__.show())
```

Output:

```
blas_mkl_info:
  NOT AVAILABLE
blis_info:
  NOT AVAILABLE
openblas_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
blas_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
lapack_mkl_info:
  NOT AVAILABLE
openblas_lapack_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
lapack_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
None
```

```
print(np.show_config())
```

Output:

```
blas_mkl_info:
  NOT AVAILABLE
blis_info:
  NOT AVAILABLE
openblas_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
blas_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
lapack_mkl_info:
  NOT AVAILABLE
openblas_lapack_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
lapack_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
None
```

np.show_config() and np.__config__.show() print out the same value and you can see what the bound linear algebra library is. numpy is fast even though it's Python because it's running inside BLAS(Basic Linear Algebra Subprograms), a linear algebra arithmetic library. Depending on which CPU you are running on, there are different types of BLASs bound to it, the most famous being Open BLUS/Intel MKL/ATLAS.
In Colab, Open BLUS was bound; Intel MKL drinks and runs on Intel CPUs.

I'm going to use the "lshw" command to see what Colab's CPU is using. First, we will install the lshw command by using the "apt-get install" command.

```
!apt-get install lshw
```

Output:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnvidia-common-440
Use 'apt autoremove' to remove it.
The following additional packages will be installed:
  libpci3 pciutils usbutils
The following NEW packages will be installed:
  libpci3 lshw pciutils usbutils
0 upgraded, 4 newly installed, 0 to remove and 35 not upgraded.
Need to get 721 kB of archives.
After this operation, 2,870 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 libpci3 amd64 1:3.5.2-1ubuntu1.1 [24.1 kB]
Get:2 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 lshw amd64 02.18-0.1ubuntu6.18.04.1 [231 kB]
Get:3 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 pciutils amd64 1:3.5.2-1ubuntu1.1 [257 kB]
Get:4 http://archive.ubuntu.com/ubuntu bionic/main amd64 usbutils amd64 1:007-4build1 [208 kB]
Fetched 721 kB in 1s (527 kB/s)
Selecting previously unselected package libpci3:amd64.
(Reading database ... 144487 files and directories currently installed.)
Preparing to unpack .../libpci3_1%3a3.5.2-1ubuntu1.1_amd64.deb ...
Unpacking libpci3:amd64 (1:3.5.2-1ubuntu1.1) ...
Selecting previously unselected package lshw.
Preparing to unpack .../lshw_02.18-0.1ubuntu6.18.04.1_amd64.deb ...
Unpacking lshw (02.18-0.1ubuntu6.18.04.1) ...
Selecting previously unselected package pciutils.
Preparing to unpack .../pciutils_1%3a3.5.2-1ubuntu1.1_amd64.deb ...
Unpacking pciutils (1:3.5.2-1ubuntu1.1) ...
Selecting previously unselected package usbutils.
Preparing to unpack .../usbutils_1%3a007-4build1_amd64.deb ...
Unpacking usbutils (1:007-4build1) ...
Setting up lshw (02.18-0.1ubuntu6.18.04.1) ...
Setting up usbutils (1:007-4build1) ...
Setting up libpci3:amd64 (1:3.5.2-1ubuntu1.1) ...
Setting up pciutils (1:3.5.2-1ubuntu1.1) ...
Processing triggers for libc-bin (2.27-3ubuntu1) ...
/sbin/ldconfig.real: /usr/local/lib/python3.6/dist-packages/ideep4py/lib/libmkldnn.so.0 is not a symbolic link

Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
```

```
!lshw |grep -A 1 -i "cpu"
```

Output:

```
   *-cpu
        product: Intel(R) Xeon(R) CPU @ 2.20GHz
        vendor: Intel Corp.
--
        bus info: cpu@0
        width: 64 bits
        capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp x86-
64 constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni
pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c
rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp fsgsbase
tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx smap xsaveopt arat
md_clear arch_capabilities
   *-pci
```

It seems that Colab dynamically selects and allocates CPU resources at runtime, and in the above example, Intel Xeon CPUs are used, but depending on the runtime, AMD may be allocated.

04: Generate a one-dimensional vector with all elements of size 10 being zero.

```
a = np.zeros(10, dtype=np.int)
print(a)
```

Output:

```
[0 0 0 0 0 0 0 0 0 0]
```

You can use np.zeros() to create a numpy array with all the elements zero. You can also specify the type of np.int, np.float16/32/64, etc. by using dtype as an argument.

05: Generate a one-dimensional list of all elements of size 10 that are zero.

```
b = []
for i in range(10):
  b.append(0)
print(b)
```

Output:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Make an assignment to the list in 06:04 and 05 so that the index is a value and the address of each element is displayed by id().

```
for i in range(len(a)):
  a[i] = i
  b[i] = i
  print(f"a[{i}]:{a[i]},{id(a[i])}, b[{i}]:{b[i]},{id(b[i])}")
```

Output:

```
a[0]:0.0,139820194057336, b[0]:0,10914464
a[1]:1.0,139820194057336, b[1]:1,10914496
a[2]:2.0,139820194057336, b[2]:2,10914528
a[3]:3.0,139820194057336, b[3]:3,10914560
a[4]:4.0,139820194057336, b[4]:4,10914592
a[5]:5.0,139820194057336, b[5]:5,10914624
a[6]:6.0,139820194057336, b[6]:6,10914656
a[7]:7.0,139820194057336, b[7]:7,10914688
a[8]:8.0,139820194057336, b[8]:8,10914720
a[9]:9.0,139820194057336, b[9]:9,10914752
```

07: Get Numpy 's np.dot documentation from the command line

```
!python -c "import numpy as np; np.info(np.dot)"
```

Output:

```
 dot(*args, **kwargs)

dot(a, b, out=None)

Dot product of two arrays. Specifically,

- If both `a` and `b` are 1-D arrays, it is inner product of vectors
  (without complex conjugation).

- If both `a` and `b` are 2-D arrays, it is matrix multiplication,
  but using :func:`matmul` or ``a @ b`` is preferred.

- If either `a` or `b` is 0-D (scalar), it is equivalent to :func:`multiply`
  and using ``numpy.multiply(a, b)`` or ``a * b`` is preferred.

- If `a` is an N-D array and `b` is a 1-D array, it is a sum product over
  the last axis of `a` and `b`.

- If `a` is an N-D array and `b` is an M-D array (where ``M>=2``), it is a
  sum product over the last axis of `a` and the second-to-last axis of `b`::

      dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

Parameters
----------
a : array_like
      First argument.
b : array_like
      Second argument.
out : ndarray, optional
```

*Output is omitted for brevity

08: After defining a vector of all elements 0 of size 10, set the fifth element to 1.

```
a = np.zeros(10)
a[4] = 1
a
```

Output:

```
array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.])
```

09: Measure the execution speed of Numpy 's ndarray and Python list to generate a vector with elements that increase by 1 from 100 to 999999, respectively.

```
import numpy as np

# Numpy
%timeit np.arange(100, 10000000)

# Python 
def gen1(n, m):
  return [i for i in range(n, m, 1)]

%timeit gen_list(100, 10000000)

# Python 
def gen2(n, m):
  res = []
  for i in range(n, m, 1):
    res.append(i)
  return res

%timeit gen_nlist(100, 10000000)
```

Output:

```
100 loops, best of 3: 13.1 ms per loop
1 loop, best of 3: 612 ms per loop
1 loop, best of 3: 1e+03 ms per loop
```

10: Generate a vector in reverse order.

```
a = np.arange(10, 50)
a[::-1]
```

Output:

```
array([49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33,
       32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16,
       15, 14, 13, 12, 11, 10])
```

(Chapter 2) Numpy is flexible.

11: Generate a 3 x 3 unit matrix.

```
np.eye(3, 3)
```

Output:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

By the way, if you write in Python without thinking about it, as shown below, the execution time is 10 lines even though the number of elements is more than 10 times slower when the number of elements is large.

```
l1 = []
for i in range(3):
  l2 = []
  for j in range(3):
    if i == j:
      l2.append(1)
    else:
      l2.append(0)
  l1.append(l2)
l1
```

Output:

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

12: Define a random number array A with a 512 x 768 shape, 12 random number arrays B of 768 x 768 (B1 - B12), and a random number array C of 768 x 2, and output the result of dot product operations on all of them in order.

```
import numpy as np
A = np.random.random((512, 768))
B = []
for _ in range(12):
  B.append(np.random.random((768, 768)))
C = np.random.random((768, 2))

TMP = np.zeros((512, 768))
TMP = A@B[0]
for i in range(1, 12, 1):
  TMP = TMP @B[i]

print(TMP@C)
```

Output:

```
[[2.02934344e+33 1.95493496e+33]
 [2.06480646e+33 1.98909768e+33]
 [2.05071390e+33 1.97552185e+33]
 ...
 [2.02272562e+33 1.94855979e+33]
 [1.92827708e+33 1.85757432e+33]
 [2.02325180e+33 1.94906668e+33]]
```

13: Create a matrix of 10 x 10 SHAPE random numbers and find the maximum

and minimum values.

```
import numpy as np
a = np.random.random((10, 10))
print(a.shape)
print(np.max(a))
print(np.min(a))
```

Output:

```
(10, 10)
0.994408691305414
0.02106660759118073
```

14: Generate an array of random numbers of size 30 and calculate the average value.

```
np.mean(np.random.random(30))
```

Output:

```
0.42715189811867893
```

15: Generate a two-dimensional array that is surrounded by 1s on all sides and 0s in the middle.

```
a = np.zeros((8, 8))
a[0, :] = 1
a[:, 0] = 1
a[-1, :] = 1
a[:, -1] = 1
a
```

Output:

```
array([[1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 0., 0., 0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])
```

16: Use Numpy to describe the initial state of Othello.

```
# reversi
a = np.full((8,8), 0)
a[4, 4] = 0
a[3, 3] = 0
a[3, 4] = 1
a[4, 3] = 1
a
```

Output:

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

17: Surround a 5 x 5 matrix with all the elements 0 by 1.

```
a = np.ones((5,5))
a = np.pad(a, pad_width=1)
print(a)
```

Output:

```
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
```

18: Surround a 7 x 7 matrix with all the elements 0 by 1. But write it in a different way than in No. 17.

```
Z = np.ones((7,7))
Z[:, [0, -1]] = 0
Z[[0, -1], :] = 0
print(Z)
```

Output:

```
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
```

19: Create a 5 x 5 0 matrix and arrange 1,2,3,4,5 on the diagonal.

```
import numpy as np

a = np.zeros((5,5), dtype='int64')
for i in range(len(a)):
  a[i, i] = i+1
print(a)
```

Output:

```
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

different solution

```
# You can also create a diagonal matrix from a one-dimensional array
# with its diagonal components
a = np.diag(np.arange(1, 6), k=0)
print(a)
```

Output:

```
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

(Chapter 3) Welcome to the depths of Tensor

Create a 20:8 x 8 matrix and represent the checkerboard pattern by 01.

```
a = np.zeros((8, 8), dtype=int)
for i in range(8):
  for j in range(8):
    if i %2 == 0 and j %2 == 0:
      a[i, j] = 1
    elif i %2 == 1 and j %2 == 1:
      a[i, j] = 1
    else:
      a[i, j] = 0
print(a)
```

Output:

```
[[1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]]
```

different solution

```
a = np.zeros((8,8),dtype=int)
a[1::2,::2] = 1
a[::2,1::2] = 1
print(a)
```

Output:

```
[[0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]]
```

21: Find the index (x,y,z) of the 100th element when there is an array of the form (6,6,6).

```
print(np.unravel_index(99, (6,6,6)))
```

Output:

```
(2, 4, 3)
```

22: Create an 8x8 checkerboard using the tile function.

```
print(np.tile(np.array([[0,1],[1,0]]), (4,4)))
```

Output:

```
[[0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]]
```

23: Normalize a 5 x 5 random number array to fit within 0-1.

```
a = np.random.random((5, 5))
print((a - np.min(a))/(np.max(a) - np.min(a)))
```

Output:

```
array([[0.62903013, 0.93971502, 0.59179131, 0.12931082, 0.69792704],
       [0.25864949, 0.26712684, 0.77169336, 0.6045717 , 0.86074969],
       [0.00405325, 0.93687097, 0.61919117, 0.45140912, 0.30380776],
       [0.21182072, 0.02065823, 0.53053422, 0.43184797, 0.80960432],
       [0.42350389, 0.35129564, 0.19431125, 0.40187998, 0.75223023]])
```

24: Create your own dtype that represents an integer on the xy coordinate axis.

```
import numpy as np
mytype = np.dtype([('x', np.int), ('y', np.int)])
a = np.array([(-1, 5), (3, 8), (4, 4)], dtype=mytype)
a
```

Output:

```
array([(-1, 5), ( 3, 8), ( 4, 4)], dtype=[('x', '<i8'), ('y', '<i8')])
```

25: Create your own dtype for RGBA (the three primary colors plus transparency).

```
color = np.dtype([('r', np.ubyte), ('g', np.ubyte), ('b', np.ubyte),
('a', np.ubyte)])
test = np.array([(255, 255, 255, 0.5)], dtype=color)
test
```

Output:

```
array([(255, 255, 255, 0)],
      dtype=[('r', 'u1'), ('g', 'u1'), ('b', 'u1'), ('a', 'u1')])
```

26: Find the matrix product of a 2 x 2 matrix and a 2 x 2 matrix.

```
import numpy as np
A = np.array([[1,2],[3,4]])
B = np.array([[1,2],[3,4]])
A@B
```

Output:

```
array([[ 7, 10],
       [15, 22]])
```

Suppose we have,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, B = \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

AB can be derived as follows:

$$AB = \begin{pmatrix} ap + br & aq + bs \\ cp + dr & cq + ds \end{pmatrix}$$

In this case,

$$AB = \begin{pmatrix} 1 \times 1 + 2 \times 3 & 1 \times 2 + 2 \times 4 \\ 3 \times 1 + 4 \times 3 & 3 \times 2 + 4 \times 4 \end{pmatrix}$$

27: Generate a sign-reversed array of 1D arrays whose elements are between 3 and 8.

Python standard library version

```
def negate38(a):
  l = []
  for i in a:
    if a[i] >= 3 and a[i] <= 8:
      l.append(-a[i])
    else:
      l.append(a[i])
  return l

a = np.arange(10)
%timeit negate38(a)
print(negate38(a))
```

Output:

```
The slowest run took 25.07 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 3: 11.5 µs per loop
[0, 1, 2, -3, -4, -5, -6, -7, -8, 9]
```

Numpy Edition

```
Z = np.arange(10)
%timeit Z[(3 <= Z) & (Z <= 8)] *= -1
Z
```

Output:

```
The slowest run took 22.44 times longer than the fastest. This could mean that an
intermediate result is being cached.
100000 loops, best of 3: 3.48 µs per loop
array([ 0,  1,  2, -3, -4, -5, -6, -7, -8,  9])
```

28: Compare the results when sum() is executed without importing numpy and when sum() is executed after numpy has been *imported.

```
print(sum(range(5),-1))
from numpy import *
print(sum(range(5),-1))
```

Output:

```
9
10
```

standard library in python

```
sum(iterable[, start])
```

numpy

```
import numpy as np
np.sum(iterable[, axis])
```

The second argument of the sum() method of the standard Python function represents start, so it returns 9, the sum of -1, 0, 1, 2, 2, 3, and 4, while the second argument of the Numpy sum() method represents axis. -The second argument of Numpy's sum() method represents the axis. *It is important to use import numpy as np instead of importing.

29: Which of the following expressions in vector Z is problematic?

```
Z**Z
2 << Z >> 2
Z <- Z
1j*Z
Z/1/1
Z<Z>Z
```

]

```
Z = np.array([1,2,3,4,5])
# raising a number to a power
print(Z**Z)
# Bitwise operations [2 << 1 >> 2]→2 are shifted left by 1 bit in
 binary (4) and then shifted right by 2 bits to it (1).
print(2 << Z >> 2)
# True or false output of Z < -
Z (True if negative, False if positive)
print(Z <- Z)
#Multiplication of complex number j
print(1j*Z)
#divisor by two returns
print(Z/1/1)
#It's OK if it's a scalar, but not if it's an integer vector, so
it's legal except for the following.
print(Z<Z>Z)
```

(Chapter 4) Working with Numbers in Numpy

30: Show the result of the following equation.

```
np.array(0) / np.array(0)
np.array(0) // np.array(0)
np.array([np.nan]).astype(int).astype(float)
```

```
#If you divide by 0, a warning is output.
RuntimeWarning: invalid value encountered in true_divide
print(np.array(0) / np.array(0))
#If you divide by 0 (to get the quotient only), a warning is outp
ut.RuntimeWarning: divide by zero encountered in floor_divide
print(np.array(0) // np.array(0))
# Converted to a minimum number of int types
print(np.array([np.nan]).astype(int).astype(float))
```

Output:

```
nan
0
[-9.22337204e+18]
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: invali
value encountered in true_divide

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:4: RuntimeWarning: divide
zero encountered in floor_divide
  after removing the cwd from sys.path.
```

31: Round up to the nearest whole number in a float array.

```
import numpy as np


a = np.array([-1.5, -2.6, 0.0, 4.2, 3.14])
# Emphasis on readability
print(np.where(a>0, np.ceil(a), np.floor(a)))
# Computer efficiency oriented
print(np.copysign(np.ceil(np.abs(a)), a))
```

Output:

```
[-2. -3.  0.  5.  4.]
[-2. -3.  0.  5.  4.]
```

32: Find the common value of the two sequences.

```
import numpy as np
A1 = np.array([1,5, 3, 4, 5])
A2 = np.array([7, 2, 3, 4, 6])
# python standard library
print(np.array(list(set(A1) & set(A2))))
# Use numpy
print(np.intersect1d(A1,A2))
```

Output:

```
[3 4]
[3 4]
```

33: Perform division by zero in the mode of ignoring

all numpy warnings.

```
# Ignore All Mode ON
defaults = np.seterr(all="ignore")
Z = np.ones(1) / 0

# go back to the start
_ = np.seterr(**defaults)

# Use the context manager (this is more secure)
with np.errstate(all="ignore"):
    np.array([1, 2, 3, 4, 5])/ 0

np.array([1, 2, 3, 4, 5])/ 0
```

Output:

```

```

By the way, if you run it in normal mode, you will get the following zero division warning.

```
np.array([1, 2, 3, 4, 5])/ 0
```

Output:

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: RuntimeWarning: divide
zero encountered in true_divide
  """Entry point for launching an IPython kernel.
array([inf, inf, inf, inf, inf])
```

34: What is the true value of the following equation?

```
np.sqrt(-1) == np.emath.sqrt(-1)
```

```
np.emath.sqrt(-1)
```

Output:

```
1j
```

```
np.sqrt(-1)
```

Output:

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: RuntimeWarning: invali
value encountered in sqrt
  """Entry point for launching an IPython kernel.
nan
```

```
#Mathematical functions with automatic domain
np.sqrt(-1) == np.emath.sqrt(-1)
```

Output:

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: invali
value encountered in sqrt

False
```

The question of whether np.emath can handle imaginary numbers. np.emath can also get the square root of -1 with the mathematical functions with automatic domain module, but not with np.sqrt.

35: Get yesterday, today and tomorrow's dates.

♦Numpy

```
import numpy as np
yesterday = np.datetime64('today') - np.timedelta64(1)
today     = np.datetime64('today')
tomorrow  = np.datetime64('today') + np.timedelta64(1)
print(yesterday)
print(today)
print(tomorrow)
```

Output:

```
2020-08-29
2020-08-30
2020-08-31
```

The Python standard library

```
from datetime import date
from datetime import timedelta
print(date.today()-timedelta(1))
print(date.today())
print(date.today()+timedelta(1))
```

Output:

```
2020-08-29
2020-08-30
2020-08-31
```

36: Get all dates for September 2020.

```
import numpy as np
print(np.arange('2020-09', '2020-10', dtype='datetime64[D]'))
```

Output:

```
['2020-09-01' '2020-09-02' '2020-09-03' '2020-09-04' '2020-09-05'
 '2020-09-06' '2020-09-07' '2020-09-08' '2020-09-09' '2020-09-10'
 '2020-09-11' '2020-09-12' '2020-09-13' '2020-09-14' '2020-09-15'
 '2020-09-16' '2020-09-17' '2020-09-18' '2020-09-19' '2020-09-20'
 '2020-09-21' '2020-09-22' '2020-09-23' '2020-09-24' '2020-09-25'
 '2020-09-26' '2020-09-27' '2020-09-28' '2020-09-29' '2020-09-30']
```

```
import calendar
y = 2020
m = 9
start, end = calendar.monthrange(y, m)
print([f"{y}-{str(m).zfill(2)}-{str(d).zfill(2)}" for d in range(start
end + 1)])
```

Output:

```
['2020-09-01', '2020-09-02', '2020-09-03', '2020-09-04', '2020-09-05', '2020-09-06', '202
09-07', '2020-09-08', '2020-09-09', '2020-09-10', '2020-09-11', '2020-09-12', '2020-09-1
'2020-09-14', '2020-09-15', '2020-09-16', '2020-09-17', '2020-09-18', '2020-09-19', '202
09-20', '2020-09-21', '2020-09-22', '2020-09-23', '2020-09-24', '2020-09-25', '2020-09-2
'2020-09-27', '2020-09-28', '2020-09-29', '2020-09-30']
```

37: If two arrays A and B have A=[1.0, 2.0] and B=[3.0, 4.0] respectively, calculate (-(A+B)xB+1) / 2 with and without memory copy, respectively.

```
A = np.array([1.0, 2.0])
B = np.array([2.0, 3.0])

# Memory copy available
print(f'Copy: {(-1*(A+B)*B + 1) / 2}')

# No memory copy
#A+B
np.add(A, B, out=A)
print(A)
print(B)
#(A+B)*B
np.multiply(A, B, out=A)
print(A)
print(B)
#(A+B)*B - 1
np.subtract(A, 1, out=A)
print(A)
print(B)
#-1*(A+B)*B + 1
np.negative(A, out=A)
print(A)
print(B)
#(-1*(A+B)*B + 1) / 2
np.divide(A, 2, out=A)
print(f'No copy: {A}')
```

Output:

```
Copy: [-2.5 -7. ]
[3. 5.]
```

38: Generate an array by extracting only the integer part from a uniform random number array of a specific range.

```
import numpy as np
Z = np.random.uniform(0, 10, 10)
print([int(str(i)[0]) for i in Z])
print(Z - Z%1)
print(Z // 1)
print(np.floor(Z))
print(Z.astype(int))
print(np.trunc(Z))
```

Output:

```
[1, 6, 6, 2, 6, 1, 1, 8, 5, 9]
[1. 6. 6. 2. 6. 1. 1. 8. 5. 9.]
[1. 6. 6. 2. 6. 1. 1. 8. 5. 9.]
[1. 6. 6. 2. 6. 1. 1. 8. 5. 9.]
[1 6 6 2 6 1 1 8 5 9]
[1. 6. 6. 2. 6. 1. 1. 8. 5. 9.]
```

39: Define a 5 x 5 array in which the elements of the row are 0 to 4.

```
import numpy as np
Z = np.zeros((5,5))
Z += np.arange(5)
print(Z)
```

Output:

```
[[0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]]
```

```
Z = np.array(np.arange(0,5))
for _ in range(4):
   Z = np.vstack((Z,  np.arange(0,5)))
print(Z)
```

Output:

```
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
```

(Chapter 5) Sequence Generation

40: Define and execute a generator function that generates an array of N elements

(elements are integers between 0 and N-1).

```
import numpy as np
def generate(N):
    for x in range(N):
        yield x
```

Output:

```

```

```
for i in generate(10):
  print(i)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

```
A = np.fromiter(generate(10),dtype=float)
print(A)
```

Output:

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

It is easy to implement using np.fromiter() which creates an array of numpy from iterable objects.

41: Generate a real array of 0 to 1 with size 10. (But exclude 0 and 1.)

```
import numpy as np
np.linspace(0, 1, 12)[1:-1]
```

Output:

```
array([0.09090909, 0.18181818, 0.27272727, 0.36363636, 0.45454545,
       0.54545455, 0.63636364, 0.72727273, 0.81818182, 0.90909091])
```

```
np.linspace(0, 1, 11, endpoint=False)[1:]
```

Output:

```
array([0.09090909, 0.18181818, 0.27272727, 0.36363636, 0.45454545,
       0.54545455, 0.63636364, 0.72727273, 0.81818182, 0.90909091])
```

np.linspace() can generate an array containing equally spaced numbers within a specified range.
If endopoint=False, then the last element of the array (i.e. the right end of the specified range) is excluded from the array. The former method creates two extra arrays and removes both ends of the range by slicing, while the latter creates an array without the right end and removes the left end by slicing.

42: Generate random number vectors of size 10 and sort them in ascending order.

```
%timeit np.sort(np.random.random(1000000))
```

43: Sum faster than np.sum for small arrays.

```
import numpy as np

a = np.arange(10)
# Continuous application of ufunc
%timeit np.add.reduce(a)
# ufunc of C implementation
%timeit np.sum(a)
```

Output:

```
The slowest run took 33.43 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 1.32 µs per loop
The slowest run took 10.32 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 3: 3.89 µs per loop
```

44: Determine if the integer sequences A and B are equal.

```
A = np.random.randint(0,2,5)
B = np.random.randint(0,2,5)
print(A)
print(B)

# Determine if all elements are equal, np.all() can be compared to a scalar
```

```
%timeit eq = np. all(A==B)
print(np. all(A==B))

# check if all elements are equal, np.array_equal() determines array equivalence
%timeit np.array_equal(A,B)
print(np.array_equal(A,B))

# Determine if all elements are close to each other (even if there is a NAN in the same position)
%timeit np.allclose(A,B,equal_nan=True)
print(np.allclose(A,B,equal_nan=True))

# Commented out IPython magic to ensure Python compatibility.
A = np.array([1,np.nan, 2])
B = np.array([1,np.nan, 2])
print(A)
print(B)

# Determine if all elements are equal, np.all() can be compared to a scalar
%timeit eq = np. all(A==B)
print(np. all(A==B))

# check if all elements are equal, np.array_equal() determines array equivalence
%timeit np.array_equal(A,B)
print(np.array_equal(A,B))

# Determine if all elements are close to each other (even if there is a NAN in the same position)
%timeit np.allclose(A,B,equal_nan=True)
print(np.allclose(A,B,equal_nan=True))
```

Output:

```
[1 1 0 1 0]
[0 1 0 1 0]
The slowest run took 7.83 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 3: 3.85 µs per loop
False
The slowest run took 8.58 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 3: 3.81 µs per loop
False
The slowest run took 5.81 times longer than the fastest. This could mean that an intermediate result is being cached.
10000 loops, best of 3: 29.2 µs per loop
False
```

45: Make an immutable array.

```
import numpy as np
a = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])
a.flags.writeable = False
a[0] = 4; # Error as it's unchangeable
```

Output:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-2b2734a60fd9> in <module>()
      2 a = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])
      3 a.flags.writeable = False
----> 4 a[0] = 4; #It's an error (because I'm making it unchangeable).

ValueError: assignment destination is read-only
```

46: Convert from Cartesian coordinates (xy) to polar coordinates (r, θ).

```
import numpy as np
import math
# cartesian
a = np.random.random((10,2))
x = a[:, 0]
y = a[:, 1]
# convert to polar(r^2 = x^2 + y^2, tanθ = y/x)
r = np.sqrt(x**2 + y**2)
theta = []
for x1, y1 in zip(x, y):
  theta.append(math.atan(y1/x1))
```

```
print(r)
print(theta)

X,Y = a[:,0], a[:,1]
R = np.sqrt(X**2+Y**2)
T = np.arctan2(Y,X)
print(R)
print(T)
```

Output:

```
[0.36557586 0.51545783 0.73283743 0.31851877 0.70162908 1.08979207
0.65337167 0.42607204 0.81758765 0.15791347]
[0.5410951641282364, 0.1946941591884679 4, 1.196519173783585,
0.9611078425515306, 0.5269888796611327, 0.6290035529936863, 0.
7316122981119595, 0.6143838477282143, 0.5415761837628309,
1.054302594907771]
```

different solution

```
X,Y = a[:,0], a[:,1]
R = np.sqrt(X**2+Y**2)
T = np.arctan2(Y,X)
print(R)
print(T)
```

Output:

```
[0.36557586 0.51545783 0.73283743 0.31851877 0.70162908 1.08979207
 0.65337167 0.42607204 0.81758765 0.15791347]
[0.54109516 0.19469416 1.19651917 0.96110784 0.52698888 0.62900355
 0.7316123  0.61438385 0.54157618 1.05430259]
```

47: Create an array of random numbers of size 10 and replace the largest element with 0.

```
import numpy as np
a = np.random.random(10)
print(a)
a[np.argmax(a)] = 0
print(a)
```

Output:

```
[0.6893331  0.67895251 0.04555048 0.75274025 0.32527895 0.63911266
 0.95694472 0.14600348 0.98576122 0.73088862]
[0.6893331  0.67895251 0.04555048 0.75274025 0.32527895 0.63911266
 0.95694472 0.14600348 0.         0.73088862]
```

48: Define a structural array where (x, y) = (0, 0) to (1, 1) is filled with evenly spaced grid sequences.

```
a = np.zeros((100, 100), [('x',float),('y',float)])
a['x'], a['y'] = np.meshgrid(np.linspace(0, 1, 100),
                             np.linspace(0, 1, 100))
print(a)
```

Output:

```
[[(0.         , 0.         ) (0.01010101, 0.         )
  (0.02020202, 0.         ) ... (0.97979798, 0.         )
  (0.98989899, 0.         ) (1.         , 0.         )]
 [(0.         , 0.01010101) (0.01010101, 0.01010101)
  (0.02020202, 0.01010101) ... (0.97979798, 0.01010101)
  (0.98989899, 0.01010101) (1.         , 0.01010101)]
 [(0.         , 0.02020202) (0.01010101, 0.02020202)
  (0.02020202, 0.02020202) ... (0.97979798, 0.02020202)
  (0.98989899, 0.02020202) (1.         , 0.02020202)]
 ...
 [(0.         , 0.97979798) (0.01010101, 0.97979798)
  (0.02020202, 0.97979798) ... (0.97979798, 0.97979798)
  (0.98989899, 0.97979798) (1.         , 0.97979798)]
 [(0.         , 0.98989899) (0.01010101, 0.98989899)
  (0.02020202, 0.98989899) ... (0.97979798, 0.98989899)
  (0.98989899, 0.98989899) (1.         , 0.98989899)]
 [(0.         , 1.         ) (0.01010101, 1.         )
  (0.02020202, 1.         ) ... (0.97979798, 1.         )
  (0.98989899, 1.         ) (1.         , 1.         )]]
```

49: Construct the Cauchy matrix Cij=1/(xi-yj) for two sequences of X and Y.

```
x = np.arange(8)
y = np.arange(11, 19)
l = []
for i in x:
  tmp = []
  for j in y:
    tmp.append(1/(i - j))
  l.append(tmp)
print(np.array(l))
```

Output:

```
[[-0.09090909 -0.08333333 -0.07692308 -0.07142857 -0.06666667 -0.0625
  -0.05882353 -0.05555556]
 [-0.1        -0.09090909 -0.08333333 -0.07692308 -0.07142857 -0.06666667
  -0.0625     -0.05882353]
 [-0.11111111 -0.1        -0.09090909 -0.08333333 -0.07692308 -0.07142857
  -0.06666667 -0.0625     ]
 [-0.125      -0.11111111 -0.1        -0.09090909 -0.08333333 -0.07692308
  -0.07142857 -0.06666667]
 [-0.14285714 -0.125      -0.11111111 -0.1        -0.09090909 -0.08333333
  -0.07692308 -0.07142857]
 [-0.16666667 -0.14285714 -0.125      -0.11111111 -0.1        -0.09090909
  -0.08333333 -0.07692308]
 [-0.2        -0.16666667 -0.14285714 -0.125      -0.11111111 -0.1
  -0.09090909 -0.08333333]
 [-0.25       -0.2        -0.16666667 -0.14285714 -0.125      -0.11111111
  -0.1        -0.09090909]]
```

If you write in the standard Python library, it's seven lines.

```
import numpy as np
x = np.arange(8)
y = np.arange(11, 19)
print(1/np.subtract.outer(x, y))
```

Output:

```
[[-0.09090909 -0.08333333 -0.07692308 -0.07142857 -0.06666667 -0.0625
   -0.05882353 -0.05555556]
 [-0.1        -0.09090909 -0.08333333 -0.07692308 -0.07142857 -0.06666667
   -0.0625     -0.05882353]
 [-0.11111111 -0.1        -0.09090909 -0.08333333 -0.07692308 -0.07142857
   -0.06666667 -0.0625     ]
 [-0.125      -0.11111111 -0.1        -0.09090909 -0.08333333 -0.07692308
   -0.07142857 -0.06666667]
 [-0.14285714 -0.125      -0.11111111 -0.1        -0.09090909 -0.08333333
   -0.07692308 -0.07142857]
 [-0.16666667 -0.14285714 -0.125      -0.11111111 -0.1        -0.09090909
   -0.08333333 -0.07692308]
 [-0.2        -0.16666667 -0.14285714 -0.125      -0.11111111 -0.1
   -0.09090909 -0.08333333]
 [-0.25       -0.2        -0.16666667 -0.14285714 -0.125      -0.11111111
   -0.1        -0.09090909]]
```

You can write in Numpy to be more concise.
outer(A, B) can apply universal functions to A, B. Subtraction is applied to all elements in np.subtract.outer. This will result in a single line.

(Chapter 6) Numpy Array Customization 1

50: Show the maximum and minimum values

that Numpy's scalar types (e.g. np.float32 and np.float64) can represent.

```
import numpy as np
print(f"np.int8.max: {np.iinfo(np.int8).max}")
print(f"np.int8.min: {np.iinfo(np.int8).min}")

print(f"np.int32.max: {np.iinfo(np.int32).max}")
print(f"np.int64.max: {np.iinfo(np.int64).max}")

print(f"np.float16.max: {np.finfo(np.float64).max}")
print(f"np.float16.min: {np.finfo(np.float64).min}")

print(f"np.float32.max: {np.finfo(np.float32).max}")
print(f"np.float32.min: {np.finfo(np.float32).min}")

print(f"np.float64.max: {np.finfo(np.float64).max}")
print(f"np.float64.min: {np.finfo(np.float64).min}")
```

Output:

```
np.int8.max:127
np.int8.min:-128
np.int32.max:2147483647
np.int64.max:9223372036854775807
np.float16.max:1.7976931348623157e+308
np.float16.min:-1.7976931348623157e+308
np.float32.max:3.4028234663852886e+38
np.float32.min:-3.4028234663852886e+38
np.float64.max:1.7976931348623157e+308
np.float64.min:-1.7976931348623157e+308
```

You can use np.finfo() to display the limits (maximum and minimum values) that can represent a scalar type.

51: Do not abbreviate all the elements in the numpy array, but output them to the standard output.

```python
import numpy as np
a = np.zeros((16, 16))
np.set_printoptions(threshold=1000)
print(a)
np.set_printoptions(threshold=0)
print(a)
np.set_printoptions(threshold=float("inf"))
print(a)
```

Output:

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

You can change the display format of the Numpy array (number of digits, exponential notation, zero filling, etc.) with np.set_options(). The values of the original array elements themselves do not change.

52 : Given a vector (a) and a scalar (b) , output the scalar value closest to b in a.

```
import numpy as np
a = np.arange(100)
b = np.random.uniform(0,100)
index = (np.abs(a-b)).argmin()
print(a[index])
```

Output:

```
81
```

```
print(a)
print(b)
print(a[index])
```

Output:

```
[0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
80.73762323040114
81
```

53: Generate a structured array to represent the (x,y) coordinates and RGB.

```
import numpy as np
A = np.zeros(10, [('position', [ ('x', float), ('y', float)]), ('color
[ ('r', int), ('g', int), ('b', int)])])
A[0] = ((3,4),(255,255,255))
print(A)
```

Output:

```
[((3., 4.), (255, 255, 255)) ((0., 0.), (  0,   0,   0))
 ((0., 0.), (  0,   0,   0)) ((0., 0.), (  0,   0,   0))
 ((0., 0.), (  0,   0,   0)) ((0., 0.), (  0,   0,   0))
 ((0., 0.), (  0,   0,   0)) ((0., 0.), (  0,   0,   0))
 ((0., 0.), (  0,   0,   0)) ((0., 0.), (  0,   0,   0))]
```

A structured array of numpy is an array that incorporates variables of different types with a self-defined dtype. The best thing is that you can refer to it with elements that you define in your own field names.

```
print(A["position"]["x"])
print(A["position"]["y"])
print(A["color"])
print(A["color"]["r"])
print(A["color"]["g"])
print(A["color"]["b"])
```

Output:

```
[3. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[4. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[(255, 255, 255) (  0,   0,   0) (  0,   0,   0) (  0,   0,   0)
 (  0,   0,   0) (  0,   0,   0) (  0,   0,   0) (  0,   0,   0)
 (  0,   0,   0) (  0,   0,   0)]
[255   0   0   0   0   0   0   0   0   0]
[255   0   0   0   0   0   0   0   0   0]
[255   0   0   0   0   0   0   0   0   0]
```

54: Find the distance matrix of (100,100) from the random number array of (100,2).

```
import numpy as np
A = np.random.random((100,2))
def calc_dist(A):
  X,Y = np.atleast_2d(A[:,0], A[:,1])
  D = np.sqrt( (X-X.T)**2 + (Y-Y.T)**2)
  return D
D = calc_dist(A)
print(D.shape)
print(D[0])
```

Output:

```
(100, 100)
[0.         0.90751035 0.57527022 0.72292981 0.73691975 0.68768711
 0.3341895  0.47301663 0.88938439 1.08349913 0.77979028 0.31408419
 0.64570021 0.25823866 0.87810787 0.64776507 0.82839315 0.82353303
 0.66954942 0.61167008 0.92581598 0.25727952 0.60908085 0.51517212
 0.10459209 1.07014534 0.67742698 0.14503016 0.50304777 0.42252417
 0.53406984 0.28127503 0.88830304 0.16775117 0.86368461 0.14829758
 0.32396949 0.66903171 0.83522993 0.44395534 0.40348798 0.59608362
 0.8309456  0.84620591 0.44279026 0.96769016 0.89481238 1.16274424
 0.64585525 0.28440462 0.31952523 0.79375321 0.6234229  0.70743907
 0.68777124 0.81466873 0.56788761 0.8354413  0.92321332 0.39408872
 0.67939639 0.51713605 0.63507978 0.69107177 0.92095709 0.7678387
 0.62736888 0.72313161 0.61004382 0.75376943 0.44938125 1.10608139
 0.65110027 0.13357437 0.9552045  0.73910648 0.12047484 0.41947176
 0.23467406 1.04869291 0.14777029 0.97777266 0.69332602 1.04870839
 0.47409066 0.60744086 0.92592881 0.57184871 0.91301854 0.87719543
 0.52250321 0.8122932  0.21631095 0.547498   0.70202204 0.71304752
 0.87080695 0.4484347  0.55781606 0.51721326]
```

Distance matrix: an N*N square matrix with (i, j) components being the

distance between the i-th element and the j-th element for N data. It is faster and more concise to write it in Scipy, as shown below.

```
# scipy is faster
import scipy

A = np.random.random((100, 2))
D = scipy.spatial.distance.cdist(A, A)
print(D.shape)
print(D[0])
```

Output:

```
(100, 100)
[0.         1.03935821 0.36079105 0.37905807 0.46905293 0.37503685
 0.44253151 0.40508408 0.44465633 0.68129221 0.7116416  0.98515857
 0.66392105 0.61676663 0.548769   0.70847085 0.81249583 0.70656385
 0.11097781 0.52070622 0.91117595 0.96820494 0.80108825 0.65569353
 0.6596279  0.31574213 0.74156201 0.95505741 0.36972777 0.24200588
 0.27094785 0.97820321 0.45558894 0.8242347  0.46031078 0.16825469
 0.57900814 0.3127563  0.61039702 0.38662549 0.48049721 0.83976051
 0.0504217  0.76088125 0.33536094 1.08420214 0.85640085 0.66410083
 0.92408413 0.42831734 0.56572093 0.34846919 0.83242653 0.79995649
 0.75583232 0.52424516 0.09874375 0.1858803  0.63202814 0.42341886
 0.68838025 0.70515334 0.69849688 0.88846944 0.59590736 0.20108878
 0.41000772 0.15246115 0.52952348 0.72735996 0.70452863 0.85573789
 0.56382676 0.82910148 0.24906211 0.69157152 0.28929279 0.37014069
 0.25328474 1.10242136 0.63267359 0.38833625 1.06526145 0.22729596
 0.22941968 0.39860422 0.34957127 0.83971631 0.85486362 0.45367742
 0.90740335 0.45164685 0.18206878 0.0748512  0.26012551 0.73674534
 0.81715242 0.99911815 0.36508667 0.1182732 ]
```

55: Change the float array to an int array.

```
import numpy as np
float_x = np.array([3.14, 1592, 653589.7932], dtype='float32')
print(float_x)
int_x = float_x.astype('int32')
print(int_x)
```

Output:

```
[3.140000e+00 1.592000e+03 6.535898e+05]
[      3    1592 653589]
```

You can change dtype later with np.ndarray.astype(). Another solution (below) uses np.ndarray.view(). astype is recommended because the original array Z has been changed.

```
Z = (np.random.rand(10)*100).astype(np.float32)
print(Z)
Y = Z.view(np.int32)
Y[:] = Z
print(Y)
print(Z)
```

Output:

```
[73.66704   52.399887 13.922787 86.01413   98.64411    6.681386 72.442444
 95.11177   49.204983 46.742943]
[73 52 13 86 98  6 72 95 49 46]
[1.02e-43 7.29e-44 1.82e-44 1.21e-43 1.37e-43 8.41e-45 1.01e-43 1.33e-43
 6.87e-44 6.45e-44]
```

56: Store the following files in a numpy array.

```
1, 2, 3, 4, 5
6,  ,  , 7, 8
 ,  , 9,10,11
```

```
import numpy as np
from io import StringIO

s = StringIO('''1, 2, 3, 4, 5

                6,  ,  , 7, 8

                 ,  , 9,10,11
''')
A = np.genfromtxt(s, delimiter=",", dtype=np.int)
print(A)
```

Output:

```
[[ 1  2  3  4  5]
 [ 6 -1 -1  7  8]
 [-1 -1  9 10 11]]
```

You can use np.genfromtxt() to store a numpy array from a text file with the missing data treatment as well.

```
lines = '''1, 2, 3, 4, 5

               6,  ,  , 7, 8

                ,  , 9,10,11
'''
l = []
for row in lines.split('¥n'):
  if row:
    row = row.replace(' ','')
    tmp = []
    for col in row.split(','):
      if col:
        tmp.append(col)
      else:
        tmp.append(-1)
    l.append(tmp)
a = np.array(l, dtype=np.int)
print(a)
```

Output:

```
[[ 1  2  3  4  5]
 [ 6 -1 -1  7  8]
 [-1 -1  9 10 11]]
```

57: Define three ways to get the index and value of a two-dimensional numpy array and

compare their speeds.

```
import numpy as np

def method1(a):
  l = []
  for i in range(a.shape[0]):
    for j in range(a.shape[1]):
      l.append(((i,j),a[i,j]))
  return l

def method2(a):
  return np.ndenumerate(a)

def method3(a):
  return np.ndindex(a.shape)
a = np.arange(9).reshape(3,3)
print(a)
```

Output:

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

method1:Python standard functions, method2:np.ndenumerate(), method3:np.ndindex(). You can use them all for the same purpose as follows.

```
for i in method1(a):
  print(i)
```

Output:

```
((0, 0), 0)
((0, 1), 1)
((0, 2), 2)
((1, 0), 3)
((1, 1), 4)
((1, 2), 5)
((2, 0), 6)
((2, 1), 7)
((2, 2), 8)
```

```
for i in method2(a):
  print(i)
```

Output:

```
((0, 0), 0)
((0, 1), 1)
((0, 2), 2)
((1, 0), 3)
((1, 1), 4)
((1, 2), 5)
((2, 0), 6)
((2, 1), 7)
((2, 2), 8)
```

```
for i in method3(a):
  print((i,a[i]))
```

Output:

```
((0, 0), 0)
((0, 1), 1)
```

With timeit, you can only know the elapsed time for each iteration, and you can't see the effect of the array size on the execution speed, so you can make your own time measurement function.

```
import time

def capture_elapsed_time(method, arg):
  start = time.time()
  res = method(arg)
  elapsed_time = time.time() - start
  return elapsed_time,res

a = np.arange(9).reshape(3,3)
b = np.arange(90000).reshape(300,300)
print(f"elapsed_time(method1, a):{capture_elapsed_time(method1, a)[0]}
print(f"elapsed_time(method2, a)::{capture_elapsed_time(method2,
a)[0]}")
print(f"elapsed_time(method3, a)::{capture_elapsed_time(method3,
a)[0]}")
print(f"elapsed_time(method1, b):{capture_elapsed_time(method1, b)[0]}
print(f"elapsed_time(method2, b):{capture_elapsed_time(method2, b)[0]}
print(f"elapsed_time(method3, b):{capture_elapsed_time(method3, b)[0]}
```

Output:

```
elapsed_time(method1, a):1.71661376953125e-05
elapsed_time(method2, a)::1.5497207641601562e-05
elapsed_time(method3, a)::0.00013065338134765625
elapsed_time(method1, b):0.10753536224365234
elapsed_time(method2, b):1.33514404296875e-05
elapsed_time(method3, b):0.0001380443572998047
```

We confirmed that the method using np.ndenumerate() is the fastest regardless of the array size.

58: Generate a two-dimensional Gaussian kernel array.

```
import numpy as np
X, Y = np.meshgrid(np.linspace(-1,1,10), np.linspace(-1,1,10))
D = np.sqrt(X*X+Y*Y)
sigma, mu = 1.0, 0.0
G = np.exp(-( (D-mu)**2 / ( 2.0 * sigma**2 ) ) )
print(G)
import matplotlib.pyplot as plt
plt.imshow(G, interpolation='none')
```

Output:

```
[[0.36787944 0.44822088 0.51979489 0.57375342 0.60279818 0.60279818
  0.57375342 0.51979489 0.44822088 0.36787944]
 [0.44822088 0.54610814 0.63331324 0.69905581 0.73444367 0.73444367
  0.69905581 0.63331324 0.54610814 0.44822088]
 [0.51979489 0.63331324 0.73444367 0.81068432 0.85172308 0.85172308
  0.81068432 0.73444367 0.63331324 0.51979489]
 [0.57375342 0.69905581 0.81068432 0.89483932 0.9401382  0.9401382
  0.89483932 0.81068432 0.69905581 0.57375342]
 [0.60279818 0.73444367 0.85172308 0.9401382  0.98773022 0.98773022
  0.9401382  0.85172308 0.73444367 0.60279818]
 [0.60279818 0.73444367 0.85172308 0.9401382  0.98773022 0.98773022
  0.9401382  0.85172308 0.73444367 0.60279818]
 [0.57375342 0.69905581 0.81068432 0.89483932 0.9401382  0.9401382
  0.89483932 0.81068432 0.69905581 0.57375342]
 [0.51979489 0.63331324 0.73444367 0.81068432 0.85172308 0.85172308
  0.81068432 0.73444367 0.63331324 0.51979489]
 [0.44822088 0.54610814 0.63331324 0.69905581 0.73444367 0.73444367
  0.69905581 0.63331324 0.54610814 0.44822088]
 [0.36787944 0.44822088 0.51979489 0.57375342 0.60279818 0.60279818
  0.57375342 0.51979489 0.44822088 0.36787944]]
<matplotlib.image.AxesImage at 0x7f9bbf6d9e80>
```

different solution

```
import numpy as np
from scipy import signal

def gkern(kernlen=21, std=3):
    gkern1d = signal.gaussian(kernlen, std=std).reshape(kernlen, 1)
    gkern2d = np.outer(gkern1d, gkern1d)
    return gkern2d

print(gkern(11,5))

import matplotlib.pyplot as plt
plt.imshow(gkern(11, 5), interpolation='none')
```

Output:

```
[[0.36787944 0.44043165 0.50661699 0.55989837 0.59452055 0.60653066
  0.59452055 0.55989837 0.50661699 0.44043165 0.36787944]
 [0.44043165 0.52729242 0.60653066 0.67032005 0.71177032 0.72614904
  0.71177032 0.67032005 0.60653066 0.52729242 0.44043165]
 [0.50661699 0.60653066 0.69767633 0.77105159 0.81873075 0.83527021
  0.81873075 0.77105159 0.69767633 0.60653066 0.50661699]
 [0.55989837 0.67032005 0.77105159 0.85214379 0.90483742 0.92311635
  0.90483742 0.85214379 0.77105159 0.67032005 0.55989837]
 [0.59452055 0.71177032 0.81873075 0.90483742 0.96078944 0.98019867
  0.96078944 0.90483742 0.81873075 0.71177032 0.59452055]
 [0.60653066 0.72614904 0.83527021 0.92311635 0.98019867 1.
  0.98019867 0.92311635 0.83527021 0.72614904 0.60653066]
 [0.59452055 0.71177032 0.81873075 0.90483742 0.96078944 0.98019867
  0.96078944 0.90483742 0.81873075 0.71177032 0.59452055]
 [0.55989837 0.67032005 0.77105159 0.85214379 0.90483742 0.92311635
  0.90483742 0.85214379 0.77105159 0.67032005 0.55989837]
 [0.50661699 0.60653066 0.69767633 0.77105159 0.81873075 0.83527021
  0.81873075 0.77105159 0.69767633 0.60653066 0.50661699]]
```

Gaussian kernel matrix used for smoothing images, which can be described more concisely using scipy.signal.

59: Place p elements randomly in a two-dimensional array.

```
import numpy as np
a = np.zeros((10,10))
p = 3
for i in range(p):
  r = np.random.choice(10)
  c = np.random.choice(10)
  a[r, c] = 1
print(a)
```

Output:

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
import numpy as np
n = 10
p = 3
Z = np.zeros((n,n))
np.put(Z, np.random.choice(range(n*n), p, replace=False),1)
print(Z)
```

Output:

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

np.put() can be used to replace the elements of a specified index with the specified value.

(Chapter 7) The More Practical Numpy

61: Sort the array by the nth column.

```
import numpy as np
a = np.array([[7,8,9],[1,2,3],[4,5,6]])
n = 2
print(a)
b = a[:,2].argsort()
print(b)
print(a[b])
```

Output :

```
[[7 8 9]
 [1 2 3]
 [4 5 6]]
[1 2 0]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

In a[:, 2], you can get the third column as an array and argsort() to get the list of indexes for the ascending sort by the third column. By specifying this index list as the index of the original numpy array (called a fancy index), we can achieve sorting by the third column. By the way, numpy arrays are zero-origin (starting from 0), so you can specify the nth column with a[:, n-1].

62: Generate another array with non-zero elements from the sequence [1,2,0,0,4,0].

```
import numpy as np
a = np.array([1, 2, 0, 0, 4, 0])
# values
print(a[a!=0])
# indices
print(np.nonzero(a)[0])
print(a[np.nonzero(a)[0]])
```

Output:

```
[1 2 4]
[0 1 4]
[1 2 4]
```

You can create another numpy array consisting of only the elements with the specified condition by placing a conditional expression at the point where you specify the index or slicing in the numpy array. Using this mechanism, the first solution is to create a separate array with only non-zero elements. As an alternative solution, in a[a!=0], we can use np.nonzero() to return another numpy array with an index of non-zero elements. We need to specify a[a[np.nonzero(a)[0]] because the return value is an index and tuple.

63: Determine if the two-dimensional array contains a column with only Null elements.

```python
import numpy as np
# 0 column
a = np.array([[1, 0, 3],[1, 0, 5]])

# all 0
b = np.array([[0, 0, 0],[0, 0, 0]])

# no 0
c = np.array([[1, 2, 3],[1, 4, 5]])

def judge_col0(a):
  dic = {}
  col_zero_flg = False
  for i, row in enumerate(a):
    for j, col in enumerate(row):
      if j not in dic and col == 0:
        dic[j] = 1
      else:
        if col == 0:
          dic[j] += 1
      if j in dic:
        if dic[j] == a.shape[0]:
          col_zero_flg = True
          break
  return col_zero_flg
print(judge_col0(a))
print(judge_col0(b))
print(judge_col0(c))

print((~a.all(axis=0)).any())
print((~b.all(axis=0)).any())
print((~c.all(axis=0)).any())

print([bool(i) for i in range(-10, 10, 1)])

print(a.all(axis=0))
```

Output:

```
True
True
False
True
True
False

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

Define two arrays (a, b) that contain columns with only Null elements, and two arrays (c) that do not contain them. Therefore, if we give a, b, and c as arguments, we expect them to be included (True) , included (True), and not included (False).

First, let's try to solve the problem in a simple way without using any numpy functions.

```
# 0 column
a = np.array([[1,0,3],[1,0,5]])

# all 0
b = np.array([[0,0,0],[0,0,0]])
```

```
# no 0
c = np.array([[1,2,3],[1,4,5]])


def judge_col0(a):
  dic = {}
  col_zero_flg = False
  for i, row in enumerate(a):
    for j, col in enumerate(row):
      if j not in dic and col == 0:
        dic[j] = 1
        else:
        if col == 0:
          dic[j] += 1
      if j in dic:
        if dic[j] == a.shape[0]:
          col_zero_flg = True
          break
  return col_zero_flg
print(judge_col0(a))
print(judge_col0(b))
print(judge_col0(c))
```

We scan all the elements and increment the count of the columns that were 0 elements in the dictionary, and then flag and return the columns that have a count by the number of rows. This can be described very simply using the numpy functions all() and any(), as follows

```
print((~a. all(axis=0)). any())
print((~b. all(axis=0)). any())
print((~c. all(axis=0)). any())
```

First of all, 0 is false when cast as bool (true or false) ; true of

type bool corresponds to 1 of type int, and false of type bool corresponds to 0 of type int, respectively. And if we check the other integers, we can see that non-zero integers are treated as true.

```
print([bool(i) for i in range(-10,10,1)])
```

Now let's expand the equation a little bit to understand it.

```
print(a. all(axis=0))
print(b. all(axis=0))
print(c. all(axis=0))
```

First of all, if axis=0 is specified, then in a two-dimensional array such as [[a, b, c],[d, e, f]], each column of [a, b, c] and [d, e, f] with the outermost parentheses removed is counted. That is, for each of (a, d), (b, e), and (c, f), if all of them are true, then they are true, and if they are not true, then they are false. In this case, we want to find a column that is all 0 elements on this axis 0 (or False if it's true or false), so we need to use the negative form. ~(tilde) means negative.

```
print(~a. all(axis=0))
print(~b. all(axis=0))
print(~c. all(axis=0))
```

True is contained in the column where all the columns were 0. Therefore, by using any() here, if any of the columns are True, then it is True, which means that it may or may not contain the columns of the Null element in this problem. It is possible to make a judgment that Please note that the position of the parentheses changes the meaning and the result.

```
print((~a. all(axis=0)). any())
print((~b. all(axis=0)). any())
print((~c. all(axis=0)). any())
```

64: Find the closest value to the given value in an array of arbitrary shapes.

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
b = 5.1
nearest = a.flat[np.abs(a - b).argmin()]
print(nearest)
```

Output:

```
5
```

65: Find the sum of sequences of the (1,3) and (3,1) forms using the universal operation or iterator.

```
import numpy as np
A = np.array([[1,2,3]])
B = np.array([[0],[1],[2]])
# universal operation
print(A + B)
print("----------")
# Use iterator
iter = np.nditer([A, B, None])
for x,y,z in iter:
  z[...] = x+y
print(iter.operands[2])
```

Output:

```
[[1 2 3]
 [2 3 4]
 [3 4 5]]
----------
[[1 2 3]
 [2 3 4]
 [3 4 5]]
```

As you can see, the universal operation is A+B. np.nditer is a function used to scan numpy multidimensional arrays, which is applied to display the result of assigning Z[...] to the first part of a numpy array. It is applied to display the result by assigning it to the first position of None. is a shorthand notation called Ellipsis, which can be used in the following way and is very useful.

```
a = np.array([[1, 2, None], [4, 5, 6]])
a[..., 0] = 3
print(a)
```

Output:

```
[[3 2 None]
 [3 5 6]]
```

66: Create your own array class with name attributes.

```
import numpy as np
class ArrayWithName(np.ndarray):
    def __new__(cls, array, name=None):
        obj  = np.asarray(array).view(cls)
        obj.name = name
        return obj
    def __array_finalize__(self, obj):
        if obj is None: return
        self.info = getattr(obj, 'name', None)

a = ArrayWithName(np.array([1,2,3,4,5]),"one to five")
print(a)
print(a.name)

a = ArrayWithName(np.array([1,2,3,4,5]))
print(a)
print(a.name)
```

Output:

```
[1 2 3 4 5]
one to five
[1 2 3 4 5]
None
```

Subclassing np.ndarray to add attribute information. There are three points in __new__() This is the main initialization process.

1) class subclass name(np.ndarray): explicit inheritance by

2) cast an existing ndarray to a subclass by view casting np.asarray(array).view(cls)

3) Generate it from the template instance return obj to create it as an instance of a new subclass. This will create a different array than the original one. For example

```
a = np.array([1,2,3])
b = ArrayWithName(a, 'one to three')
a is b
```

Output:

```
False
```

Then, __array_finalize__() does the post-processing after creating a new instance from the template. For example, in this example, we put None in the name attribute if it was initialized without a name attribute argument.

67: Given a sequence A and another sequence B, add 1 to the value of the element of A indexed by the value of the element of B.

```
import numpy as np
A = np.array([1, 2, 3])
print(A)
B = np.array([0, 0, 1])
print(B)
C = A + np.bincount(B, minlength=len(A))
print(C)
```

Output:

```
[1 2 3]
[0 0 1]
[3 3 3]
```

Array B has 2 0s and 1 1, so we add 2 to the 0th and 1 to the 1st. np.bincount allows us to count the number of occurrences of each element.

```
print(A)
print(B)

np.add.at(A, B, 1)
print(A)
```

at(a, indices, b=None) behaves like a[indices] += b.

```
A = np.array([1, 2, 3])
print(A)
B = np.array([0, 0, 1])
print(B)
for i in B:
  A[i]+=1
print(A)
```

Output:

```
[1 2 3]
[0 0 1]
[3 3 3]
```

If the for loop is an acceptable use, this is the easiest way to write it.

68: How to accumulate the elements of a vector (X) into an array (F) based on an index list (I)?

Solution 1

```
import numpy as np

X = np.array([1, 2, 3])
print(X)
I = np.array([5, 2, 3])
print(I)
F = np.bincount(I, weights=X)
print(F)
```

Output:

```
[1 2 3]
[5 2 3]
[0. 0. 2. 3. 0. 1.]
```

Solution 2

```
X = np.array([1, 2, 3])
I = np.array([5, 2, 3])
F = np.zeros(np.max(I)+1)

for x, i in zip(X, I):
  F[i] = x
F
```

Output:

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

Solution 1 uses np.bincount to aggregate the frequency of occurrence of each value in the given input array.

Solution 2 is a for-loop solution: zip to scan the input array and the index

together, and store the values in an array with the length of the index pre-initialized by 0.

69: Define a numpy array of length x width x color (RGB) and count the number of unique colors.

```
import numpy as np
w = 32
h = 32
a = np.random.randint(0, 256, (h, w, 3))
f = a[..., 0]*256*256 + a[..., 1]*256 + a[..., 2]
print(f.shape)
print(len(np.unique(f)))
```

Output:

```
(32, 32)
1024
```

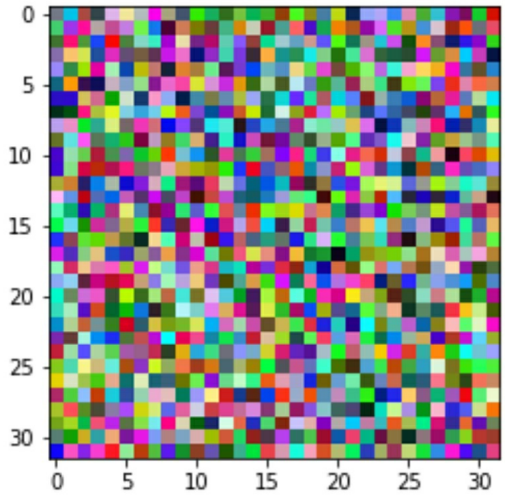
This is accomplished by converting the number of unique colors from RGB to a scalar and then using np.unique() to get the number of unique elements.

32 x 32 x RGB looks like the following when displayed.

```
import matplotlib.pyplot as plt
plt.imshow(a)
```

Output:

```
<matplotlib.image.AxesImage at 0x7f3d62f4a8d0>
```

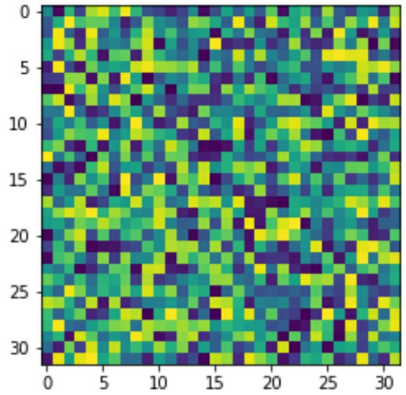


The colors are represented in scalar type as follows.

```
plt.imshow(f)
```

Output:

```
<matplotlib.image.AxesImage at 0x7f3d66bda470>
```



A simpler example is below.

```
a = np.array([[[1, 2, 8], [1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 67], [1, 255, 3]]])
f = a[..., 0]*256*256 + a[..., 1]*256 + a[..., 2]
print(len(np.unique(f)))
```

Output:

```
5
```

The alternative solution is below. You can also specify an axis and call it unique.

```
a= np.array([[[255, 255, 255],[0, 123, 0],[255, 255, 255],[0, 0, 0]]])
print(a.shape)
print(len(np.unique(a, axis=1)[0, :, :]))
```

Output:

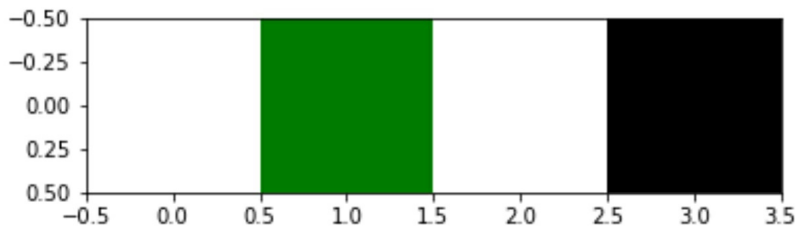
```
(1, 4, 3)
3
```

As you can see below, there are really three colors.

```
plt.imshow(a)
```

Output:

```
<matplotlib.image.AxesImage at 0x7f3d5ad132b0>
```



(Chapter 8) Statistics and Aggregation with Numpy

70: Calculate the sum of the last two

axes in a 4-dimensional vector in a lump sum.

```
import numpy as np
a = np.random.randint(0, 10, (2, 2, 2, 1))
sum = a.sum(axis=(-2, -1))
print(a)
print("---")
print(sum)
```

Output:

```
[[[[5]
   [7]]

  [[0]
   [3]]]


 [[[8]
   [4]]

  [[1]
   [4]]]]
---
[[12  3]
 [12  5]]
```

np.sum() can calculate the sum of a numpy array, and the axes can be specified in the argument axis.

71: Using a vector S of the same size for a one-dimensional vector D (the index is stored in the value), compute the average of the subset specified by the index of D.

```
import numpy as np
D = np.array([1, 1, 1, 2, 4, 4])
S = np.array([0, 1, 1, 1, 2, 2])
print(D)
print(S)
# Numpy
print(np.bincount(S, weights=D)/np.bincount(S))
# Pandas
import pandas as pd
print(pd.Series(D).groupby(S).mean())
```

Output:

```
[1 1 1 2 4 4]
[0 1 1 1 2 2]
[1.         1.33333333 4.        ]
0    1.000000
1    1.333333
2    4.000000
dtype: float64
```

The mean of the subset specified in the index in D is intuitive when solved by groupby of pandas, but it can also be solved by np.bincount.

72: Get the diagonal elements of the dot product of the two matrices.

```
import numpy as np
A = np.random.uniform(0,1,(32,32))
B = np.random.uniform(0,1,(32,32))

# Slow version
%timeit np.diag(np.dot(A, B))

# Fast version
%timeit  np.sum(A * B.T, axis=1)

# Faster version
%timeit np.einsum("ij,ji->i", A, B)
```

Output:

```
The slowest run took 15.42 times longer than the fastest. This coul
mean that an intermediate result is being cached.
100000 loops, best of 3: 9.09 µs per loop
The slowest run took 12.74 times longer than the fastest. This coul
mean that an intermediate result is being cached.
100000 loops, best of 3: 9.11 µs per loop
The slowest run took 16.76 times longer than the fastest. This coul
mean that an intermediate result is being cached.
100000 loops, best of 3: 4.27 µs per loop
```

Einstein notation style is the fastest.

73: Swap the two rows in the array.

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
a[[0, 1]] = a[[1, 0]]
print('-----')
print(a)
```

Output:

```
[[1 2 3]
 [4 5 6]]
-----
[[4 5 6]
 [1 2 3]]
```

74: Given a bincount of B sequence named C, generate an array A such that np.bincount(A) == C.

```
import numpy as np
B = np.array([1, 2, 2, 2, 3])
print(B)
C = np.bincount(B)
print(C)
A = np.repeat(a=np.arange(len(C)), repeats=C)
print(A)
```

Output:

```
[1 2 2 2 3]
[0 1 3 1]
[1 2 2 2 3]
```

It is possible to repeat the elements of the array given by np.repeat. The process is achieved by repeating the interval frequency of the original array. If we check the values, we can see that np.bincount(A) == C, as shown below.

```
np.bincount(A)
```

Output:

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

75: Swap the two rows in the array.

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print(a)
a[[0,1]] = a[[1,0]]
print('-----')
print(a)
```

Output

```
[[1 2 3]
 [4 5 6]]
-----
[[4 5 6]
 [1 2 3]]
```

Fancy indexing makes this possible.

While normal indexing allows you to specify elements by [row, col], fancy indexing allows you to generate another array in the order of its index, if you pass in a list.

Indexing

```
a[0,1]
```

Output:

```
5
```

Fancy Indexing

```
a[ [ 0, 1] ]
```

Output:

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

```
a[ [ 1, 0] ]
```

Output:

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

```
a[ [ 1, 0, 0] ]
```

Output:

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

The address of the original array and the array generated by fancy indexing are different.

```
print(id(a))
print(id(a[[0, 1, 0]]))
```

Output:

```
140416401301792
140416270773984
```

76: Generate 10 triangles in which each point is represented by an (x, y) coordinate, and find 10 unique coordinates that represent the edges of multiple triangles sharing an edge.

If the three sides of the triangle are a, b, and c, then the following inequality holds for the triangle to be a triangle.

$$a < b + c$$
$$b < a + c$$
$$c < a + b$$

Based on this, we generate an array representing the coordinates of many of the three points of the triangle.

```
import numpy as np
faces = np.random.randint(0, 1000, (100,3))
a = faces[faces[:, 0] < faces[:,1]+faces[:,2]]
b = faces[faces[:, 1] < faces[:,0]+faces[:,2]]
c = faces[faces[:, 2] < faces[:,0]+faces[:,1]]
```

Output:

```
print(faces.shape)
print(a.shape)
print(b.shape)
print(c.shape)
```

Output:

```
(100, 3)
(83, 3)
(81, 3)
(83, 3)
```

Find the coordinates of a unique edge where multiple triangles share an edge by

```
faces = []
n = 10
for idx,(i,j,k) in enumerate(zip(a,b,c)):
    if idx == n:
        break
    if i.all() == j.all() and i.all() == k.all():
        faces.append(list(i))
print(len(faces))
print(faces)
faces = np.array(faces)
F = np.roll(faces.repeat(2,axis=1),-1,axis=1)
F = F.reshape(len(F)*3, 2)
F = np.sort(F,axis=1)
G = F.view( dtype=[('p0',F.dtype),('p1',F.dtype)] )
G = np.unique(G)


print(G)
```

Output:

```
10
[[176, 528, 687], [692, 939, 869], [606, 776, 557], [916, 852, 807], [59, 228, 402], [99, 6
452], [129, 937, 938], [505, 269, 542], [423, 188, 783], [280, 260, 469]]
[( 59, 228) ( 59, 402) ( 99, 452) ( 99, 690) (129, 937) (129, 938)
  (176, 528) (176, 687) (188, 423) (188, 783) (228, 402) (260, 280)
  (260, 469) (269, 505) (269, 542) (280, 469) (423, 783) (452, 690)
  (505, 542) (528, 687) (557, 606) (557, 776) (606, 776) (692, 869)
  (692, 939) (807, 852) (807, 916) (852, 916) (869, 939) (937, 938)]
```

77: Given a bincount of B sequence named C,

generate an array A such that np.bincount(A) == C.

```
import numpy as np
B = np.array([1, 2, 2, 2, 3])
print(B)
C = np.bincount(B)
print(C)
A = np.repeat(a=np.arange(len(C)), repeats=C)
print(A)
```

Output:

```
[1 2 2 2 3]
[0 1 3 1]
[1 2 2 2 3]
```

```
np.bincount(A)
```

Output:

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

This is achieved by np.repeat(). np.repeat can create another array that repeats the elements of the array.

You can specify the number of repetitions of each element by giving repeats as an int array as an argument. repeats are broadcast. This can be used to fit the shape of a given axis. For example, the following generates another array that repeats 0 0 times, 1 once, and 2 twice.

```
np.repeat([0, 1, 2], [0,1,2])
```

Output:

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

In the following, we define another sequence that repeats 1 once, 2 twice, and 3 three times.

```
np.repeat([1, 2, 3], [1, 2, 3])
```

Output:

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

78: Calculate the average using the sliding window on the array.

a = [0,1,... ,9].

```
import numpy as np
#Input
a = np.arange(10)
a
```

Output:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

np.cumsum() is a function that takes a cumulative sum. b=np.cumsum(a), where b is [0,(0+1),(0+1+2),... ,(0+1+2+3+4+5+6+7+8+9)]

```
# cumulative sum
b = np.cumsum(a, dtype=float)
b
```

Output:

```
array([ 0.,  1.,  3.,  6., 10., 15., 21., 28., 36., 45.])
```

In order to get the average of the sliding window, we need to convert it to an interval-by-interval sum rather than a cumulative sum. For example, the sum of every three elements can be obtained by b[3:] - b[:-3].

```
# slide window source
n = 3
print(b[n:])
print(b[:-n])
```

Output:

```
[ 6. 10. 15. 21. 28. 36. 45.]
[ 0.  1.  3.  6. 10. 15. 21.]
```

```
# Substitution
b[n:] = b[n:] - b[:-n]
```

Output:

```

```

The sequence b obtained in the last calculation has a missing interval sum that should come first. Since this is equal to b[n-1], concatenating b[n-1] with the sequence b[n:] - b[:-n] from the last calculation yields an array that sums

in the sliding window on top of sequence a. Divide this by n to get the average.

```
# Take an average(mean)
print(b[n-1:]/n)
```

Output:

```
[1. 2. 3. 4. 5. 6. 7. 8.]
```

```
# As a whole
def moving_average(a, n=3) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n
Z = np.arange(5)
print(Z)
print(moving_average(Z, n=2))
```

Output:

```
[0 1 2 3 4]
[0.5 1.5 2.5 3.5]
```

79: Given a one-dimensional array of type int, get a three-gram (trigram) as a two-dimensional array.

The following is a hard-coded confirmation of the idea.

```
import numpy as np
Z = np.array([1, 2, 3, 4, 5, 6])
print(np.vstack((Z[0:3], Z[1:4], Z[2:5], Z[3:])))
```

Output:

```
[[1 2 3]
 [2 3 4]
 [3 4 5]
 [4 5 6]]
```

The following is a for-loop expression. This is sufficient if the one-dimensional array is not so large.

Here are the key points.
(1) The slicing must be taken out three at a time.
(2) The end decision should be set by len(Z)-2 so that i+3 does not exceed the list range.

```
l = []
for i in range(len(Z)-2):
  l.append(Z[i:i+3])
print(np.array(l))
```

You can use numpy.lib.stride_tricks.as_strided() to speed up the same process as above if you want efficiency. If the size of the array to be handled is large, use this method.

```
from numpy.lib import stride_tricks

def rolling(a, window):
    shape = (a.size - window + 1, window)
    strides = (a.itemsize, a.itemsize)
    return stride_tricks.as_strided(a, shape=shape, strides=strides)
Z = rolling(np.array([1,2,3,4,5,6]), 3)
print(Z)
```

Output:

```
[[1 2 3]
 [2 3 4]
 [3 4 5]
 [4 5 6]]
```

(Chapter 9) Numpy Batch Processing

80: Output the negation of a boolean element array. Also, do a sign inversion of the floating-point array.

```
import numpy as np
Z = np.array([True, False, True])
print(Z)
np.logical_not(Z, out=Z)
print(Z)
Z = np.random.randint(0, 2, 20)
print(Z)
np.logical_not(Z, out=Z)
print(Z)
Z = np.random.uniform(-1.0, 1.0, 20)
print(Z)
np.negative(Z, out=Z)
print(Z)
```

Output:

```
[True False  True]
[False  True False]
[0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 0]
[1 0 1 1 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1]
[ 0.84842774 -0.95504515 -0.95587627 -0.02348855 -0.15140248 -0.65198257
  -0.91105411  0.32985491 -0.47127542 -0.75243955  0.6365752   0.68818052
  -0.09576728  0.63625954 -0.48643856  0.31726615 -0.59815515  0.68796083
  -0.64255442  0.60884036]
[-0.84842774  0.95504515  0.95587627  0.02348855  0.15140248  0.65198257
   0.91105411 -0.32985491  0.47127542  0.75243955 -0.6365752  -0.68818052
   0.09576728 -0.63625954  0.48643856 -0.31726615  0.59815515 -0.68796083
   0.64255442 -0.60884036]
```

81: Consider two points P0, P1 and a set of

points p that represent a line in two-dimensional space and calculate the distance from p to each line i (P0[i], P1[i]).

```
import numpy as np

P0 = np.array([[1,5],[3,3]])
P1 = np.array([[0,0],[2,10]])
p = np.array([5,5])

def get_line_equation(points):
  p0 = points[0]
  p1 = points[1]
  a = (p1[1] - p0[1]) / (p1[0] - p0[0])
  b = -1
  c =  p0[1] - a * p0[0]
  return a,b,c

def get_pl_distance(line, point):
  a = line[0]
  b = line[1]
  c = line[2]
  x0 = point[0]
  y0 = point[1]
  return np.abs(a * x0 + b * y0 + c)/np.sqrt(a**2+b**2)

line0 = get_line_equation(P0)
print(get_pl_distance(line0, p))

line1 = get_line_equation(P1)
print(get_pl_distance(line1, p))
```

Output:

```
2.82842712474619
```

In order to solve this problem, you need to know the following formula.

Formula for finding the equation given two points $(x_0, y_0), (x_1, y_1)$

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

leads to $a = \frac{y_1 - y_0}{x_1 - x_0}$ and $b = -1$と$c = -ax_0 + y_0$.

The distance between straight line $ax + bx + c$ and point (x_0, y_0) can be lead by

$$\frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

To reduce the amount of description, I used Ellipsis for the following code.

```
def distance(P0, P1, p):
    T = P1 - P0
    L = (T**2).sum(axis=1)
    U = -((P0[:,0]-p[...,0])*T[:,0] + (P0[:,1]-p[...,1])*T[:,1]) / L
    U = U.reshape(len(U), 1)
    D = P0 + U*T - p
    return np.sqrt((D**2).sum(axis=1))

P0 = np.array([[1, 5], [3, 3]])
P1 = np.array([[0, 0], [2, 10]])
p = np.array([5, 5])
print(distance(P0, P1, p))
```

Output:

```
[3.9223227 2.2627417]
```

82: Consider an arbitrary array and

extract a sub-array whose shape is fixed around a given element. (0padding if necessary.)

```
# original sequence
Z =  np.arange(1, 26).reshape(5, 5)
# The center of the subsequence
center =(2, 2)
# The shape of the sub-array
shape =(3, 3)
# The first line of the subsequence
r_start =center[0] - shape[0]//2
# End row of the sub-array
r_end =center[0] +shape[0]//2 + 1
# The starting sequence of the sub-array
c_start =center[1] - shape[1]//2
# end column of a sub-array
c_end =center[1] +shape[1]//2 + 1
# 0 Padding
pad =[]
padding =0
if r_start <0:
  pad.append(np.abs(r_start))
  r_start =0
  r_end +=np.abs(r_start) + 1
if c_start <0:
  pad.append(np.abs(c_start))
  c_start =0
  c_end +=np.abs(c_start) + 1
if pad:
  padding =np.max(pad)
# Output
print(f"original array:¥n{Z}")
print(f"center index:¥n{center}")
print(f"subpart shape:¥n{shape}")
Z =np.pad(Z, pad_width=padding)
print(f"subpart array:¥n{Z[r_start:r_end, c_start:c_end]}")
```

A more Numpy-like solution is as follows

```
Z = np.arange(1, 26).reshape(5, 5)
shape = (3, 3)
fill  = 2
position = (2,2)

R = np.ones(shape, dtype=Z.dtype)*fill
P  = np.array(list(position)).astype(int)
Rs = np.array(list(R.shape)).astype(int)
Zs = np.array(list(Z.shape)).astype(int)

R_start = np.zeros((len(shape),)).astype(int)
R_stop  = np.array(list(shape)).astype(int)
Z_start = (P-Rs//2)
Z_stop  = (P+Rs//2)+Rs%2

R_start = (R_start - np.minimum(Z_start,0)).tolist()
Z_start = (np.maximum(Z_start,0)).tolist()
R_stop = np.maximum(R_start, (R_stop - np.maximum(Z_stop-
Zs,0))).tolist()
Z_stop = (np.minimum(Z_stop,Zs)).tolist()

r = [slice(start,stop) for start,stop in zip(R_start,R_stop)]
z = [slice(start,stop) for start,stop in zip(Z_start,Z_stop)]
R[r] = Z[z]
print(Z)
print(R)
```

Output:

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

83: Calculate the matrix rank.

numpy.linalg contains a number of useful functions for linear algebra, such as np.linalg.matrix_rank(), which makes it easy to find the matrix rank.

```
import numpy as np
Z = np.random.uniform(0, 1, (10, 10))
print(np.linalg.matrix_rank(Z))
```

Output:

```
10
```

The number of non-zero vectors of S obtained by singular value decomposition by np.linalg.svd() is the matrix rank of the original matrix.

```
# SVD
U, S, V = np.linalg.svd(Z) # Singular Value Decomposition
rank = np.sum(S > 1e-10)
print(rank)
```

Output:

```
10
```

By the way, SVD (Singular Value Decomposition) is a type of matrix decomposition, often used for dimensionality reduction applications, where Z and U*S@V are equal.

```
Z
```

Output:

```
array([[0.17452443, 0.53317398, 0.76292137, 0.28131551, 0.54085357,
        0.06551601, 0.31270761, 0.62242133, 0.91230656, 0.9472987 ],
       [0.41085101, 0.54233488, 0.02733112, 0.43369666, 0.85254499,
        0.40478753, 0.7293474 , 0.53459287, 0.61549601, 0.6466823 ],
       [0.01124159, 0.94516998, 0.80784951, 0.63538497, 0.67283477,
        0.54718083, 0.98332937, 0.52240187, 0.69809813, 0.47412896],
       [0.9245148 , 0.48761133, 0.68587766, 0.80932098, 0.8548447 ,
        0.06054501, 0.98352713, 0.35265272, 0.36927536, 0.29196765],
       [0.76788255, 0.10455337, 0.48728792, 0.16426604, 0.67728011,
        0.83783513, 0.3066827 , 0.45130992, 0.73439357, 0.31305585],
       [0.85084006, 0.3858666 , 0.98102577, 0.29747122, 0.45368769,
        0.65099858, 0.23535423, 0.7355205 , 0.76344527, 0.64746627],
       [0.76475884, 0.35484245, 0.67513718, 0.91257135, 0.23866977,
        0.48665112, 0.69785581, 0.359934  , 0.79671053, 0.28359138],
       [0.28715849, 0.80462372, 0.49645375, 0.82327737, 0.84464555,
        0.60909916, 0.24189853, 0.90291116, 0.18757124, 0.32971499],
       [0.00426189, 0.30231965, 0.91178877, 0.46530449, 0.87396665,
        0.32747678, 0.7075492 , 0.5673174 , 0.17583044, 0.74158712],
       [0.32272947, 0.76428078, 0.45095563, 0.31528919, 0.06653964,
        0.47666953, 0.26825947, 0.06667143, 0.14299401, 0.96137395]])
```

```
U*S@V
```

Output :

```
array([[0.17452443, 0.53317398, 0.76292137, 0.28131551, 0.54085357,
        0.06551601, 0.31270761, 0.62242133, 0.91230656, 0.9472987 ],
       [0.41085101, 0.54233488, 0.02733112, 0.43369666, 0.85254499,
        0.40478753, 0.7293474 , 0.53459287, 0.61549601, 0.6466823 ],
       [0.01124159, 0.94516998, 0.80784951, 0.63538497, 0.67283477,
        0.54718083, 0.98332937, 0.52240187, 0.69809813, 0.47412896],
       [0.9245148 , 0.48761133, 0.68587766, 0.80932098, 0.8548447 ,
        0.06054501, 0.98352713, 0.35265272, 0.36927536, 0.29196765],
       [0.76788255, 0.10455337, 0.48728792, 0.16426604, 0.67728011,
        0.83783513, 0.3066827 , 0.45130992, 0.73439357, 0.31305585],
       [0.85084006, 0.3858666 , 0.98102577, 0.29747122, 0.45368769,
        0.65099858, 0.23535423, 0.7355205 , 0.76344527, 0.64746627],
       [0.76475884, 0.35484245, 0.67513718, 0.91257135, 0.23866977,
        0.48665112, 0.69785581, 0.359934  , 0.79671053, 0.28359138],
       [0.28715849, 0.80462372, 0.49645375, 0.82327737, 0.84464555,
        0.60909916, 0.24189853, 0.90291116, 0.18757124, 0.32971499],
       [0.00426189, 0.30231965, 0.91178877, 0.46530449, 0.87396665,
        0.32747678, 0.7075492 , 0.5673174 , 0.17583044, 0.74158712],
       [0.32272947, 0.76428078, 0.45095563, 0.31528919, 0.06653964,
        0.47666953, 0.26825947, 0.06667143, 0.14299401, 0.96137395]])
```

84: Find the mode of the sequence.

```
import numpy as np
```

```
a = np.array([1,2,3,4,4,5])
print(np.argmax(np.bincount(a)))
```

Output:

```
4
```

85: Extract all consecutive 3x3 blocks from a random 10x10 matrix.

```
import numpy as np
a = np.random.random((10, 10))
print(a)
n = 3
i = (a.shape[0]-n) + 1
j = (a.shape[1]-n) + 1
c = np.lib.stride_tricks.as_strided(a, shape=(i, j, n, n),
strides=a.strides + a.strides)
print(c)
```

Output:

```
[[0. 73854811  0. 23797304  0. 31476473  0. 32105891  0. 82017286
0. 86924965
  0. 94245105  0. 09913214  0. 33551643  0. 19789798]
…
  [[0. 1126378   0. 4985062   0. 99706545]
   [0. 25365781  0. 69037875  0. 47259774]
   [0. 13971084  0. 89357712  0. 77723999]]]
```

*Output is omitted for brevity.

86: Find a two-dimensional array in which the

rows and columns of the two-dimensional array Z are swapped. (That is, a two-dimensional array with Z[i,j] == Z[j,i])

```
import numpy as np
Z = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(Z)
print(Z.transpose())
```

Output:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

```
print(Z)
Z_sub = np.zeros_like(Z)
for i in range(Z.shape[0]):
  for j in range(Z.shape[1]):
    Z_sub[i, j] = Z[j,i]
print(Z_sub)
```

Output:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

87: Consider a set of p matrices whose shape is (n,n) and a set of p vectors whose shape is (n,1). How do you calculate the sum of the products of p matrices at once? (The result has the shape (n,1)

```
#p, n = 1, 3
a = np.array([[[1,2,3],[4,5,6],[7,8,9]]])
b = np.array([[[1],[1],[1]]])
print(a.shape)
print(b.shape)
np.sum(a@b, axis=0)
```

Output:

```
(1, 3, 3)
(1, 3, 1)
array([[ 6],
       [15],
       [24]])
```

You can specify an axis in "@" (inner product calculation). You can calculate the sum of products of p matrices at once by specifying the axis 0. You can do the same in np.tensordot().

```
a = np.array([[[1,2,3],[4,5,6],[7,8,9]]])
b = np.array([[[1],[1],[1]]])
np.tensordot(a, b, axes=[[0, 2], [0, 1]])
```

Output:

```
array([[ 6],
       [15],
       [24]])
```

88: Consider a 16x16 array, get the sum of a sub-array of block size 4x4.

```
import numpy as np

a = np.ones((16,16))
k = 4
print(np.add.reduceat(np.add.reduceat(a, np.arange(0, a.shape[0], k),
axis=0),
                                      np.arange(0, a.shape[1], k),
axis=1))
```

Output:

```
[[16. 16. 16. 16.]
 [16. 16. 16. 16.]
 [16. 16. 16. 16.]
 [16. 16. 16. 16.]]
```

Reduces with a specified slice on one axis in np.add.reduceat. This is done by specifying the block size on axes 0 and 1, respectively.

89: How do I implement Game of Life using numpy arrays?

```
def iterate(Z):
    # Live Cell Counting
    N = (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
         Z[1:-1,0:-2]                + Z[1:-1,2:] +
         Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])

    # The Birth of the Next Generation
    birth = (N==3) & (Z[1:-1,1:-1]==0)
    # Continuing the Next Generation
    survive = ((N==2) | (N==3)) & (Z[1:-1,1:-1]==1)
    # Initializing the return cell
    Z[...] = 0
    # Flagging of survival cells
    Z[1:-1,1:-1][birth | survive] = 1
    return Z

Z = np.random.randint(0,2,(50,50))
for i in range(5):
  Z = iterate(Z)
  print(Z)

import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print(a[:, 2])
print(a[...,2])
print(a[Ellipsis,2])

Ellipsis?
```

Output:

```
[[0 0 0 ... 0 0 0]
  [0 1 0 ... 0 0 0]
  [0 1 1 ... 0 0 0]
  ...
  [0 0 0 ... 1 1 0]
  [0 0 1 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
[[0 0 0 ... 0 0 0]
  [0 1 1 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 1 1 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 1 1 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
[[0 0 0 ... 0 0 0]
  [0 0 1 ... 0 0 0]
  [0 0 1 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
```

Conway's Game of Life (Conway's Game of Life[1]) is a simulation game invented by the English mathematician John Horton Conway in 1970 that uses a simple model to recreate the process of birth, evolution, and selection of life. It follows the following rules.

1) Birth: if there are exactly three living cells adjacent to a dead cell, the next generation is born.

2) Survival: If there are two or three living cells adjacent to a living cell, it will survive in the next generation.

3) Overpopulation: if there is less than one living cell adjacent to a living cell, it dies due to overpopulation.

4) Overcrowding: if there are four or more living cells adjacent to a living cell, it dies due to overcrowding. Below is an example of life and death at the next step in the middle cell. Living cells are represented by ■ and dead cells by □.

Implementing this would look like the code above (see comments for details). The "..." that appears a few times stands for the Python built-in Ellipsis (abbreviation). is a special value used mainly in the extended slice syntax and user-defined container data types (official reference), but is recommended for numpy's multi-dimensional array slicing as it's easier to write in a concise way if you know it.

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print(a[:, 2])
print(a[...,2])
print(a[Ellipsis,2])
```

Output:

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

(Chapter 10) Numpy Cornering

90: Get the nth largest number in the array where the numbers are stored.

```
import numpy as np
a = np.array([5,4,3,1,2])
def max_n(a, n):
  return a[np.argsort(a)[-n]]
max_n(a, 2)
```

Output:

```
4
```

91: Given any number of vectors, find the direct product (all combinations of all items).

```
import numpy as np
def direct_product(arrays):
    arrays = [np.asarray(a) for a in arrays]
    shape = (len(x) for x in arrays)

    ix = np.indices(shape, dtype=int)
    ix = ix.reshape(len(arrays), -1).T

    for n, arr in enumerate(arrays):
        ix[:, n] = arrays[n][ix[:, n]]

    return ix

print (direct_product(([1, 2, 3], [4, 5], [6, 7])))
```

Output:

```
[[1 4 6]
 [1 4 7]
 [1 5 6]
 [1 5 7]
 [2 4 6]
 [2 4 7]
 [2 5 6]
 [2 5 7]
 [3 4 6]
 [3 4 7]
 [3 5 6]
 [3 5 7]]
```

For two sets A and B, the set made up of all the pairs made from one element of A and one element of B is called the direct product set. The above is implemented in a foolproof way.

Using itertools.product, you can implement direct product in a simple and fast way.

```
import itertools
cartesian = itertools.product([1,2,3], [4,5], [6, 7])
for tpl in cartesian:
  print(tpl)
```

Output:

```
(1, 4, 6)
(1, 4, 7)
(1, 5, 6)
(1, 5, 7)
(2, 4, 6)
(2, 4, 7)
(2, 5, 6)
(2, 5, 7)
(3, 4, 6)
(3, 4, 7)
(3, 5, 6)
(3, 5, 7)
```

92: Generate a recalay from ndarray.

```
import numpy as np

Z = np.array([("Hello", 2.5, 3),
              ("World", 3.6, 2)])
R = np.core.records.fromarrays(Z.T,
                               names='col1, col2, col3',
                               formats = 'S8, f8, i8')
print(type(Z), Z)
print(type(R), R)
```

Output:

```
<class 'numpy.ndarray'>[['Hello' '2.5' '3']
 ['World' '3.6' '2']]
<class 'numpy.recarray'>[(b'Hello', 2.5, 3) (b'World', 3.6, 2)]
```

Record arrays can expose structured array fields as properties. ndarray also supports named fields, but the difference is that recarray can use attribute references to refer to the fields.

You can also use view() to convert it.

```
a = np.array([('Hi', 1), ('NLP', 2)], dtype=[('X', 'S3'), ('Y',
np.int)])
b = a.view(np.recarray)
print(b.X)
print(b.Y)
```

Output:

```
[b'Hi' b'NLP']
[1 2]
```

Replace dtype=[('X','S3'), with dtype=[('X','S2 '), the string that can be stored in X will be truncated by two characters.

```
a = np.array([('Hi', 1), ('NLP', 2)], dtype=[('X', 'S2'), ('Y',
np.int)])
b = a.view(np.recarray)
print(b.X, b.Y)
```

Output:

```
[b'Hi' b'NL'] [1 2]
```

This is what happens when you generate a recarray with a single line.

```
a=np.array([('Hi', 1), ('np', 2)],dtype=[('X','S2'),('Y',
np.int)]).view(np.recarray)
print(a.X, a.Y)
```

Output:

```
[b'Hi' b'np'] [1 2]
```

93: Consider a large vector Z (ten million random numbers) and calculate the cube using four different methods.

```
import numpy as np

# 10 million random values
 x = np.random.rand(int(10**7))

%timeit np.power(x, 3)
%timeit np.einsum('i,i,i->i', x, x, x)
%timeit x**3
%timeit x*x*x
```

Output:

```
1 loop, best of 3: 740 ms per loop
10 loops, best of 3: 27.1 ms per loop
1 loop, best of 3: 740 ms per loop
10 loops, best of 3: 29.9 ms per loop
```

np.einsum (Einstein's contraction notation) is the fastest; in Jupyter Notebook, type np.einsum? to see an example of einsum.

94: Consider two arrays A and B of the form (8,3) and (2,2); how do you find a row of A that contains an element in each row of B, regardless of the order of the elements in B?

```
A = np.random.randint(0, 5, (8, 3))
B = np.random.randint(0, 3, (2, 2))
l = []
for i, j in enumerate(A):
  if len(set(j) & set(B.flatten())) > 0:
    l.append(i)
print(f'A: {A}')
print(f'B: {B}')
print(f'Row{l} A containing the elements of each row of B')

C = (A[..., np.newaxis, np.newaxis] == B)
rows = np.where(C.any((3, 1)).all(1))[0]
print(rows)
```

Output:

```
A: [[2 4 3]
 [1 3 4]
 [4 1 1]
 [0 3 0]
 [2 0 3]
 [4 3 0]
 [2 2 0]
 [1 1 4]]
B: [[2 1]
 [1 0]]
Row[0, 1, 2, 3, 4, 5, 6, 7] A containing the elements of each row o
[1 2 4 6 7]
```

Written more numpy-like, it looks like the following.

```
C = (A[..., np.newaxis, np.newaxis] == B)
rows = np.where(C.any((3,1)).all(1))[0]
print(rows)
```

Output:

```
[0 1 2 3 4 7]
```

Extract the rows in a 95:10x3 array that are not all equal (e.g., [2,2,3]).

First, prepare a 10 x 3 array.

```
import numpy as np

Z = np.array(
 [[1, 0, 3],
 [3, 3, 0],
 [4, 4, 4],
 [1, 4, 0],
 [2, 4, 1],
 [2, 3, 3],
 [0, 0, 0],
 [1, 2, 2],
 [1, 1, 1],
 [3, 3, 3]]
)
print(Z)
```

Output:

```
[[1 0 3]
 [3 3 0]
 [4 4 4]
 [1 4 0]
 [2 4 1]
 [2 3 3]
 [0 0 0]
 [1 2 2]
 [1 1 1]
 [3 3 3]]
```

```
Z.shape
```

Output:

```
(10, 3)
```

A straightforward for-loop solution of the subject would be as follows.

```
# Steadily
l = []
for row in Z:
  if row[0] == row[1] == row[2]:
    print('excluded!')
  else:
    l.append(row)
print(np.array(l))
```

Output:

```
excluded!
excluded!
excluded!
excluded!
[[1 0 3]
 [3 3 0]
 [1 4 0]
 [2 4 1]
 [2 3 3]
 [1 2 2]]
```

If the maximum and the minimum are different among the three columns, all three columns will be different, as shown below. However, please note that a non-numeric type will cause an error.

```
#Only numeric types are supported.
 U = Z[Z.max(axis=1) != Z.min(axis=1),:]
print(U)
```

Output:

```
[[1 0 3]
 [3 3 0]
 [1 4 0]
 [2 4 1]
 [2 3 3]
 [1 2 2]]
```

The following code is written in a numpy-like manner and supports all data types.

```
# All data types are supported
 # numpy like

E = np.all(Z[:,1:] == Z[:,:-1], axis=1)
U = Z[~E]
print(U)



Z2 = np.array([[True, True, False],[True, True, True]])
E = np.all(Z2[:,1:] == Z2[:,:-1], axis=1)
U = Z2[~E]
print(U)
```

Output:

```
[[1 0 3]
 [3 3 0]
 [1 4 0]
 [2 4 1]
 [2 3 3]
 [1 2 2]]
[[ True  True False]]
```

96: Convert a vector of type int to a 01 representation.

```
import numpy as np
l = np.array([0, 1, 2, 3, 15, 16, 32, 64, 128])

def padding_zero(a, length):
  res = []
  diff = length - len(a)
  #print(f"diff:{diff}")
  if diff < 0:
    return a

  for i in range(diff):
    res += [0]

  for i in a:
    res += [i]

  return res

l = []
for e in l:
  #print(e)
  a = bin(e)
  #print([int(i) for i in a if i != 'b'][1:])
  #print(padding_zero([int(i) for i in a if i != 'b'][1:],
len(bin(max(l))) - 2))
  l.append(padding_zero([int(i) for i in a if i != 'b'][1:],
len(bin(max(l))) - 2))
print(np.array(l))
```

Output:

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 1 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]]
```

You can use bin() to convert the integer value to 01 representation (binary), but you need 0padding to store it as an element in np.ndarray. The above code is a simple implementation of this, and although I've commented out the output of print(), it may help the reader to understand it better if I delete the comment # from the code as needed.

By the way, if you type np.ubyte, this can be written in one line.

```
I = np.asarray(I, dtype=np.ubyte)
print(np.unpackbits(I[:, np.newaxis], axis=1))
```

Output:

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 1 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]]
```

97: Given a two-dimensional array, extract a unique row.

```
import numpy as np
Z = np.array([[1,2,3],[1,2,3],[4,5,6]])
ZZ = np.unique(Z, axis=0)
print(ZZ)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

98: Considering two vectors A and B, write the

subscript of the einsum of the inner product, outer product, summation, and mul function.

```
import numpy as np
A = np.array([1, 3, 5, 7, 9])
B = np.array([2, 4, 6, 8, 10])

print(np.einsum('i->', A))          # np.sum(A)
print(np.einsum('i,i->i', A, B))    # A * B
print(np.einsum('i,i', A, B))       # np.inner(A, B)
print(np.einsum('i,j->ij', A, B))   # np.outer(A, B)
```

Output:

```
25
[ 2 12 30 56 90]
190
[[ 2  4  6  8 10]
 [ 6 12 18 24 30]
 [10 20 30 40 50]
 [14 28 42 56 70]
 [18 36 54 72 90]]
```

99: Consider a two-dimensional array Xk= [[x1,y1],[x2,y2]] representing two coordinates. Find the Euclidean distance between the two points of X1=[[0,0],[1,1]] and X1=[[1,1], [2,2]], respectively.

```
x_1 = np.array([[0, 0], [1, 1]])
x_2 = np.array([[1, 1], [2, 2]])

z = np.linalg.norm(x - y, axis=1)
z
```

Output:

```
array([1.41421356, 1.41421356])
```

100: Given an integer n and a two-dimensional array X, select a row that can be interpreted as being drawn from X from an n-degree polynomial distribution, i.e., a row that contains only integers and sums to n.

```
import numpy as np
X = np.array([[1.0, 0.0, 3.0, 8.0],
              [2.0, 0.0, 1.0, 1.0],
              [1.5, 2.5, 0.0, 0.0]])
n = 4
M = np.logical_and.reduce(np.mod(X, 1) == 0, axis=-1)
M &= (X.sum(axis=-1) == n)
print(X[M])
```

Output:

```
[[2. 0. 1. 1.]]
```

The part of the statement that doesn't make sense at first glance, i.e., a row that contains only integers and totals to n, is key. First, np.logical_and.reduce(np.mod(X, 1) == 0, axis=-1), extracting only those rows where all the elements are integers (i.e., divide by 1 to get 0), and then M &= (X.sum(axis=-1) == n) to see if the sum of each element is n The solution is derived by fancy indexing by X[M] by extracting an array of true and false values of

101: Compute the bootstrapped 95% confidence interval of the mean of the 1D array X (i.e., replace the elements of the array N times and resample, compute the mean of each sample, and compute the percentile for

that mean).

```
import numpy as np
X = np.array([3, 1.4, 1.5, 92, 6.5])
N = 100
idx = np.random.randint(0, X.size, (N, X.size))
means = X[idx].mean(axis=1)
confint = np.percentile(means, [2.5, 97.5])
print(confint)
```

Output:

```
[1.4695 56.8   ]
```

We will discuss the bootstrap confidence interval. First of all, the bootstrap method is a method that allows us to estimate a statistic that is theoretically difficult to derive by a simple resampling from an empirical distribution. The bootstrap method also allows us to obtain confidence intervals for the estimated statistic, and such confidence intervals are called bootstrap confidence intervals. Since it is called an empirical distribution, the results of the run will change each time.

It is possible to solve the problem without understanding the bootstrapping method in detail, i.e., from the following problem statement.

Conclusion

Thank you very much for your 101 training. Thank you for reading this book to the end. There is a reason why there were 101 problems instead of 100. My favorite word is "A small difference is a big difference". It means that a little bit of effort and ingenuity can lead to a big difference in results. If the result you get with 100% effort is 1.0, if you continue this effort every year for 80 years, 1.0 x 1.0 x ...(1.0^80) remains at 1.0. However, if you put in 101% effort slightly above your limit, 1.01 x 1.01 x ...(1.01^80) is

2.2167... That's the number. That means that just 1% more effort than others every time you do it 80 times will make twice as much difference. If this were a 105% effort, 80 times that number would be 49.56... For those of you who have solved the 101 questions, I encourage you to keep studying. I'll be sending out daily updates on the results of my own study on Twitter, so if you'd like to follow me, please do so.

Twitter account: Joshua K. Cage @JoshuaKCage1

Finally, we would like to thank Rougier for providing us with Numpy 100 exercises under an MIT License.